A Functional Flow Framework for Cloud Computing

by

Amy Zhang

B.S., EECS, Massachusetts Institute of Technology (2011)
B.S., Mathematics, Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
May 29, 2012

Certified by .....
Muriel Médard
Professor of EECS
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A Functional Flow Framework for Cloud Computing

by

Amy Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 29, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering

Abstract

This thesis covers a basic framework to calculate the maximum computation rate of a set of functions over a network. These functions are broken down into a series of computations, which are distributed among nodes of the network, with the output sent to the terminal node. We analyze two models with different types of computation costs, a linear computation cost model and a maximum computation cost model. We show how computation distribution through the given network changes with different types of computation and communication limitations. This framework can also be used in cloud design, where a network of given complexity is designed to maximize computation rate for a given set of functions. We provide a greedy algorithm that provides one solution to this problem, and create simulations for each framework, and analyze the results.

Thesis Supervisor: Muriel Médard
Title: Professor of EECS

Acknowledgments

I would like to thank my thesis advisor Muriel Médard for her encouragement and guidance through all of my work. I would also like to thank Soheil Feizi for valuable help and insight, and being there for every little problem and all the questions. I would like to thank Dr. Michael Kilian for helpful discussions on practical issues in cloud computing applications and for his help in creating the parallel computing framework for my UAP work. Finally, I would like to thank my parents for all their support.

Contents

List of Figures

Chapter 1

Introduction

With the advent of cloud computing, many problems need to be addressed to keep the cloud secure, reliable, and low-cost. Much research has been conducted on possible protocols at a system level for cloud computing to minimize problems in privacy, reliability, and accountability, but research on a theoretical level has been more scarce. The goals of my project are to research and implement algorithms to both calculate and minimize access rates to the cloud under certain models, and to investigate the trade offs between communication and computation costs.

Our goal is to be able to design and analyze a network flow framework by modeling the amount of computation in each node of the network and using convex optimization methods to calculate the maximum computation rate and route flows through the network to achieve that rate. We consider an approach with given computation and communication limitations that affect the maximum computation rate. The two computation models we examine are a maximum computation cost (MCC) model and a linear computation cost (LCC) model. In the MCC model, the computation cost over the network is a function of the maximum computation, over all nodes in the network. In the LCC model, the computation cost in each node is a linear function of the amount of computation in that node.

We also propose an algorithm to design a cloud network under given communication and computation constraints. For a given set of functions, we can find a network topology that maximizes the computation rate under the given computation

and communication constraints. We explore a greedy algorithm that designs a cloud network with a given network complexity, which we measure as the number of edges in the network. We compare its performance with a random algorithm, which outputs a network topology of the given complexity by examining the trade-off between network complexity and computation rate for both algorithms.

This thesis is organized as follows.

Chapter 1 provides an introduction to the topic of cloud computing and outlining the problem we are trying to solve.

Chapter 2 gives a recap of both my previous work done in 6.UAP and related work in the field.

Chapter 3 provides the framework we use for the problem. It is an adaptation of the framework used by Shah *et al.* [8], where we assume we are given a network and a set of functions to be computed over that network, and analyze its performance.

Chapter 4 introduces the Maximum Computation Cost (MCC) model, where the computation cost over the network is proportional to the largest total self-flows over any node. In effect, we are trying to spread the load more evenly over the network if there tends to be clusters of high computation.

Chapter 5 introduces the Linear Computation Cost (LCC) model, where we place computation costs on specific nodes in the network, proportional to the amount of self-flow through each node.

Chapter 6 discusses a possible algorithm for network design, where we are given the set of functions to be computed and their probability distribution, and we find a network that maximizes its total computation rate.

Chapter 7 goes over the results found in Chapters 4, 5, and 6 and presents areas of future research.

Chapter 2

Previous Work

2.1 UAP work

My UAP work was to use estimated graph entropies to find theoretical upper and lower bounds on computation rates. As an introductory model, we created a simulation to test bounds on communication and computation costs, and find an algorithm that gives the optimal ratio between the two types of costs. We began by coding a cloud simulation with only a few nodes, where the source nodes and compute nodes are objects in C++. A function is decomposed into coloring functions which are assigned to nodes. We defined color functions for each object that can be used in a lookup table to find the desired function value for each node. There are also functions that compute over the subset of data received. We started off with a easily verifiable function, a sort. The first round is defined as when subsets of the source nodes are input into compute nodes.

I created a working simulation that took in an adjacency matrix of the graph and created a parallel set of the source nodes. It then recursively created the next parallel set until it encountered an empty parallel set, at which point it returned the solution. We also put in an on-off matrix that indicated which node data is output to, for cases where a node has two children it can potentially send data to and we only want to output to one to avoid repetition. A parallel set is a set of jobs to be done concurrently. The structure checks if there are threads available in the threadpool,

and if there are, it takes one and assigns one of the jobs within the set to that thread. Once all the jobs in that parallel set have been assigned and completed, the parallel set returns the solution.

In the most basic version of our model, we start off with three source nodes, each of which randomly generates a list of integers of size n. These nodes are put in the first parallel set. As each node from this set is picked up by a thread, it queries the on-off matrix for the node it is supposed to output to, and adds that node to the next parallel set. Once this has been done to every node in the first parallel set, it is empty and returns that the job is done. We then pick up the next parallel set and do the same, except now this is a set of compute nodes. We again query for the output nodes and put them in the next parallel set, but then, we also query for the input nodes, and grab the corresponding lists of data. If a node as two parent nodes putting out data to it, the data lists are combined by concatenation. The data is sorted and set as output for the next parallel set of nodes. We calculate the computation time by collecting the timestamp before and after each parallel set runs, and calculate the communication time by taking $\sum \log(n)$, where n is the size of the data set transmitted from a parent node to its child. Therefore, the first parallel set has a communication time of $3\log(n)$, because each of the three source nodes transmits n integers to a child node in the next parallel set.

2.2 Related Work

A major part of my UAP work was to investigate information theoretic rate bounds for computing a function over a network, where the input is found at the sources and the output at the receiver. Körner introduced graph entropy [6], which Feizi *et al.*[5] used to investigate graph coloring approaches to this problem. Distributed computing and in-network computation are both topics that have had significant interest, and have lead the way to network flow techniques, [2], [9]). Shah *et al.* [8] also used this framework for function computation considering only communication constraints. He simplifies cloud computing as a set of functions being computed over a network, with

a probability distribution for how often each function is called. There are jointly communication and computation limitations in a cloud computing network that need to be taken into account during design. The links between nodes have communication capacities, which are our communication constraints, and different computations require different computation power, which we represent as computation costs.

Shah provided a linear optimization method to find the maximum computation rate of a function over a network, and an algorithm to route the flows to achieve that rate [8]. The flow conservation constraint does not hold for functional flows because when a computation is done at a node, the flow coming out of that node is no longer the same as the flow coming in. However, [8] circumvented this by defining a self-flow on each node for each computation type. These self-flows make up the difference in flow when a computation is done at that node, allowing for flow conservation. These self-flows are proportional to the amount of computation done at each node, and therefore by putting a cost on these flows we can simulate computation costs. This framework we modify further as discussed in Chapter 3.

Reference [7] also considered the problem of minimum-cost multicast over coded networks with a decentralized approach, and used a max norm approach that we shall discuss in Chapter 4.

Chapter 3

Framework

3.1 Network

We are using the framework described in [8]. The network $\mathcal{N} = (V, E)$ is a directed acyclic graph, where V is the set of nodes, and E is the set of edges. Each edge (u, v) has a capacity of $c_{(u,v)}$. $\mathcal{N}$ has k source nodes, $\{n_1, \ldots, n_k\}$, that have data values $\{X_i(l)\}_{l \geq 0}$, where $X_i(l)$ belongs to a finite alphabet. For source node i, we use X_i to represent its data. In cloud computing, source nodes can be considered as datasets. In this case, X_i represents data in the dataset i. $N(v)$ is the set of neighbors of node v in the network.

The terminal node, n_t, receives the output of a set of functions from the source data. These functions are given as computation trees.

3.2 Computation Trees

A computation tree is a graph $\mathcal{G} = (\zeta, \Gamma)$, where ζ is the set of nodes, and Γ is the set of directed edges. Each directed edge θ_i represents a separate computation to be mapped to a node in the network. $\{\theta_1, \ldots, \theta_k\}$ are outgoing edges from source nodes that represent the input data and θ_t is the incoming edge to the terminal node ζ_t, representing the output of the given function the computation tree represents.

$\{\zeta_1, \ldots, \zeta_k\}$ represents the source nodes. There are many possible computation

trees (shown in Section 3.3 for a given function, and, for the scope of this problem, we assume that a computation tree is given. If the given computation tree excludes certain function mappings, those mappings will not be considered.

