Data Storage Networks

Authors: Preet Bola, Mike Earnest, Kevin Varela-O'Hara and Han Zou

Advisor: Walter Rusin

1 Introduction

Ever wondered how GPS worked? When we give our own location and the destination that we are trying to get to (meaning from and to information), GPS finds us the best route possible. Some systems provide few more options than couple basic ones for the user to select. For instance, some systems let the user select if they want to get to their destination fast as possible, shortest distance, by car or on foot etc. Half of our research problem is just that.

Imagine a small cluster of computer users; let's call this cluster a node for our own convenience. These users are connected with each other in a way so that they can communicate with each other. Now, add few other nodes of such kind to our original node and connect these nodes in some nontrivial way. This new cluster of nodes represents, what we call in our project, a "Network". When a user from a node requests a file that is on some other node, our job is to figure out the most optimal way to stream this file to the user over the given network. We are optimizing the file streaming, in simplest words. This is just beginning half of the problem, as I mentioned earlier.

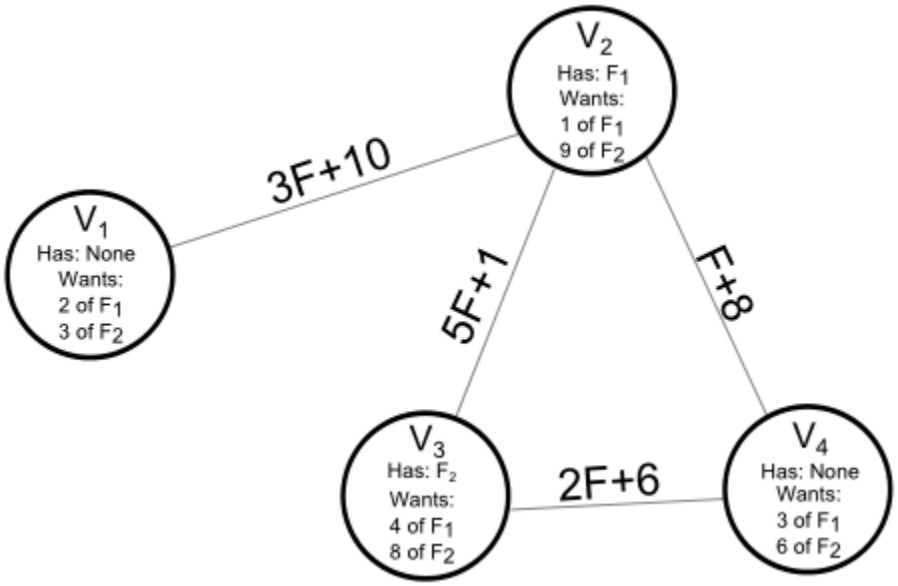
The second half of our problem is to figure out the best location/node to place a particular file on. For a certain demand, it might be better to move some files to other nodes to make the file streaming more efficient. One thing to keep in mind while doing this is that there are capacity constraints on every system, else we can just copy all the files on each and every node and hence the cost of download will be zero. Our ultimate goal is to keep the cost of download as low as possible. For Instance, the time it takes to download a file will be

important to keep down and if the traffic on certain node is heavy, we need to overcome the congestion problem. Basically, our algorithms will be looking for another route to send some of the files over to get rid of congestion while considering next most efficient path.

In addition to all of the above concerns we discussed, there one another thing to take care of-node failure or connection failure. Node failure symbolizes the event where no files can be accessed at a particular node and it can't be used to pass any traffic through. In other words, a node is completely disconnected from the network. Where, connection failure is about unusable connections, but we can still access the data from nodes by either using other connections or by building new connections. This project gets a lot tricky as we add more requirements in.

