Maximizing Virtual Machine Pair Placement in Data Center Networks

_____

A project

Presented

to the Faculty of

California State University Dominguez Hills

_____

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

In

Computer Science

_____

By

Jeffrey Lutz

Summer 2019

ACKNOWLEDGEMENTS

The author of this paper would like to thank Dr. Bin Tang for his invaluable guidance without which the creation of this paper would not be possible. The author would also like to thank Eliza Roño for her helpful feedback on earlier drafts of this paper.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The overall purpose of this study was to attempt to maximize concurrent cloud user requests inside of a data center. The way this was done was to produce different algorithms to accommodate as many Virtual Machine pairs as possible into a k-ary fat tree with bandwidth constraints and to evaluate their effectiveness using multiple simulations. These Virtual Machine pairs are assigned a random bandwidth demand within a given range and are placed within the network. The basic setup for our study was to split these algorithms into two different parts, the first part being the algorithm that determines the order of VM pair placement, and the second part being the path finding algorithm. These two algorithms combine to create a method that accommodates varying amounts of VM pairs into our tree given different constraints. Three different pathfinding algorithms were used to decide which VM pair path was taken, one Dijkstra algorithm, one greedy algorithm, and one that simply chooses the first available edge that can accommodate the necessary bandwidth. One of these path finding algorithms is an approximation algorithm which has a provable performance guarantee, thus its performance is near optimal. There are two different VM pair order placement algorithms. The first sorts the VM Pairs by shortest path first, which is determined by which physical machines they are placed in, and the other is by lowest total weight first. There were many different simulations done varying all of the relevant parameters and the Shortest Path First Most Available Bandwidth(SP_MBF) and Approximation Algorithm performed similarly for some cases with the Approximation Algorithm performing consistently better for all cases.

CHAPTER 1

INTRODUCTION

Every two years the amount of total data that has been created is expected to double and is projected to reach 45,000 exabytes by 2020 [10]. This presents quite a few challenges for large companies that have to move large amounts of data around quickly, efficiently, and perhaps most importantly securely. Some of the data centers that store and process this incredibly large amount of data can support physical machines that number in the tens of thousands. Many of the processing tasks that these data centers undertake can be quite large and access tremendous amounts of data. All of this data clearly can't be stored on the same machine, so data has to be accessed across many machines and the processing required has to be distributed among many machines as well. For certain requests or tasks this can create quite large bandwidth demands across many links inside of their own network.

These processing tasks can be very complex for data centers because they are serving clients that have a wide range of requirements than can be quite different from one to the next. This makes it difficult to optimize all of the internal routing protocols because they have to be handled very dynamically. Due to the fact that these requests may require many different physical machines in different locations inside of this large data center the networking path becomes very important. It is important because data centers want to accommodate as many simultaneous user requests as possible while also providing its clients with an exceptionally expedient user experience.

In addition to all of the routing difficulties for accomplishing complex user tasks there is also the problem of energy consumption. Energy itself costs money of course, but as the energy consumption of the data center increases so does its demand for cooling. Properly

cooling a data center can be extremely expensive. Typically, this is done by air-conditioning where all the servers and network equipment lie and recently there have been some new suggestions such as water cooling or even submerging the equipment in non-conducting fluids. Regardless it is important to minimize the length of the networking path that the data flows through such that the equipment necessary to successfully route the data is kept to a minimum.

Data centers have many different factors to consider. They want to minimize running costs by minimizing the power consumption of their hardware while serving as many concurrent users as they can. To accomplish this, they must efficiently use the resources that are available to them i.e. bandwidth, storage, memory, and power. Thus, they will want to accommodate as many virtual machine pairs or traffic as possible while minimizing the consumption of bandwidth given certain bandwidth constraints. This is the problem that we will attempt to aid in solving.

CHAPTER 2

RELATED WORK AND SYSTEM MODEL

RELATED WORK

Power consumption is becoming more and more relevant as data centers grow due to the fact that cooling represents a significant amount of the data centers running costs. Also, the size of these user requests can be so large that the necessary number of physical machines required to complete it are in the tens of thousands [2]. In addition to incredibly large number of physical machines involved the sheer size of the data sets can be almost unimaginable and in the order of magnitude of petabytes [3]. Many of these tasks aren't necessarily bottlenecked by the resources on the physical machines actually doing the computations, but rather on the actual networking distribution [4]. Therefore, where data needs to be routed and the actual path that it takes becomes increasingly important as the size of these data centers and the complexity of the tasks that they are required to perform continues to grow.