For a specific computation $\theta \in \Gamma$, $tail(\theta)$ and $head(\theta)$ represent the tail and the head nodes of that edge in the computation tree, respectively. A computation θ_i is a parent of a computation θ if $tail(\theta_i) = head(\theta)$. Children of a computation θ are defined similarly. We refer to the set of parents and children of computation θ as $\Phi_{\uparrow}(\theta)$ and $\Phi_{\downarrow}(\theta)$, respectively.

3.3 Mapping

A given computation tree can be mapped over a network by mapping its computations (edges) θ to different nodes of the network. A mapping M is feasible when all computations θ can be computed by the data received from their parents in the network. In other words, each computation must be mapped to a node that has a path leading from the nodes mapped to $\Phi_{\uparrow}(\theta)$.

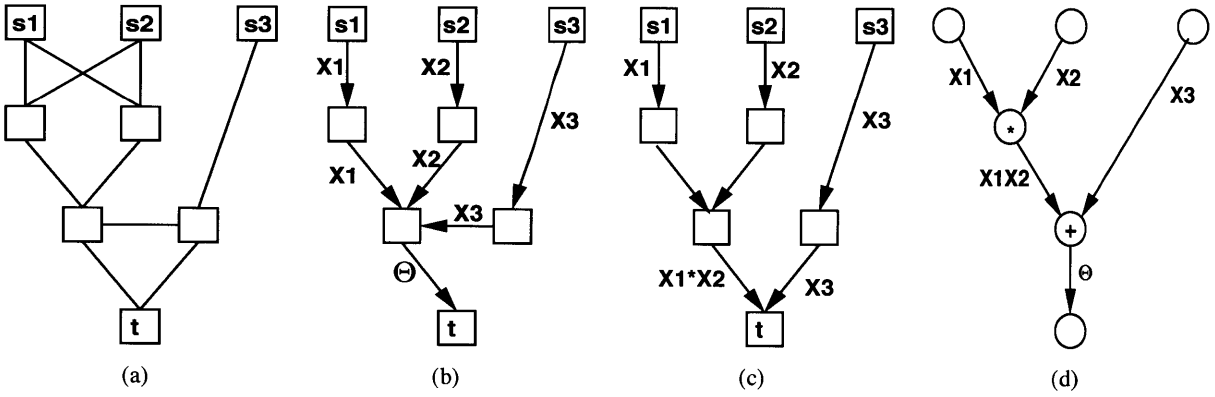


Figure 3-1: Mapping example: (a) Network $\mathcal{N}$. (b) A mapping of computation tree $\mathcal{G}$ (d) on $\mathcal{N}$. (c) A different mapping of $\mathcal{G}$ on $\mathcal{N}$. (d) A computation tree $\mathcal{G}$ for the function $\theta = X_1 * X_2 + X_3$. [8]

Figure 3.3 shows a sample network $\mathcal{N}$ and a possible computation tree for the function to be computed, $\theta = X_1 * X_2 + X_3$. Figure 3.3-(b) and (c) show possible

mappings of that computation tree on $\mathcal{N}$, one of many that can be found by our algorithm Embedding Edge LP, Figure 3-2.

Algorithm 1: Finding equivalent solution of the *Embedding-Edge LP* from a feasible solution of the *Node-Arc LP*.

input : Network graph $\mathcal{N} = (V, E)$, capacities $c(e)$, set of source nodes S, terminal node t, computation tree $\mathcal{G} = (\Omega, \Gamma)$, and a feasible solution to its *Node-Arc LP* that consists of the values of λ, $f^{\theta}_{uv}\ \forall\theta \in \Gamma, \forall uv \in E$, and $f^{\theta}_{uu}\ \forall\theta \in \Gamma,\ \forall u \in V$.

output: Solution $\{x(B)|B \in \mathcal{B}\}$ to the *Embedding-Edge LP* with $\sum_{B\in\mathcal{B}} x(B) = \lambda$.

Initialize $x(B) := 0, B(\theta_i) = \emptyset$ (the null sequence), $\forall B \in \mathcal{B}$ and $\forall\theta_i \in \Gamma, \lambda' = 0$

while $\lambda' \neq \lambda$ **do**

  $z(t) := \lambda$;

  $B(\theta_{|\Gamma|}) := t$;

  for $i := |\Gamma|$ **to** 1 **do**

    $v := B(\theta_i)$; `// valid, as` $B(\theta_i)$ `has of only one node at this step`

    $u :=$ an element in $N'(v)$ such that $f^{\theta_i}_{uv} > 0$;

    if $u \neq v$ *and* $u \in B(\theta_i)$ **then**

      `// A cycle of redundant flow found: remove the flow from all the edges in the cycle`

      Let P be the path in $B(\theta_i)$ upto the first appearance of u in it.;

      Delete P from $B(\theta_i)$. ;

      $y := \min_{u'v'\in\{uv\}\cup P} \left(f^{\theta_i}_{u'v'}\right)$;

      $f^{\theta}_{u'v'} := f^{\theta}_{u'v'} - y\ \forall u'v' \in \{uv\} \cup P$

    end

    else

      $z(u) := \min\left(z(v), f^{\theta_i}_{uv}\right)$;

    end

    if $u \neq v$ **then**

      Prefix u in $B(\theta_i)$;

      $v := u$;

      Jump to the second statement inside the **for** loop ;

    end

    else

      $B(\eta) := u, \forall\eta \in \Phi_{\uparrow}(\theta_i)$;

    end

  end

  $x(B) := z(s_1)$; `// Flow extracted on` B

  $\lambda' := \lambda' + x(B)$; `// Total flow extracted`

  `// Remove` $x(B)$ `amount of flow from all the edges in` B.

  $f^{\theta}_{u'v'} := f^{\theta}_{u'v'} - x(B)\ \forall\theta \in \Gamma$ and $\forall u'v' \in B(\theta)$;

  `// Remove` $x(B)$ `amount of flow from all the relevant self-loops.`

  $f^{\theta}_{v'v'} := f^{\theta}_{v'v'} - x(B)\ \forall\theta \in \Gamma$ and $v' = \text{start}(B(\theta))$;

end

Figure 3-2: Embedding Edge LP [8]

3.4 Algorithms

Note that, one computation tree can have several feasible mappings over a network. Define $\mathcal{M}$ as the set of all mappings of a computation tree $\mathcal{G}$ on our network $\mathcal{N} = (V, E)$. To obtain the maximum computation rate R, we need to find the set of mappings that obey capacity constraints of the network that sum to achieve R. We define R with a discrete model, where the sources put out discrete packets of data, and the terminal node receives discrete packets of the function output.

In Figure 3-4, Shah *et al.* gives a basic linear program that finds the maximum computation rate λ for a given computation tree over network $\mathcal{N}$. It also outputs the amount of flow of computation type $\theta_1, \ldots, \theta_{|\Gamma|}$, where $|\Gamma|$ is the number of computations, over every edge $e \in E$ in the network. A computation flow is the amount of data being sent over an edge, which is easier to visualize when dealing with a discrete model. f_{uv}^{θ} represents the flow from node u to v of computation type θ. $N(v)$ is defined as the set of nodes neighboring v and $N'(v)$ is the set $\{N(v) \cup v\}$.

However, the scope of our problem is beyond this linear program. We want our algorithm to be able to compute multiple functions simultaneously on the network, and to give us the maximum total rate for both, weighted with their probability distribution. The probability distribution is the probability of each function being called by the network, and therefore we want to maximize the total average computation rate $R_F = \sum_{i=1}^{n} p_i \lambda_i$. Shah's algorithm also does not take into account computation and communication costs found in cloud computing. The multiple functions with probability distribution problem is solved as shown below for a set of functions $\Theta = \{\Theta_1, \ldots, \Theta_n\}$ given computation trees $\mathcal{G}_i = (\Omega_i, \Gamma_i)$ for $i = \{1, \ldots, n\}$, where $\mathcal{G}_i$ is the given computation tree for function Θ_i. The set of functions Θ is given probability distribution $\{p_1, \ldots, p_n\}$.

Algorithm 1. max $\sum_{i=1}^{n} p_i \lambda_i$

subject to

$$f_{vv} + \sum_{u \in N(v)} f_{vu}^{\theta} - \sum_{u \in N'(v)} f_{uv}^{\theta} = 0, \forall \theta \in \Gamma_i \backslash \{\theta_{|\Gamma_i|}\}, \quad \forall \Gamma_i \in \Gamma \text{ and } \forall \eta \in \Phi_{\downarrow}(\theta). \quad (3.1)$$

$$\sum_{u \in N(v)} f_{vu}^{\theta_{|\Gamma_i|}} - \sum_{u \in N'(v)} f_{uv}^{\theta_{|\Gamma_i|}} = \begin{cases} -\lambda_i & v = t \\ 0. & \text{otherwise} \end{cases}, \quad \text{for } i \in \{1, \ldots, n\} \quad (3.2)$$

$$f_{vv}^{\theta_l} = \begin{cases} \lambda_i & v = s_l \\ 0. & \text{otherwise} \end{cases}, \quad \text{for } i \in \{1, \ldots, n\} \quad (3.3)$$

$$\sum_{\theta \in \Gamma} (f_{uv}^{\theta} + f_{vu}^{\theta}) \leq c(uv), \quad \forall uv \in E. \quad (3.4)$$

$$f_{uv}^{\theta} \geq 0, \quad \forall uv \in E, \forall \theta \in \Gamma_i \text{ and } \forall \Gamma_i \in \Gamma \quad (3.5)$$

$$f_{uu}^{\theta} \geq 0, \quad \forall u \in V, \forall \theta \in \Gamma_i \text{ and } \forall \Gamma_i \in \Gamma \quad (3.6)$$

$$\lambda_i \geq 0. \quad \text{for } i \in \{1, \ldots, n\}. \quad (3.7)$$

Different ways of combining computation and communication costs will be discussed in Chapters 4 and 5. Computation costs are costs placed on certain, or all, nodes on a network for certain computations. These can represent inefficient machines or if certain computations are heavier than others. Communication costs are the capacities of each edge, a fixed ceiling that is provided.