2 Our Model

We can look at a computer network as a weighted graph where the nodes are computers, the edges are connections between computers, and the edge weights are the time it takes to send a file down an edge. We decided that simple constant edge weight would not represent the complexity of the system, so the weights were set to be liner functions of the number of files passing through the edge in a cycle to more accurately express bandwidth. We can represent our network as two $N \times N$ adjacency matrices one for the slope of the edges and the other for the constants. For example, here is a network:



And here are the matrices representing it:

$$\text{Slopes} = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{pmatrix} 0 & 3 & 0 & 0 \\ 3 & 0 & 5 & 1 \\ 0 & 5 & 0 & 2 \\ 0 & 1 & 2 & 0 \end{pmatrix} \end{array} \quad \text{Constants} = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{pmatrix} 0 & 10 & 0 & 0 \\ 10 & 0 & 1 & 8 \\ 0 & 1 & 0 & 6 \\ 0 & 8 & 6 & 0 \end{pmatrix} \end{array}$$

In addition to the network structure, we must store the location of all of the files within the network in an $N \times F$ matrix. This matrix will change when we change the files locations to optimize access time. For our example network this matrix is:

$$\begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cc} F_1 & F_2 \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array}$$

The capacity is the maximum number of files that a node can hold. The final thing that we must store about our network is the nxF demand matrix, which nodes want how many, of which files. For our example network the demand matrix is:

$$\begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cc} F_1 & F_2 \end{array} \\ \begin{pmatrix} 2 & 3 \\ 1 & 9 \\ 4 & 8 \\ 3 & 6 \end{pmatrix} \end{array}$$

3 Optimizing Download Time

The overreaching goal of our project is to, given a network layout and the relative demand of each file, to find the placement of files which optimizes the total time it takes each user to download these files. However, in order for this to be accomplished, we first had to determine, given an allocation of files, exactly how these files can be optimally downloaded, so we have a metric with which to compare the optimality of two given file allocations. By this, we mean that each file can be sent along multiple paths, and that some of these paths have different download times. Our goal, then, is to find the paths for each required transfer which minimizes the total download time. We came up with three algorithms to do this, one of which we refer to as Greed.

3.1 The Greedy Algorithm

Initially, we decided that we would use a greedy algorithm to solve this problem, where each file is sent once at a time, and chooses the path that minimizes its own download time given the files that have already been sent. To be more precise, initially the network, a weighted graph whose edge weights are a function of the number of files, $A * F + B$, has each of edges set to a weight of $A * 1 + B$. These are the weights from the "point of view" of the first

file to be sent; that file wants to minimize its download time, and this is given by adding up the values of the bandwidth functions of the edges of the path he chose, each evaluated at one because there is only one file passing through them. Then, using the Floyd-Warshall algorithm to find the shortest paths between all pairs of nodes, the file whom can be sent with the lowest time is sent first. This means that if file F is sent to node N, the N,F entry of the demand matrix is decreased by 1,and the weights of the edges used are updated to be $A * 2 + B$. This process is continued until all files have been sent. The amount of times a file passes through each edge is stored in a matrix, as well the equivalent data for each node (the latter will be used later). Afterwords, the total download time (the sum of the download times for each file) of the process is found by

$$\sum (A_i * F_i + B_i) * F_i$$

where the ith edge has Fi files passing through it and has a bandwidth function $A_i * F + B_i$. The reader should be able to verify that this is the total download time.

This complexity of this algorithm is $O(T \times N^3)$, where N is the number of nodes and T is the number of files to be sent. This is because each application of the Floyd-Warshall algorithm is $O(N^3)$[3], and it is used T times.

3.2 The Considerate Algorithm

The second algorithm, we devised to solve this problem used a similar principle. Again, the files are sent one at a time, but instead of attempting to minimize their download time, they minimize the increase they would cause to the total download time. This is subtly different than the previous method, because the greedy algorithm did not account for the fact that when a file takes a path, it retroactively increases the download times of the previously sent files. Because this takes this increase into account, we dubbed it Considerate. The increase to the total download time by a particular file is given by:

$$\sum (A_i \times (F_i + 1) + B_i) \times (F_i + 1) - \sum (A_i \times F_i + B_i) \times F_i$$

which simplifies to:

$$\sum A_i \times (2F_i + 1) + B_i$$

where i ranges over the edges the file is sent along (this is simply new total cost minus old total cost). Thus, in order to minimize this increase, the weight of the ith edge used in Floyd-Warshall algorithm is simply $A_i(2F_i + 1) + B_i$ instead of $A_iF_i + B_i$.