Most data centers use hierarchical trees with small cheap edge switches simply connected to the end hosts [5]. Since these topologies are so simple to overcome the limitation in port densities for commercial switches they have to be placed in multiple layers of switches on the order of maybe two or three. As the data centers grow this no longer remains a viable option and it is suggested that the best way to allow them to grow is horizontally rather than vertically [6]. Instead of using expensive switches with incredibly high speeds and port densities they will be interconnected via multiple redundant parallel paths between any source and destination (Fig. 1).

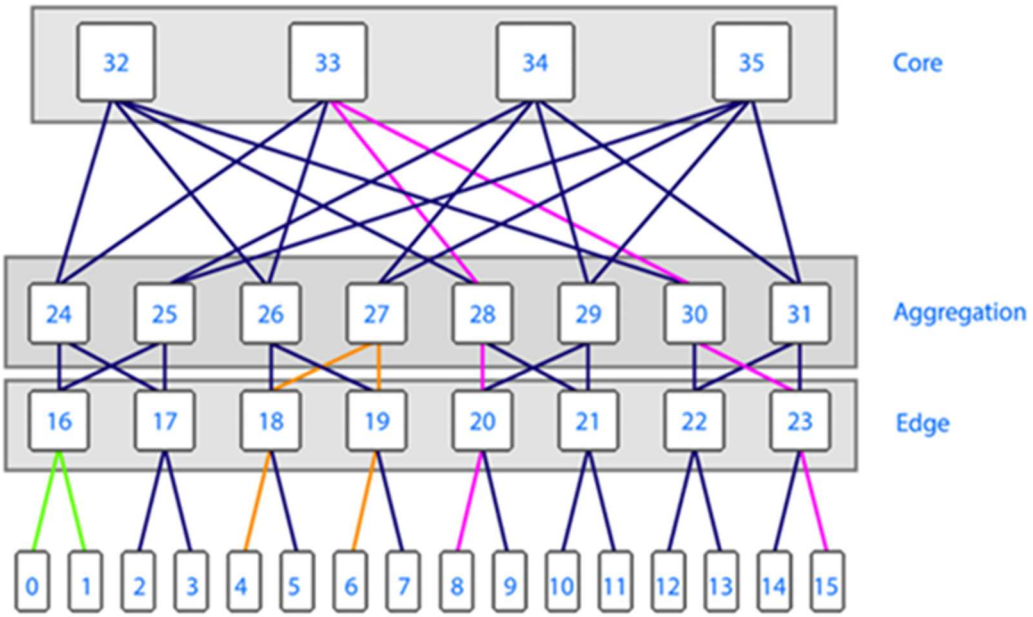

Fig. 1 k-ary fat tree where k = 4 which contains 16 physical machines showing examples of different routing paths

How to place virtual machines inside of these physical machines that reside inside of these networks is an area that is very actively being researched. Meng et al. proposed using traffic-aware virtual machine placement to improve network scalability, by assigning virtual machines with large mutual bandwidth usage to machines in close proximity [7]. Alicherry et al. considered the problem of optimal placement of computational nodes and presented algorithms for assigning virtual machines to nodes that minimize various latency metrics under different constraints [8].

SYSTEM MODEL

We chose to represent our data centers network as a fat tree as it has been proven to be an incredibly efficient topology in terms of the number of physical machines it can allocate while maintaining a relatively short path between them. The structure of the tree with basic pathing examples is given in Fig. 1. Specifically, we will be using a k-ary fat tree. Where k is a special parameter that defines many different relationships inside the tree, such as: how many pods the tree has, how many aggregation switches the tree has, how many edge switches the tree has, how many core switches the tree has, and how many physical machines the tree has.

A k-ary fat-tree has some important relationships that are worth noting and were involved for many different calculations and essential to building the simulation. There are three layers of switches. The switches in the top row are Core switches, the switches in the second row are Aggregation switches, the switches in the third row are Edge switches, and finally at the bottom are the physical machines. Core switches handle most of the data so in a fat-tree they have the largest bandwidth capacity of any other links. The bandwidth capacity of the links decreases as one moves down the tree until you reach the physical machines at the bottom which are connected to edge switches. These links have the lowest bandwidth capacity. The Aggregation and Edge switches are separated into k pods. Each pod contains k/2 aggregation switches and k/2 edge switches. Each edge switch is connected to all other Aggregation switches in its pod. There are $\frac{k^2}{4}$ core switches all of which are connected to each pod. There are $\frac{k^3}{4}$ physical machines in the bottom layer [1]. In the example tree, there only 16 physical machines, but ask grows the number of physical machines supported by the tree increases exponentially.