We will explore the first constraint, the functional conservation of flows, further. The usual definition of flow conservation does not hold in the case of function flows, as shown in Figure 3-3. When a computation involving $f_{(1,3)}^{\theta}$ and $f_{(2,3)}^{\theta}$ occurs at node 3, then

$$f_{(1,3)}^{\theta} + f_{(2,3)}^{\theta} \neq f_{(3,4)}^{\theta}$$

To achieve flow conservation, we introduce self-flows, $f_{(i,i)}^{n}$ at each node, so that our new conservation rule is,

$$f_{(1,3)}^{\theta} + f_{(2,3)}^{\theta} = f_{(3,4)}^{\theta} + f_{(3,3)}^{\eta}.$$

These self-flows represent the amount of computation done at each node, although different computations may have different amounts of self-flow. We shall use these self-flows when including computation constraints, as shown in Chapters 4 and 5.

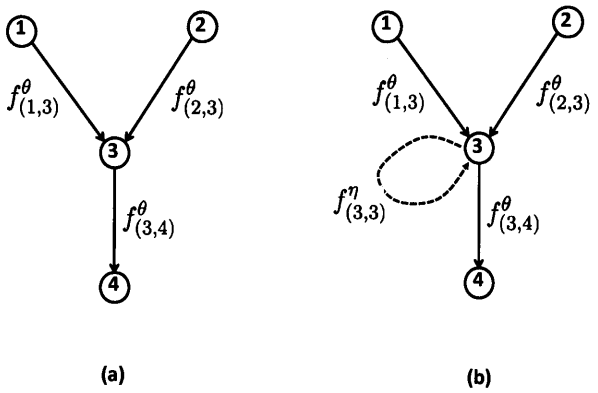


Figure 3-3: (a) An example that shows the flow conservation constraint does not hold in the function computation setup. (b) By adding a self-loop with infinite capacity to node 3, a modified flow conservation constraint holds in this case.

Furthermore, we know basic bounds for the total computation rate over a network given k sources and the min-cut rate of our network, γ.

Theorem 2. *Given a set of functions with k sources and a network $\mathcal{N}$ with min-cut rate γ, the total computation rate R_F has bounds,*

$$\frac{\gamma}{k} \leq R_F \leq \gamma$$

Proof. The lower bound represents the simplest scenario, a centralized scheme where all computations are done at the terminal node. In this case, each source can send $\frac{\gamma}{k}$ to the terminal node, following the min-cut max-flow theorem for multicast networks [1], giving us a total computation rate of $\frac{\gamma}{k}$.

The upper bound is achievable by envisioning a virtual node connected to all k sources with infinite capacity links. We also assume this virtual node has infinite computational capacity. In this case, the limiting factor is the min-cut rate of the network, γ, and again using the min-cut max-flow theorem, the upper bound of R_F is γ. □

Node-Arc LP: Maximize λ subject to following constraints any node $v \in V$

1. Functional conservation of flows:

$$f_{vv}^{\eta} + \sum_{u \in N(v)} f_{vu}^{\theta} - \sum_{u \in N'(v)} f_{uv}^{\theta} = 0, \ \forall \theta \in \Gamma \setminus \{\theta_{|\Gamma|}\} \text{ and } \forall \eta \in \Phi_{\downarrow}(\theta). \tag{3}$$

2. Conservation and termination of $\theta_{|\Gamma|}$:

$$\sum_{u \in N(v)} f_{vu}^{\theta_{|\Gamma|}} - \sum_{u \in N'(v)} f_{uv}^{\theta_{|\Gamma|}} = \begin{cases} -\lambda & v = t \\ 0. & \text{otherwise} \end{cases} \tag{4}$$

3. Generation of $\theta_l \ \forall l \in \{1, 2, \ldots, \kappa\}$:

$$f_{vv}^{\theta_l} = \begin{cases} \lambda & v = s_l \\ 0. & \text{otherwise} \end{cases} \tag{5}$$

4. Capacity constraints

$$\sum_{\theta \in \Gamma} \left(f_{uv}^{\theta} + f_{vu}^{\theta} \right) \leq c(uv), \ \forall uv \in E. \tag{6}$$

5. Non-negativity constraints

$$f_{uv}^{\theta} \geq 0, \forall uv \in E \text{ and } \forall \theta \in \Gamma \tag{7}$$
$$f_{uu}^{\theta} \geq 0, \forall u \in V \text{ and } \forall \theta \in \Gamma \tag{8}$$
$$\lambda \geq 0. \tag{9}$$

Figure 3-4: Node Arc LP [8]

Figure 3-2 details a linear program that finds the set of mappings that achieve rate λ. It does so by starting at terminal node t and exploring upwards to neighboring nodes while checking if these paths contain nonzero flow of type $\theta_{|\Gamma|}$. When the algorithm hits a node u where none of the edges to neighbors contain nonzero flow of type $\theta_{|\Gamma|}$, the algorithm assigns that computation to u, saves the path and the smallest flow through that path, and assigns u as the starting point for all computations in the set $\Phi_{\uparrow}(\theta_{|\Gamma|})$. The algorithm then repeats for computations $\theta_{|\Gamma|-1}, \ldots, \theta_1$. Every run of the `for` loop, as shown in Figure 3-5, outputs an embedding and its rate, the smallest amount of flow on each path (the bottleneck), and the algorithm only stops when it achieves the maximum computation rate given by Node Arc LP.

```
for i := |Γ| to 1 do
    v := B(θ_i) ;        // valid, as B(θ_i) has of only one node at this step
    u := an element in N'(v) such that f^{θ_i}_{uv} > 0 ;
    if u ≠ v and u ∈ B(θ_i) then
        // A cycle of redundant flow found: remove the flow from all
           the edges in the cycle
        Let P be the path in B(θ_i) upto the first appearance of u in it.;
        Delete P from B(θ_i). ;
        y := min_{u'v' ∈ {uv} ∪ P} (f^{θ_i}_{u'v'}) ;
        f^{θ}_{u'v'} := f^{θ}_{u'v'} − y ∀u'v' ∈ {uv} ∪ P
    end
    else
        z(u) := min (z(v), f^{θ_i}_{uv}) ;
    end
    if u ≠ v then
        Prefix u in B(θ_i) ;
        v := u ;
        Jump to the second statement inside the for loop ;
    end
    else
        B(η) := u, ∀η ∈ Φ_↑(θ_i) ;
    end
end
```

Figure 3-5: `for` loop in Embedding Edge LP [8]

Chapter 4

A Maximum Computation Cost Model

4.1 Description

In the Maximum Computation Cost Model, we are analyzing the behavior of computation flows over a network under certain limitations. The limitation is the maximum computation power constraint in a network. In this case, we show how our framework allows redistribution of computation flows over the network to achieve a better total computation rate. However, a max-norm is not everywhere differentiable, and therefore cannot be modeled as a convex program. Instead, we use the l^k-norm approximation [7] which converges to max-norm as $k \to \infty$. The new convex program constraints are below.

Algorithm 3.

$$\max \quad \sum_{i=1}^{n} p_i \lambda_i - \delta(\sum_{v \in V}(\sum_{\theta \in \Gamma} f_{vv}^{\theta})^p)^{\frac{1}{p}}$$

subject to equations 3.1-3.7.

δ is the maximum computation cost parameter, a non-negative weight on the maximum self-flow that controls how evenly the computation is spread through the

network.

4.2 Simulations

In our simulations, we consider two functions to be computed at the receiver:

$$f_1 = X_1 * X_2 + X_3 * X_4 + X_5 * X_6$$
$$f_2 = X_1 * X_2 * X_3 + X_4 * X_5 * X_6.$$

The computation trees for f_1 and f_2 are shown in Figures 4-1 and 4-2. There are many possible computation trees for these functions, but the trees in Figures 4-1 and 4-2 encompass all possible mappings, because all possible computation orderings are achievable with these computation trees.