It would appear that at each step, this algorithm makes the best choice, because each file sent increases the download time as little as possible. However, we found that Greed actually performs better than Considerate, and we have been unable to find a satisfactory explanation for this.

3.3 Quadratic Programming

Finally, we created a method, QuadOpt, which always produces the optimal total download time, but is much slower than the other algorithms and therefore less practical. An optimal transfer can be described by the a number of variables of the form $T_{e,f}^{+/-}$, which is the number of times file f is sent along edge e in the + (-) direction (the algorithm arbitrarily assigns which direction is plus and minus). Given these variables, QuadOpt then creates a series of linear equations which govern these variables. All of these are based on two rules:

1. The net inflow of a particular file into a node is equal the the demand at that node for that file.

2. The sum of the net outflows of a particular file from all nodes where that file is located is equal to the total demand for that file.

This will generate several linear equations which constrain the solution. Then, QuadOpt uses MATLAB's quadratic programming tool to minimize the function

$$\sum (A_i \times F_i + B_i) \times F_i$$

where each F_i is given by

$$\sum T^{+}_{e,F_i} + \sum T^{-}_{e,F_i}$$

4 File Allocation Methods

After investigating the above algorithms of optimally streaming files in a fixed network with fixed file layout and demand, our group took a second approach to the problem: finding the optimal file arrangement for a specific network. Optimal file arrangement refers to the one that offers the lowest average streaming cost for the whole demand of the network at a certain time.

4.1 Genetic Algorithm

Genetic Algorithm is a popular global optimization method designed for discrete problems. At the time of its first invention, GA was inspired by biological evolution. It is based on the idea that promising potential solutions to the problem could be mixed up in some way to produce even better solutions. [1] GA is a population based algorithm, which uses a fixed number of sample solutions as the current generations. Several operators are implemented on the current generation to produce the next generation. The following sections will provide a brief review of GA operators together with how they are applied in the File Allocation Problem.

4.1.1 Organism

Each individual in a generation of Genetic Algorithm is called an organism. In the FAP, each organism stands for a valid file arrangement that could potentially be the solution to

the problem. Valid means that all the nodes do not exceed their storing capacity. Organisms are represented by $n \times F$ file matrices in code.

4.1.2 Fitness Level and Fitness Function

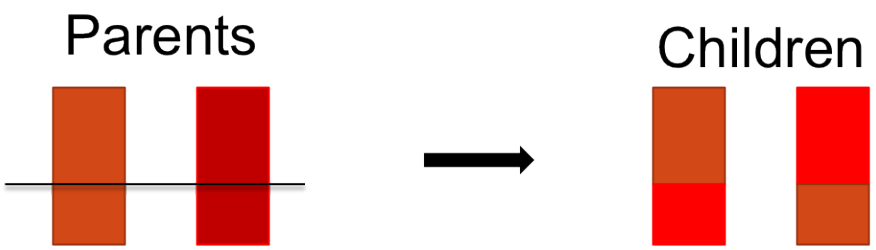
Each organism in GA is assigned a Fitness Level by Fitness Function. This level represents how close the solution is to the optimal solution. By convention, the higher the Fitness Level, the better the solution is. Usually, the Fitness level is assigned by a single Fitness Function that is defined for all the organisms in the solution space. In our implementation of GA, we use Greedy Algorithm as the fitness function and the Fitness Level is calculated as the inverse of the average streaming time (returned by Greedy Algorithm).

4.1.3 Selection

This operator selects organisms in the population for reproduction. [2] The fitter the organism, the higher the chance that it is selected as parents for the next generation. In our implementation, we use Fitness Proportionate Selection. It means that the probability that a single organism is selected is equal to its fitness divided by the total fitness of all the organisms in the current generation. This selection method guarantees a faster convergence of GA since promising organisms retain through out generations.