In terms of the connecting physical machines via the shortest path there are only four different possibilities. The physical machine is connecting to itself, the physical machine is connecting to another machine on the same edge switch, the physical machine is connecting to another physical machine on a different edge switch, but on the same pod, and finally that the physical machine is connecting to another physical machine in a different pod. These are denoted as 0 hop, 2 hops, 4 hops, and 6 hop cases respectively. The probability of each of these outcomes changes as a function of k and is given by the following relationships. A simple explanation for the derivation of each is as follows.

0 hop: If one virtual machine is placed the second virtual machine can be placed in any one of the $\frac{k^3}{4}$ physical machines. The chance that it is placed in the same physical machine is simply 1 divided by the total number of physical machines or $\frac{4}{k^3}$.

2 hops: The chances of the second virtual machine being placed on the same edge switch as the first is given by the number of physical machines per edge switch ($\frac{k}{2}$), while not going into the same physical machine(1) divided by the total number of physical machines($\frac{k^3}{4}$). This reduces to $\frac{2}{k^2} - \frac{4}{k^3}$.

4 hops: Given by the number of physical machines per pod ($\frac{k^2}{4}$) minus the number of physical machines per edge switch ($\frac{k}{2}$) divided by the total number of physical machines ($\frac{k^3}{4}$). This reduces to $\frac{1}{k} - \frac{2}{k^2}$.

6 hops: The probability of a 5-hop case is given by the probability that the virtual machine doesn't spawn in the same pod. This is given by the total number of physical machines ($\frac{k^3}{4}$) minus the number of physical machines in a pod ($\frac{k^2}{4}$) divided by the total number of physical machines ($\frac{k^3}{4}$). This reduces to 1 - $\frac{1}{k}$.

- 0 hop Case: $\frac{4}{k^3}$
- 2 hop Case: $\frac{2}{k^2} - \frac{4}{k^3}$
- 4 hop Case: $\frac{1}{k} - \frac{2}{k^2}$
- 6 hop Case: 1 - $\frac{1}{k}$

These relationships are illustrated graphically via Fig. 2.

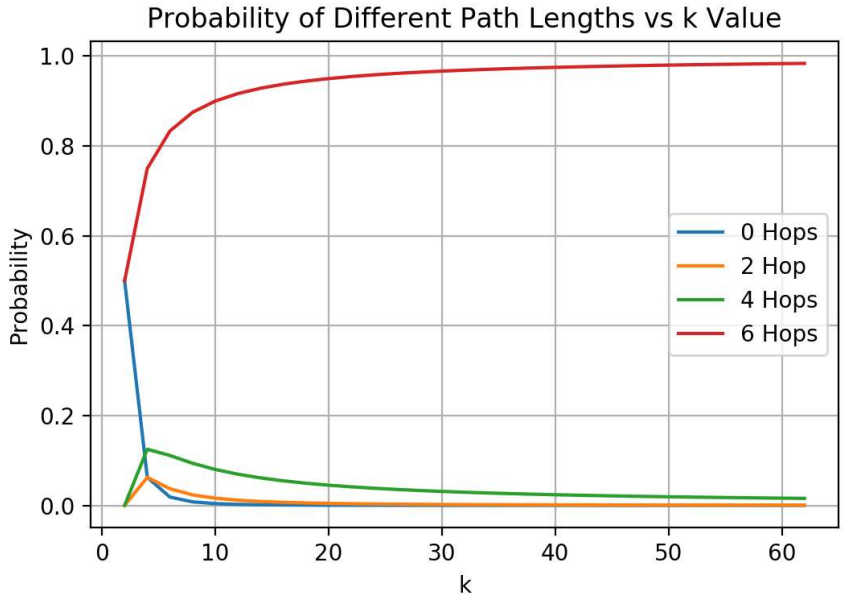

Fig. 2 Probabilities of Path Lengths vs. Value of k

Notice that all four values added together at any value of k is equal to 1. Also notice as the tree grows the probability that two physical machines taken at random are in different pods approaches 1. This may not have been obvious since every part of the tree grows as a function of k. As a consequence of this, the probability that they are in the same pod is reduced.