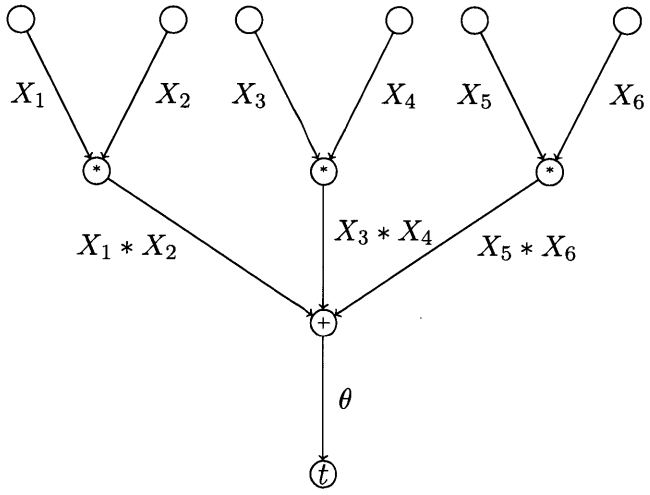


Figure 4-1: Function 1.

We demonstrate the effect of adding an approximation of Maximum Computation Cost (MCC) to our optimization problem, where we maximize the computation rate λ with an added l^k-norm cost, $\delta * (\sum_{i,j} g_{i,j}^k)^{\frac{1}{k}}$ where $g_{i,j}$ is the sum of all flow over edge (i,j) for all edges in the network and δ is a control variable. As $k \to \infty$ we are effectively putting a constraint on the maximum edge flow in the network [7].

The initial network for this simulation $\mathcal{N}_1$ is shown in Figure 4-3. All edges have

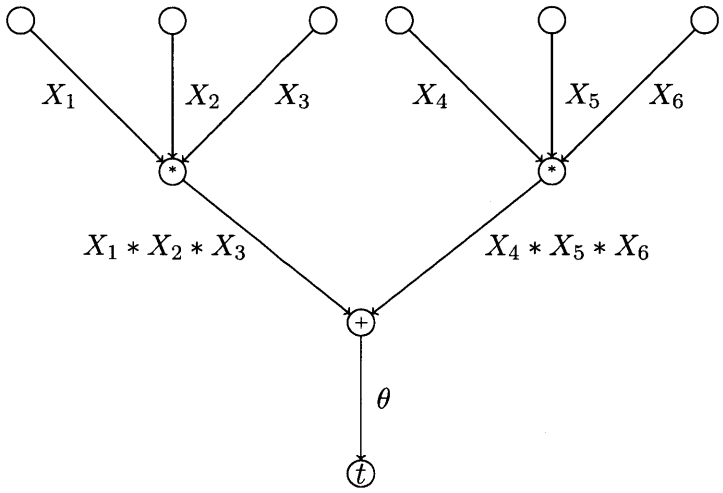


Figure 4-2: Function 2.

capacity 10 except the edge from n_{15} to the terminal node, n_t, has capacity 1. We chose edge capacities that cause flows to be denser in certain sections of the network than others. The min-cut rate of this network is therefore 11. We are computing f_1 and f_2 over the network simultaneously and with equal weights.

4.3 Results

Figure 4-4 shows the change in distribution of computation over the nodes in the network when we use the MCC constraint vs. when we do not use it. In this simulation we are using $\delta = .1$ and $k = 15$. We achieved these values through trial and error. A δ value too small did not affect flows in the network, and too large a value caused all flows to go to zero. The k is sufficiently large to provide us a reasonably accurate approximation of the max-norm. Using higher values did not change the computation costs, which means our k was able to find the maximum computation flow. We can see that the computation rate in nodes in the second layer, $\{n_7, \ldots, n_{12}\}$ increases in the model with the MCC parameter, spreading the computation load through the network more effectively than in the case without the MCC parameter.

Figure 4-5 shows the total computation rate over the network with varying δ values. When $\delta = 0$ we can achieve the maximum computation rate of 22, but as

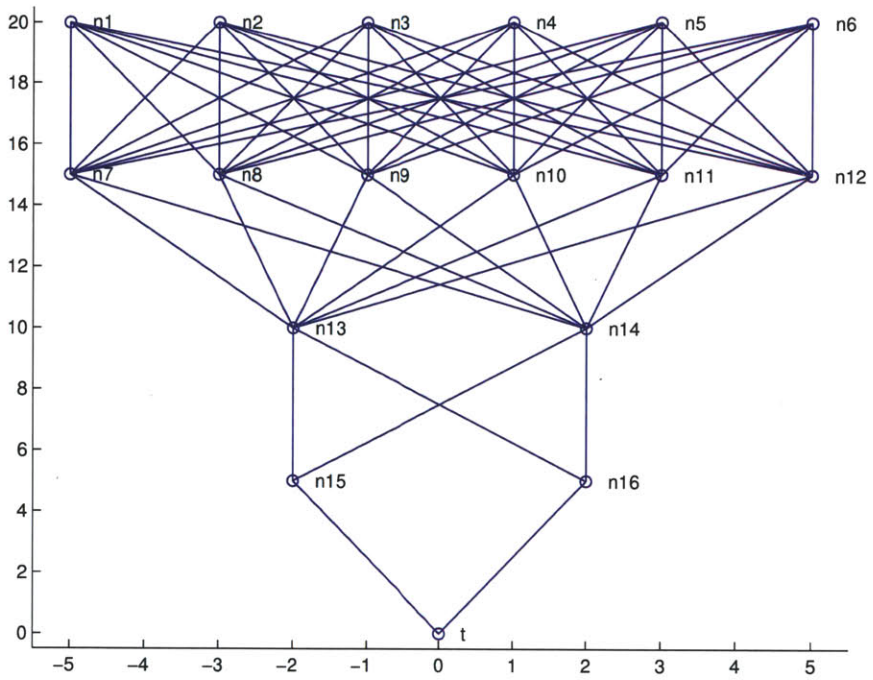


Figure 4-3: Initial communication network $\mathcal{N}_1$.

δ increases, the computation rate decreases, eventually to zero when δ becomes too large.