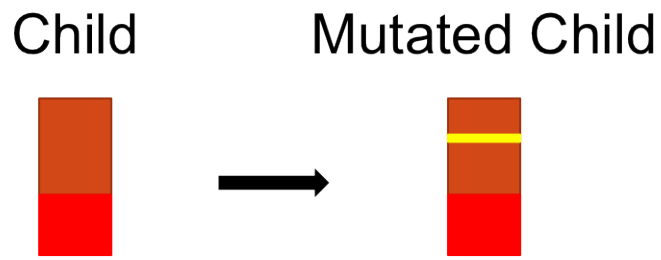
4.1.4 Breeding

Breeding is the main evolutionary operator of Genetic Algorithm. After a pair of parents is selected, they are now combined and modified in some fashion to produce their children. We use single-point crossover as our breeding scheme, which means that a single cutting point is generated through parent organisms and both organisms are cut into 2 parts. Then the head of one parent is combined with the tail of the other parent and vice versa as shown in the picture below. Two children are generated in this process.



4.1.5 Mutation

Mutation guarantees that GA is a global optimization method. It works in the following way. After breeding produces a child, there is a chance that part or parts of the newborn will get randomly mutated (just like genovariation). The event that a single entry of the organism is mutated is independent to the event that any other entry gets mutated. Mutation happens by chance and if it is to happen, it will look like the following.



4.1.6 Stopping Criteria

At each generation, Selection, Breeding, and Mutation are operated in order to produce new organisms for the next generation until the new generation reaches the same size as the current generation, which is called the population size. Genetic Algorithm terminates when a certain number of generations are iterated or when the top fitness of generation stops increasing.

4.2 Node Scoring Method

As the reader may recall, the Greedy algorithm in addition to finding the approximate optimal download time for a given file arrangement, it also counts how many times each

file passes through each node. The reason we keep track of this information is because we hypothesized that it would be a good indicator of which nodes would be good locations for which files. Say that a file is being sent along the path $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow \ldots \rightarrow N_k$.

Then the download time could be decreased by moving this file to an intermediate node in this path (if it were moved to N_3, the optimal path would be shortened to $N_3 \rightarrow N_4 \rightarrow \ldots \rightarrow N_k$). Thus, moving a file tothe node which is used as an intermediate node the most will shorten as many of these paths as possible.

Thus, we designed an algorithm which first assigned each node a series of scores representing how good a location they would be for each file, then replacing low scoring files with high scoring ones. This algorithm, named FileTransfer3, takes an initial file matrix as an input, which we recommend the user randomly generate using RandNet. It then uses Greed to find out how many times each file passes through each node when the files are downloaded. This data is stored in a N by F, where N is the number of nodes in the network and F is the number of files (the i,j entry is the number of times file j passes through node i). This is the matrix of node scores. Then, the algorithm cycles through each node, and at each node, it replaces the lowest scoring file with the highest scoring one. It then recalculates the new optimal download time; if the move increases the cost, it undoes the switch. It does this a number of times equal to the node's capacity, and repeats this loop over the nodes until no moves are made.

We found that, on average, this process created a file arrangement whose performance was 65% better than the original input.

5 Node Failure

At the beginning of the paper we discussed our concern over node and connection failure. Connection failure is some what easy to take care of because there is really good chance that some node will have more than one or two connections. In other words, we can find other

ways to send the file over the network. However, a failing node will cause all the information stored on that node to be inaccessible and it will also lead to the failure of all the connections on that node. What tend to make things worse is that, the removal of a cut vertex/node from a network would disconnect the remaining network. Why we are emphasizing this situation more than losing the connections (of course, we are taking node failure to be the same as if all connection to a node fail) is because this could cause the network to lose all copies of certain files. Here we will offer a probabilistic approach to help drop such risk and a heuristic solution to this problem by fixing disconnected parts of network by building new connections.