To better understand why the 6 hop case becomes more probable as the value of k grows large it makes sense to look at what makes the 6 hop case different than the 4,2, and 0 hop cases. What makes the 6 hop case different is that the pairs must reside in different pods, whereas with all other cases the pairs reside in the same pod. So, when trying to determine the probability of a random pair being a 6 hop pair it is necessary to find the ratio of physical machines in the same pod compared to physical machines in a different pod. Ultimately, this is given by the aforementioned relationship 1 - $\frac{1}{k}$. The number of physical machines in each pod is given by $\frac{k^2}{4}$ and the total number of physical machines in the whole tree is given by $\frac{k^3}{4}$. So, the odds that a pair will be selected that reside in different pods is simply $1 - \frac{k^2}{4} * \frac{4}{k^3}$ which reduces to 1 - $\frac{1}{k}$.

Table 1 Summary of Notation

Notation	Meaning
E	Set of all edges
k	Number of ports on each switch
n	Parameter describing the number of VM pairs allocated into the tree
b_{min}	Parameter describing the minimum value for bandwidth range per VM pair
b_{max}	Parameter describing the maximum value for bandwidth range per VM pair
B	Parameter describing the maximum allowed bandwidth per edge

CHAPTER 3

VM PAIR PLACEMENT PROBLEM

We formulate the max placement problem as follows: There are a set of n routing requests $R = \{r_1, r_2, \dots, r_n\}$ in the network where each request $r_i = (s_i, t_i)$ represents that message m_i is sent from the source node s_i to destination node $t_i, 1 \leq i \leq p$.

Let $N_i = \{r_1, r_2, \dots, r_n\}$ $(1 \leq i \leq n)$ be the routing path of message m_i, denoting the sequence of distinct edges along which m_i is routed from s_i to t_i. Let X_{ij} be the bandwidth cost incurred by edge E in routing the message m_j, and let ε_E' denote E's remaining bandwidth after all the messages are routed. Then,

$$\varepsilon'_E = \varepsilon_E - \sum_{j=1}^{n} X_{Ej}, \qquad \forall \mathrm{E} \in V_{edges}$$

Where $X_{Ej} = 1$ if $\mathrm{E} \in N_j$ and $X_{Ej} = 0$ otherwise.

The objective of max placement is to find a subset N_{sat} of the set of routing paths $N = \{N_1, N_2, \dots, N_N\}$, such that the number of messages routed in N_{sat} is maximized. i.e., $max\ |N_{sat}|$,

Under the bandwidth constraint that , $\varepsilon'_E \geq 0, \ \forall \mathrm{E} \in V_{edges}$.

CHAPTER 4

ALGORITHMIC SOLUTIONS

To solve this problem two different sets of algorithms are needed, one set of algorithms that determine in which order the virtual machine pairs are placed and one set of algorithms to find the path that each pair will take. So, one selects from a set of virtual machine pairs waiting to be placed, runs that pair through a path finding algorithm and then that pair has been allocated.

There are many different ways the order of the virtual machine pairs can be placed. For example, they could be sorted so that the lowest bandwidth pairs are placed first, they could be sorted so that the pairs with the fewest number of hops are placed first, conversely, they could be sorted so the pairs with the largest number of hops are placed first. Examples of potential path decision making algorithms include greedy algorithms, Dijkstra algorithms, and A* algorithms with different heuristics just to name a few.

The pair placement algorithms that are being used in this paper are shortest path first and random. For example, imagine four pairs are to be placed inside of Fig. 1. These pairs have path lengths of 4 hops, 6 hops, 6 hops, and 2 hops. These pairs will be sorted in the order of 2 hops, 4 hops, 6 hops, and 6 hops. Now that they have been sorted, the 2 hops pair will be processed and placed by one of the three pathfinding algorithms, then the 4 hop pair, and so on.

$$C_{ij}^{Fat-tree} = \begin{cases} 0 & \text{if } i = j \\ 2 & \text{if } \lfloor \frac{2i}{k} \rfloor = \lfloor \frac{2j}{k} \rfloor, \quad i \neq j \\ 4 & \text{if } \lfloor \frac{2i}{k} \rfloor \neq \lfloor \frac{2j}{k} \rfloor \wedge \lfloor \frac{4i}{k^2} \rfloor = \lfloor \frac{4j}{k^2} \rfloor \\ 6 & \text{if } \lfloor \frac{4i}{k^2} \rfloor \neq \lfloor \frac{4j}{k^2} \rfloor \end{cases}$$

Fig. 3 Distance between Any Two Physical Machines [7]

C in Fig. 3 represents the cost, or distance, of two physical machines labeled with indexes i and j. In a k = 4 fat tree there are 16 physical machines labeled with i,j ranging from 1 – 16. If for example, i = j then the cost is 0 because those two physical machines are the same physical machine and thus there is no distance between them.