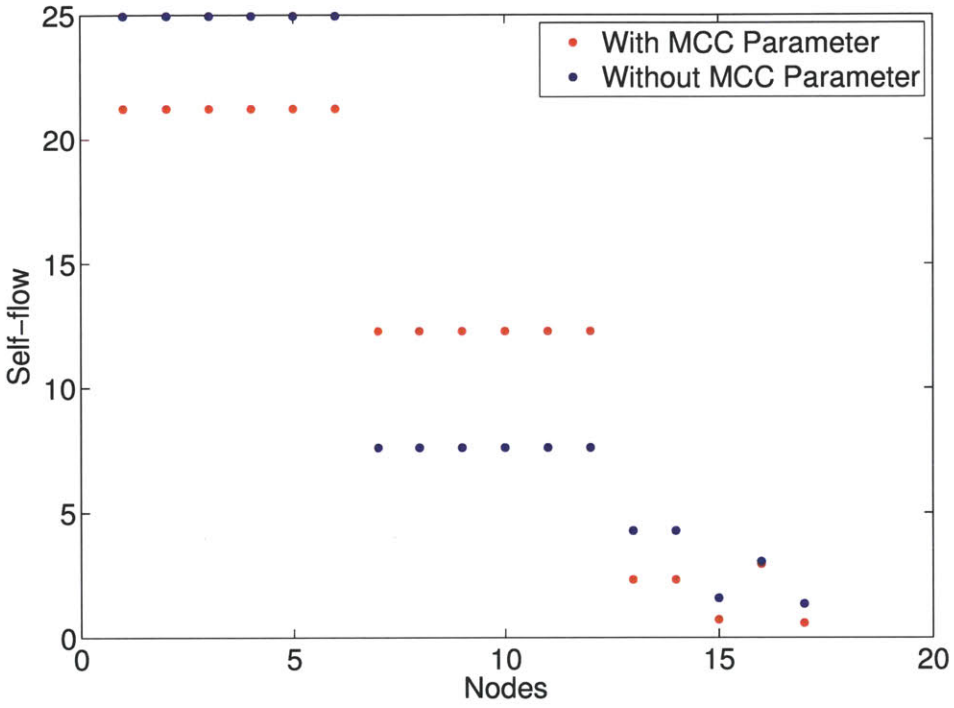


Figure 4-4: Self-Flow over nodes in the network $\mathcal{G}_1$

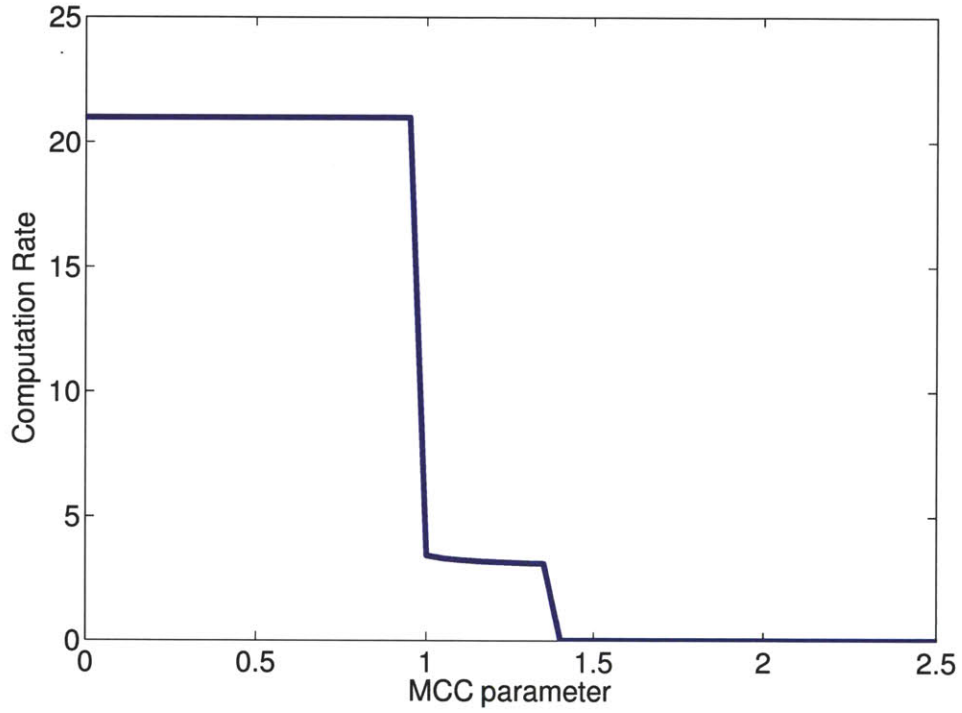


Figure 4-5: Total Computation Rate of f_1 and f_2 vs. MCC constraint.

Chapter 5

A Linear Computation Cost Model

5.1 Description

The Linear Computation Cost Model consists of modifying the original Node Arc LP in Figure 3-4 by adding costs to specific nodes proportional to the amount of self-flow through that node.

Algorithm 4.

$$\max \quad \sum_{i=1}^{n} p_i \lambda_i - \sum_{v \in V} \delta_v \left(\sum_{\theta \in \Gamma} f_{vv}^{\theta} \right)$$

subject to equations 3.1-3.7.

$\delta_v, v \in V$ are the linear computation cost parameters, non-negative weights on the total self-flow through each node $v \in V$ that controls the computation cost. Unlike the MCC model, this is a linear function and therefore can be solved with a linear program.

5.2 Simulations

In this section, we consider the communication network $\mathcal{N}_2(V_2, E_2)$ as depicted in Figure 5-1 and the receiver node only demands to compute f_1 as defined in Figure 4-1. The network in Figure 5-1 has all edge capacities of 10, and we can see the bottleneck is at the minimum cut of the network, across the edges from n_9 to t and n_{10} to t, and the min-cut rate of this network is $\gamma = 20$. We place costs on nodes $n_7 - n_{10}$ to analyze the redistribution of flow and the effect on computation rate.

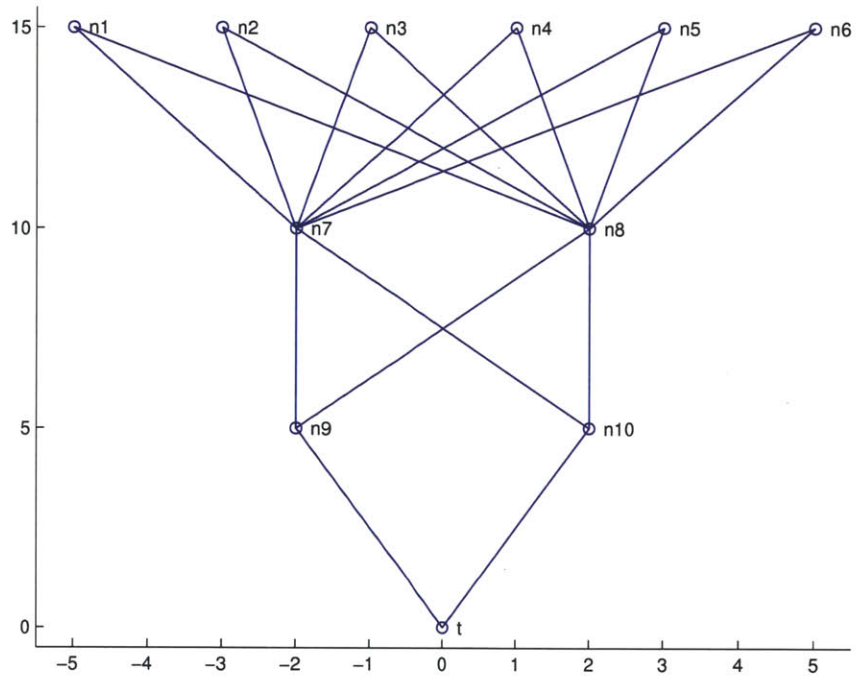


Figure 5-1: Initial communication network $\mathcal{N}_2$.

5.3 Results

Figure 5-2 shows the redistribution of flows occurring when all computations at n_8 are given a Linear Computation Cost (LCC) constraint of 10. With no computation costs, flow is distributed equally among nodes n_7, n_8, n_9, and n_{10}. However, by

making n_8 an expensive node at which to compute, flow is redistributed and getting passed on further to n_9 and n_{10}.

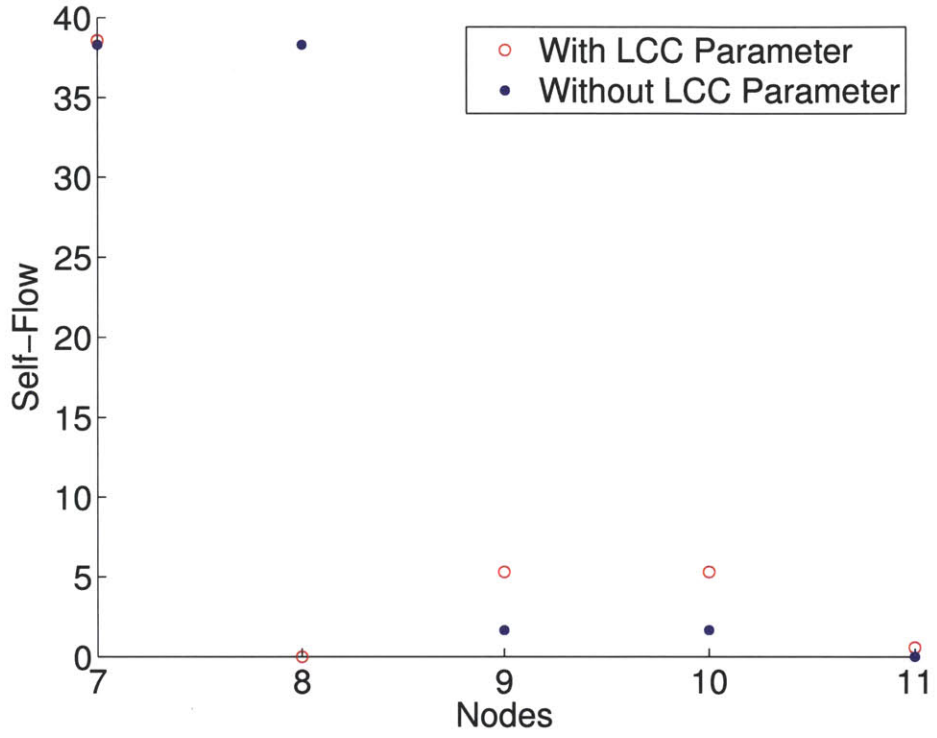


Figure 5-2: Self-Flow over nodes in the network affected by the bottleneck $\mathcal{G}_2$.

Figure 5-3 shows the effect on computation rate of varying LCC constraint δ from 0 to 0.5 for all computations at $n_7 - n_{10}$ (the bottleneck). We have placed costs on all nodes necessary for computation except the terminal node. We can see that, when $\delta = 0$, we achieve the upper bound for R_F as given by Theorem 2. As the LCC constraint increases, all of the computation gets redistributed to the terminal node, a more centralized scheme, which limits the computation rate to the bottleneck capacity of the network over the number of sources.