5.1 Probabilistic Approach

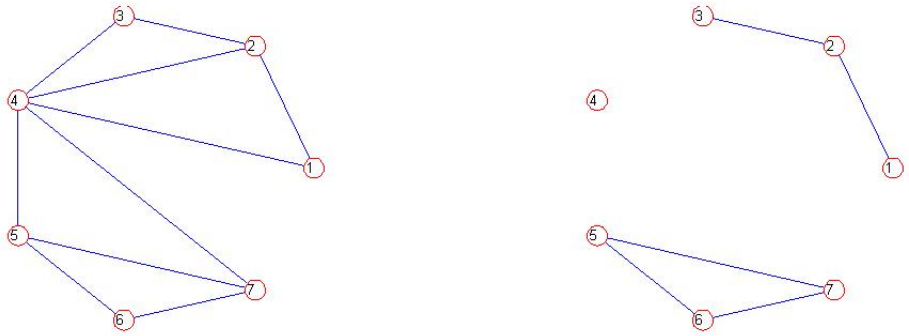
We use probability concept of Inclusion-Exclusion Principle in this approach. To avoid such melt down, we assigned small and equal probability to each node. Then we calculate the probability of losing files with a goal to keep this number down. We used inclusion-exclusion principle to finish this task only up to three steps. For instance, if there are seven files, then we calculate the probability of losing one file for each file, then subtract the probability of losing any two files and then at the end add in the probability of losing any three files. This way we get an upper bound on the probability of losing all files. We can incorporate this probability in our cost calculating and optimizing algorithm to keep this probability under control.

5.2 Heuristic Solution

5.2.1 Step I

Detect and identify components of the remaining graph. A broken network with N components will need at least $N - 1$ new connections to connect back together. As shown in the example below, Node No. 4 fails (together with its connections) and the algorithm detects

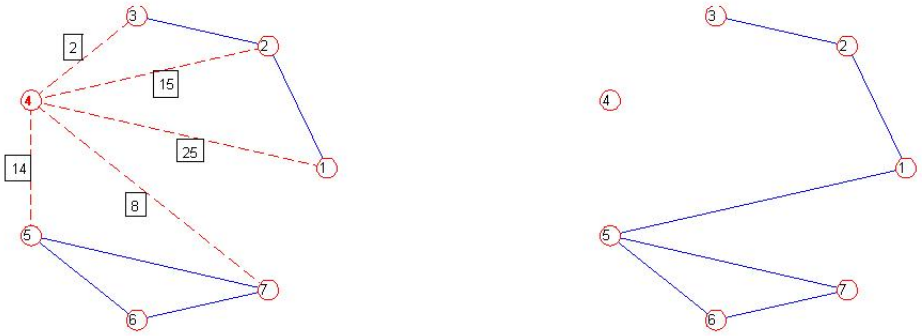
two components. The first component has Node No. 1-3 and the second one has Node No. 5-7. This



network need one new connection.

5.2.2 Step II

Among the nodes previously connected to the failing node, select those with the highest traffic (on the connection) to build new connections to. By selecting the nodes with the heaviest traffic, we assume that the new connection will cater more demand as compared to other potential new connections. In this specific example, Node No.1 and No.4 are chose to build the new connection to.



6 Conclusion

Though our methods for determining the file layout that will allow fast download time are significantly better than randomly generated file layouts, we believe there is still room for improvement. Both of our algorithms involve a degree of guesswork; the genetic algorithm randomly generates an initial population, and the node scoring algorithm randomly chooses the initial file distribution and the order in which nodes are examined. In addition, as of now the model we have is not exactly realistic, and would be more applicable if we used a different model for the ways that files were downloaded and for the time it takes data to pass through a network cable. Still, with more development, I think that this research could be used to help storage methods that greatly improve the efficiency with which data is accessed from storage networks.

References

[1] Seliger, Philip. *Engineering Design Optimization*. Thesis, University of Southern California, 2008. Print.

[2] Mitchell, Melanie. *An Introduction to Genetic Algorithms*. Cambridge, MA:MIT 1996. Print.

[3] Floyd, Robert. *Communications of the ACM 5*. 1962. Print.