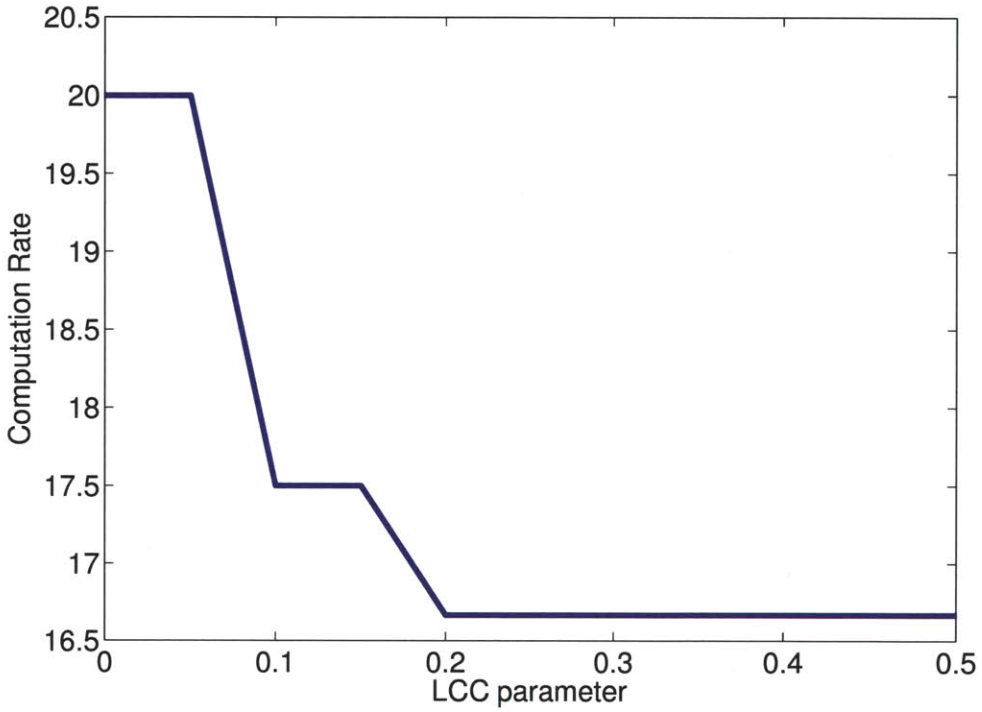


Figure 5-3: Computation Rate vs. LCC constraint on a subset of nodes.

Chapter 6

Network Design

In this chapter, we shall outline and simulate an algorithm that designs a network that maximizes computation rate for a specific set of functions and their probability distribution while minimizing network complexity.

Definition 5. *Given a network $\mathcal{N} = (V, E)$, where V is the set of nodes and E the set of all possible edges in $\mathcal{N}$, a sparse network $\mathcal{N}_s = (V, E_s)$ is optimal for a set of functions $\mathcal{F}$ if $E_s \subseteq E$ and its computation rate R_s is maximized over all possible subsets of E of size $|E_s|$.*

In our algorithm, we are defining network complexity by the number of edges in our network. Therefore, an optimal network design of a given complexity is defined as the network with a given number of edges e with the highest computation rate of the set of all network topologies with a set of e edges that are a subset of all possible edges of our initial network.

6.1 Greedy Algorithm

In this section, we are sparsifying a dense network with a greedy approach. At each iteration, we run Node Arc LP and take away the edge with the smallest ratio of flow to capacity, which is the locally optimal solution to improving the ratio of computation rate and network complexity. The algorithm stops when we have reached the desired

number of edges. This algorithm can be combined with the models described in Chapters 4 and 5, but for the simulation described in 6.2 we stick with the basic model as described in the framework in Section 3.4.

Algorithm 6. *The greedy algorithm is as follows,*

1. *Use Node Arc LP to compute all flows f_{ij}^{θ} for each edge for the network $\mathcal{N}^r = (V, E^r)$, where E^r is the set of possible edges in $\mathcal{N}$.*

2. *Eliminate the edge (u, v) with the smallest ratio of flow to capacity, $(u, v) = \arg\min_{(u,v)\in E^r} \frac{g_{ij}^r}{c(i,j)}$ where $g_{ij}^r = \sum_{\theta\in\Gamma} f_{ij}^{\theta}$ and $c(i, j)$ is the capacity of edge (i, j).*

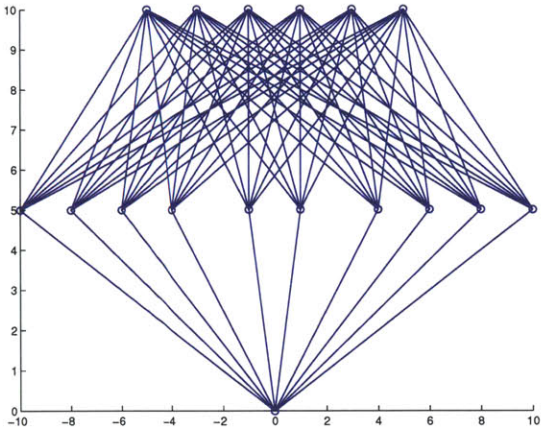
3. *Update the set of edges: $E^{r+1} = E^r \backslash \{(u_1, v_1)\}$.*

4. *If $|E^{r+1}|$ is the desired number of edges, terminate. Else, repeat.*

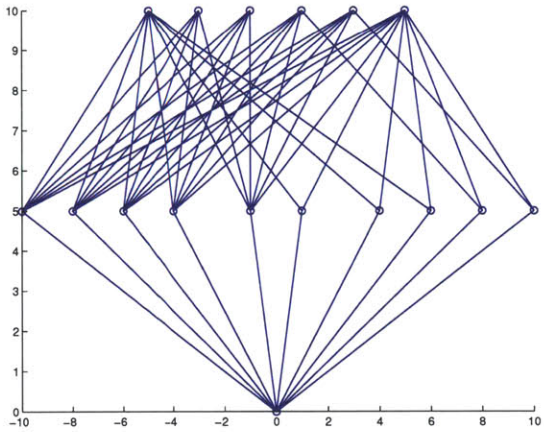
6.2 Simulations

We start out with an initial communication network $\mathcal{N}_3 = (V, E)$ with 6 sources, one layer of 10 nodes, and a terminal node as shown in Figure 6-1(a). The set of possible edges are all possible pairings of nodes that are between layers. Therefore possible edges include and are limited to: all edges between $\{n_1, \ldots, n_6\}$ and $\{n_7, \ldots, n_{16}\}$ (but none within sets) and all edges between $\{n_7, \ldots, n_{16}\}$ and n_t.

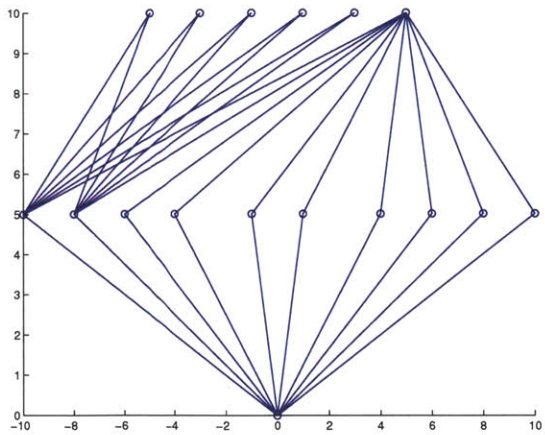
We calculate the ratio of flow to capacity along each edge, and remove the edge with the smallest ratio. No nodes on the same layer or non-adjacent layers are connected, as shown in Figure 6-1(a).



(a) Initial communication network $\mathcal{N}_3$



(b) Iteration 20. Rate 1: 5.0411, Rate 2: 4.9589



(c) Iteration 40. Rate 1: 3.4555, Rate 2: 3.0445

Figure 6-1: Network Sparsification

6.3 Results

Figures 6-1(b) and 6-1(c) show the networks after iterations 20 and 40, where the original network in Figure 6-1(a) has been reduced by 20 and 40 edges respectively. Also in the description are the computation rates for f_1 and f_2.

Figure 6-2 shows the total computation rate over $\mathcal{N}_3$ as an edge is taken away in each iteration. We compare its performance with a random algorithm, where at each iteration a random edge is taken from the network. We run the random algorithm 5 times and average over the runs to account for scenarios where edges that are taken out cause the computation rate to drop to zero. The greedy curve does not drop at every iteration, because, in cases where an edge is removed, it is still possible to redistribute flows, which means the network still has an excess of edges that, when removed, do not diminish the min-cut of the network. If an edge is removed in a portion of the network that is not the min-cut, this does not affect rate because the rate is determined by the min-cut, by the min-cut max-flow theorem [1].

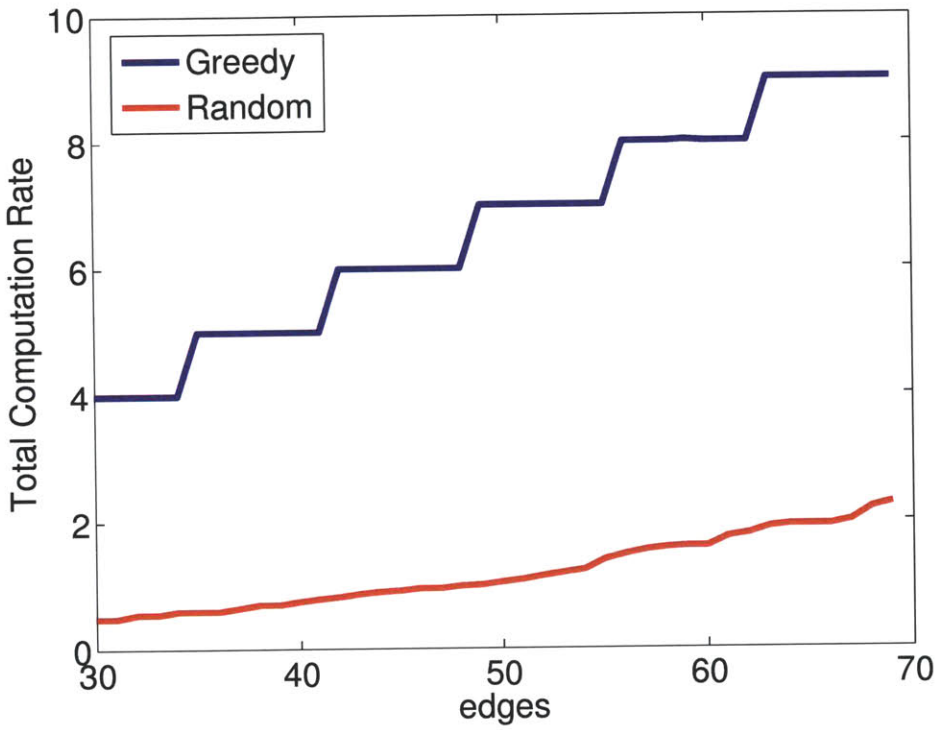


Figure 6-2: Computation rates summed over f_1 and f_2.

Chapter 7

Conclusions

7.1 Project Accomplishments

In this thesis we have outlined a framework to find the maximum computation rate of a set of functions for a given network with communication limits (capacity constraints) in Chapter 3. We then discuss frameworks with different types of computation limitations, the Linear Computation Constraint Model (Chapter 5), and the Maximum Computation Constraint Model (Chapter 4). The LCC model uses a linear function at each node to represent computation cost, while the MCC model uses the maximum computation over all nodes in the network. Since this is not a convex problem, a relaxation of l^p-norm is used as a substitute for the max-norm [3]. While this is useful for achieving the best computation rate out of a given network, a better and more useful optimization problem to solve for cloud computing is how to design a network specialized for a set of functions. This problem is broached with a sub-optimal greedy algorithm, as described in Chapter 6. We found we were able to sparsify a given network while maintaining the same computation rate, and obtain a locally optimal sparsifying solution at each iteration. However, future work still needs to be done to find an optimal algorithm for designing a cloud network, as defined in Definition 5.

7.2 Future Work

We have tried various algorithms to achieve optimal sparse networks using the basic idea of our computation cost models, and adding a sparsity term δ to minimize the number of edges in the network. We added it as a cost to all the flows over the network, excluding self flows. Unfortunately, the network is very sensitive to δ, and flows go to zero too quickly to obtain a sparse network. More work will need to be done to find an optimal algorithm, but another direction for optimizing cloud networks is determining the physical location of source data. Given a set of possible data locations, we can create a cost model that represents their distances from each other to find the model with the highest computation rate.

In this thesis we have only analyzed two cost models, the LCC model and the MCC model. However, there are other types of costs in cloud computing to consider and model when routing flows. Common costs to model are network congestion and slower nodes with higher computation costs. A variation of the MCC model that can be used is a convex cost function, where the cost is proportional to the amount of self-flow through each node. Another aspect of to improve performance we did not address is functional compression. There is a certain amount of data distortion that occurs with compression and decompression of data, and therefore we do not always need an exact rate of computation when analyzing a network. We should investigate alternative methods, like graph coloring, to our convex programming solution and analyze the performance vs. accuracy of using information theory tools to approximate rates over the cloud.

Function flows can also be useful in dealing with privacy issues in the cloud. To protect private information and filter it from the results requested at the receiver of a network (e.g. pulling information from insurance records), we limit the areas of the network that Untrusted areas of the network can have limits on the type of information they can access, which can be represented using the LCC model with $\delta = \infty$ on the computations requiring the sensitive information.

Future work can also be done to analyze different computation trees for functions.

A function has multiple computation trees, and while we have addressed that it is best to choose a computation tree that encompasses all mappings on a network, there also may be runtime advantages to having computation trees with similar topologies when dealing with multiple functions, even if it means excluding some mappings.

Appendix A

Code

A.1 Node Arc LP code

```
function [ MaxFlow, SparseNet, flows, X, SelfFlows, edgeflows,
    FlowConsConstraintA, FlowConsConstraintB] = NodeArcLPa( CommNet,
    CompWeights, FuncWeights, sources, t, CompTrees, Capacity)
% Calculates flow for a set of functions with associated weights
% CommNet is a adjacency matrix representing the Communication Network.
% NetEdges is the list of directed edges in the network
% CompWeights is a set of arrays that are the penalties on each
    computation edge for computation cost.
% FuncWeights = weight associated with each function
% Sources is an set of arrays of integer values specifying the nodes
    that are sources for each function.
% t is an array of indices of the terminal nodes for each function.
% CompTrees is a set of matrices of the Computation Tree, where there is
     a +1 value if the row node is the predecessor of the col node.
% Capacity: capacity of each edge on the communication network
% MAXIMIZE: rate-sum(weight*rate of edge)-sparsity constant*sum(all edge
     flows)
nodes = [];
selfflows = [];
NumFunctions = length(FuncWeights);
[rows, cols]=size(CommNet);

flows = CommNet;
for row = 1:rows,
    for col = 1:cols,
        if row==col,
            flows(row,col)=1;
        end
    end
end

x=find(flows); %All nodes and edges in the network that have flow
```

```
%get the number of computation edges in each CompTree by counting
%the number of parent nodes
CompEdges=zeros(1,NumFunctions);
totalWeights=[];
sparseDesign=[];
X=zeros(1,NumFunctions);
for index = 1:NumFunctions,
    CompEdges(index) = length(find(CompTrees{index}));
    X(index)=CompEdges(index)*length(x);
    totalWeights = [totalWeights; zeros(X(index),1)];
    selfflows = [selfflows; zeros(X(index),1)];
    sparseDesign = [sparseDesign; ones(X(index),1)];
    for theta=1:CompEdges(index),
        for j=1:rows,
            sparseDesign(Matrix2Vector(j,j,flows,theta,index,X))=0;
            selfflows(Matrix2Vector(j,j,flows,theta,index,X))=j;
        end
    end
    weights = CompWeights{index};
    [j,~] = size(weights);
    for i=1:j,
        totalWeights(Matrix2Vector(weights(i,2),weights(i,2), ...
            flows,weights(i,3),index,X))=weights(i,1);
    end
end

totalX=sum(X);
%minimizing over computation costs, and computation rates given function
%weights
f=[totalWeights; -ones(NumFunctions,1).*FuncWeights'];
%FlowConservation
FlowConsConstraintA = [];
for func=1:NumFunctions,
    for node = 1:rows,
        for theta = 1:(CompEdges(func)-1),
            newRow = zeros(1, totalX+NumFunctions);
            for eta = GetSuccessors(theta, CompTrees{func}),
                newRow(Matrix2Vector(node, node, flows, eta, func,X)) =
                    1;
            end
            for neighbor = GetNeighbors(node, CommNet,[]),
                newRow(Matrix2Vector(node, neighbor, flows, theta, func,
                    X)) = 1;
                newRow(Matrix2Vector(neighbor, node, flows, theta, func,
                    X)) = -1;
            end
            newRow(Matrix2Vector(node, node, flows, theta, func, X)) =
                -1;
            FlowConsConstraintA = [FlowConsConstraintA; newRow];
        end
    end
end
FlowConsConstraintB = zeros(length(FlowConsConstraintA(:,1)), 1);
```

```
%TConservation
TConservationA = [];
for func = 1:NumFunctions,
    for node = 1:rows,
        newRow = zeros(1,totalX+NumFunctions);
        for neighbor = GetNeighbors(node, CommNet, []),
            newRow(Matrix2Vector(node, neighbor, flows, CompEdges(func),
                func, X)) = 1;
            newRow(Matrix2Vector(neighbor, node, flows, CompEdges(func),
                func, X)) = -1;
        end
        newRow(Matrix2Vector(node, node, flows, CompEdges(func), func, X
            )) = -1;
        if node == t(func),
            newRow(totalX+func) = 1;
        end
        TConservationA = [TConservationA; newRow];
    end
end
TConservationB = zeros(rows*NumFunctions, 1);

%Generation of theta_l
totalSources = [];
for i=1:NumFunctions,
    totalSources = [totalSources,sources{i}];
end
numSources = length(totalSources);
ThetaGenerationA = [];
ThetaGenerationB = zeros(rows*numSources, 1);
row = 0;
for node = 1:rows,
    for func = 1:NumFunctions,
        for theta = 1:length(sources{func}),
            newRow = zeros(1,totalX+NumFunctions);
            if node==sources{func}(theta),
                newRow(totalX+func) = -1;
            end
            newRow(Matrix2Vector(node, node, flows, theta, func, X)) =
                1;
            ThetaGenerationA = [ThetaGenerationA; newRow];
        end
    end
end

EQConstraintA = [FlowConsConstraintA; TConservationA; ThetaGenerationA];
EQConstraintB = [FlowConsConstraintB; TConservationB; ThetaGenerationB];

%Capacity Constraints (LEQ)
CapConstraintA = zeros(length(Capacity), totalX+NumFunctions);
rowcount = 0;
for row = 1:rows,
    for col = 1:row,
        if CommNet(row,col)~= 0 && row>col,
            rowcount = rowcount + 1;
```

```
127             for func = 1:NumFunctions,
                    for theta = (1:CompEdges(func)),
129                     CapConstraintA(rowcount, Matrix2Vector(row, col,
                            flows, theta, func, X))=1;
                        CapConstraintA(rowcount, Matrix2Vector(col, row,
                            flows, theta, func, X))=1;
131                 end
                end
133         end
    end
135 end
    CapConstraintB = Capacity;
137
    %Non-negativity Constraints (GEQ)
139 NonNegConstraintA = -1*eye(totalX+NumFunctions,totalX+NumFunctions);
    NonNegConstraintB = zeros(totalX+NumFunctions, 1);
141
    size(CapConstraintA);
143 size(CapConstraintB);
    size(NonNegConstraintA);
145 size(NonNegConstraintB);
    LEQConstraintA = [CapConstraintA; NonNegConstraintA];
147 LEQConstraintB = [CapConstraintB; NonNegConstraintB];

149 %Run linear program
    MaxFlow = linprog(double(f), double(LEQConstraintA), double(
        LEQConstraintB), double(EQConstraintA), double(EQConstraintB));
151
    %Generate sparse network
153 SparseNet = zeros(size(CommNet));

155 for row = 1:rows,
        for col = 1:rows,
157         if CommNet(row,col)~= 0,
                for func = 1:NumFunctions,
159                 for theta = 1:CompEdges(func),
                        SparseNet(row,col)=SparseNet(row,col)+ MaxFlow(
                            Matrix2Vector(row, col, flows, theta, func, X));
161                 end
                end
163         end
        end
165 end

167 SparseNet = SparseNet+SparseNet';
    edgeflows = SparseNet;
169 SparseNet = SparseNet>1e-5;

171 SelfFlows = [];
    for i=1:rows,
173     SelfFlows = [SelfFlows, sum(MaxFlow(find(selfflows==i)))];
    end
175 end
```

code/NodeArcLPa.m

A.2 Embedding Edge LP code

```
function [ embeddings,weights ] = EmbeddingEdgeLP( CommNet, Capacity,
    FuncWeights, sources, t, CompTrees, MaxFlow )
%EmbeddingEdgeLP Given the Network graph, capacities, set of arrays
    source nodes
%s, set of terminal nodes t, computation trees, and the linprog solution
    , we generate
%the set of embeddings where sum(weight_i*x(B_i))=lambda
%in each iteration of the while loop, we find an embedding with a
    nonzero
%flow and remove the corresponding edge flows to obtain another feasible
%solution with a reduced rate. We start by finding a mapping of
%theta_terminal.
[rows, cols]=size(CommNet);
weights={};
embeddings={};
flows = CommNet;
for row = 1:rows,
    for col = 1:cols,
        if row==col,
            flows(row,col)=1;
        end
    end
end

x=find(flows);
NumFunctions = length(t);
lambda = MaxFlow(length(MaxFlow)-NumFunctions+1:length(MaxFlow));
totalLambda=0;
X=zeros(1,NumFunctions);
for index = 1:NumFunctions,
    CompEdges(index) = length(find(CompTrees{index}));
    X(index)=CompEdges(index)*length(x);
end

for func=1:NumFunctions,
    newLambda = 0;
    weights{func}=[];
    embeddings{func}={};
    while abs(lambda(func) - newLambda)>1 && newLambda<lambda(func),
        stopflag=0;
        thetaT = length(find(CompTrees{func}));
        B{thetaT}=t(func);
        Embeddingflow=[];
        trackpath={};
        for i = thetaT:-1:1,
            trackpath{i}=[];
            v = B{i}(length(B{i}));
            vtried = [];
            %goes into a loop, keeps trying the same nodes over and over
                again.
            while(stopflag==0),
```

```
zeroflows=0;
possible = [v, GetNeighbors(v, CommNet, vtried)];
maxf = 0;
u = 0;
for each=1:length(possible)
    if maxf < MaxFlow(Matrix2Vector(possible(each),v,
        flows,i,func,X)),
        u = possible(each);
        maxf = MaxFlow(Matrix2Vector(possible(each),v,
            flows,i,func,X));
    end
end
if u == 0,
    stopflag = 1;
end
if MaxFlow(Matrix2Vector(u,v,flows,i, func, X))<=0,
    index = Matrix2Vector(u,v,flows,i, func, X);
    MaxFlow(index)= MaxFlow(index)-trackmin;
    stopflag=1;
end
vtried = [vtried, u];
if u~=v && ~isempty(find(B{i}==u,1)),
    P=B{i}(1:find(B{i}==u));
    if length(P)~=length(B{i}),
        B{i}=B{i}(length(P)+1:length(B{i}));
    else
        B{i}=[];
    end
    temp = [];
    for nodei=1:length(P)-1,
        if P(nodei) ~= P(nodei+1) && CommNet(P(nodei),P(
            nodei+1))~=0,
          temp = [temp, MaxFlow(Matrix2Vector(P(nodei),P
              (nodei+1),flows,i,func,X))];
        end
    end
    temp = [temp, MaxFlow(Matrix2Vector(u,v,flows,i,func
        ,X))];
    y = min(temp);
    %update the flows
    for nodei=1:length(P)-1
        current = Matrix2Vector(P(nodei),P(nodei+1),
            flows,i,func,X);
        MaxFlow(current)=MaxFlow(current)-FuncWeights(
            func)*y;
    end
    MaxFlow(Matrix2Vector(u,v,flows,i,func,X))=MaxFlow(
        Matrix2Vector(u,v,flows,i,func,X))-FuncWeights(
        func)*y;
else
    Embeddingflow = [MaxFlow(Matrix2Vector(u,v,flows,i,
        func,X)), Embeddingflow];
    trackpath{i} = [u,v; trackpath{i}];
    trackmin = min(Embeddingflow);
```

```
                end
                if v~=u,
                    B{i}=[u,B{i}];
                    v=u;
                else
                    for eta = GetPredecessors(i, CompTrees{func}),
                        B{eta} = u;
                    end
                    break;
                end
            end
        end

        if stopflag==0,
            xB= min(Embeddingflow);
            FuncWeights(func)*xB
            newLambda = newLambda + FuncWeights(func)*xB
            %Remove x(B) amount of flow from all edges in B
            Path=[];
            for theta=1:thetaT,
                for i=1:length(trackpath{theta}(:,1)),
                    index=Matrix2Vector(trackpath{theta}(i,1),trackpath{
                        theta}(i,2),flows, theta, func, X);
                    MaxFlow(index)=MaxFlow(index)-FuncWeights(func)*xB;
                end
            end
        else
            xB=0;
        end
        check=0;
        for e=1:length(embeddings{func}),
            check=0;
            for b = 1:thetaT,
                if length(embeddings{func}{e}{b})==length(B{b}),
                    if embeddings{func}{e}{b}==B{b},
                        check= check+1;
                    end
                end
            end
            if check==thetaT,
                break;
            end
        end
        if check==thetaT,
            weights{func}(e) = weights{func}(e)+FuncWeights(func)*xB;
        else
            embeddings{func}{length(embeddings{func})+1} = B;
            weights{func} = [weights{func},FuncWeights(func)*xB];
        end
    end
end
```

code/EmbeddingEdgeLP1.m

Bibliography

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. 46:1204–1216, 2000.

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Network flows: Theory, algorithms, and applications, 1993.

[3] S. Deb and R. Srikant. Congestion control for fair resource allocation in networks with multicast flows. *Networking, IEEE/ACM Transactions on*, 12(2):274–285, 2004.

[4] S. Feizi, A. Zhang, and M. Médard. A network flow approach in cloud computing. *see arxiv.org*, 2012.

[5] Soheil Feizi and Muriel Médard. When do only sources need to compute? on functional compression in tree networks. In *2009 Annual Allerton Conference on Communication, Control, and Computing*, September 2009.

[6] János Körner. Coding of an information source having ambiguous alphabet and the entropy of graphs. In *6th Prague Conference on Information Theory*, pages 411–425, 1973.

[7] D. Lun, N. Ratnakar, M. Médard, R. Koetter, D. Karger, T. Ho, E. Ahmed, and F. Zhao. Minimum-cost multicast over coded packet networks. *IEEE Transactions on Information Theory*, 52(6):2608–2623, 2006.

[8] V. Shah, B.K. Dey, and D. Manjunath. Network flows for functions. In *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, pages 234–238. IEEE, 2011.

[9] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem. *Journal of the ACM (JACM)*, 37(2):318–334, 1990.