If, i not equal j and $\lfloor \frac{2i}{k} \rfloor = \lfloor \frac{2j}{k} \rfloor$ it means that i and j are different PMs but under the same edge switch, because each edge switch connects to $\frac{k}{2}$ PMs. If, $\lfloor \frac{2i}{k} \rfloor$ not equal $\lfloor \frac{2j}{k} \rfloor$ and $\lfloor \frac{4i}{k^2} \rfloor = \lfloor \frac{4j}{k^2} \rfloor$ it means that i and j are not under the same edge switch, but they are under the same pod. This is because each pod has $\frac{k}{2}$ edge switch and each edge switch has $\frac{k}{2}$ PMs, therefore each pod connects to $\frac{k^2}{4}$PMs. Thus, if $\frac{i}{\frac{k^2}{4}} = \frac{j}{\frac{k^2}{4}}$, i and j must be under the same pod. Finally, $\lfloor \frac{4i}{k^2} \rfloor$ not equal $\lfloor \frac{4j}{k^2} \rfloor$ means i and j are under a different pod.

There are three different pathfinding algorithms that are being used. The first pathfinding algorithm is to find the shortest path between any two physical machines. Once this path is found the algorithm examines all connected edges at each hop that could potentially connect the two physical machines. It considers the potential edge with the lowest index value first and will choose it unless this edge does not have enough available bandwidth. It continues with this process until there are no more available edges

and then fails. For example, if this algorithm was finding a path for a 6 hop pair, it would choose the edge switch that one of the VMs is connected to. Next, it would examine all of the edges that connect that edge switch to aggregation switches above it and choose the first edge that it encounters that has enough bandwidth accommodate the bandwidth required for that pair. Next, it would examine all of the edges that connect that aggregation switch to core switches and choose the first edge that has enough available bandwidth to accommodate that VM pair. It then continues this process down the other side of the tree with the constraint that it must connect to the edge switch that is connected to the physical machine that the other VM of the pair resides in.

The second pathfinding algorithm is a greedy algorithm. When presented with choices between a set of edges, this algorithm will choose the edge that has the maximum available bandwidth. It continues this process until there are no more available edges and then fails. For example, if this algorithm was finding a path for a 6 hop pair, it would choose the edge switch that one of the VMs is connected to. Next, it would examine all of the edges that connect that edge switch to aggregation switches above it and choose the edge that has the maximum available bandwidth for that pair. Next, it would examine all of the edges that connect that aggregation switch to core switches and choose the edge that currently has the lowest utilized bandwidth on it and enough available bandwidth to accommodate that VM pair. It then continues this process down the other side of the tree with the constraint that it must connect to the edge switch that is connected to the physical machine that the other VM of the pair resides in.

The third pathfinding algorithm is an approximation algorithm with a provable performance guarantee. This algorithm works differently than the aforementioned

algorithms by how it determines the order in which the VM pairs are placed and how the paths for these VM pairs are found. Instead of examining the available bandwidth on each edge in the graph it instead examines a more abstract value which we call the weight. After an edge has had bandwidth allocated to it the weight on the edge is increased. The weight on an edge is multiplied by the weighting factor

$$W_f = M^{\frac{1}{B+1}}$$

Where B represents the maximum bandwidth capacity of the edge and M represents the total number of edges in the graph, which is given by the relationship $\frac{3k^3}{4}$. To calculate the total number of edges in the graph one approach is to count all the edges connected to core switches, aggregation switches, and edge switches then add them together. Each pod consists of $\frac{k^2}{4}$ servers and 2 layers of $\frac{k}{2}$ switches, Aggregation and Edge. Each edge switch connects to $\frac{k}{2}$ servers and $\frac{k}{2}$ aggregation switches. Each Aggregation switch connects to $\frac{k}{2}$ edge and $\frac{k}{2}$ core switches. Each Core switch $\frac{k^2}{4}$ connects to k pods. Each of these 3 switch sets contributes $\frac{k^3}{4}$ edges for a grand edge total of $\frac{3k^3}{4}$. As each edge is used the same edge is disincentivized from being used again.

The method of operation for the Approximation Algorithm is as follows: First it assigns an edge weight of one to each edge. Then all VM pairs are sorted by lowest total weighted path first. It does this by running a Dijkstra algorithm on all VM pairs while examining the weight on every edge. It places all of these pairs into a priority queue with the VM pair that has the lowest overall weight being on top. It then places the lowest overall weight VM pair into the graph. After checking that all of necessary edges can support the bandwidth

required by the pair, it places the pair into the network. All of the used edges have their bandwidths and weights updated. This process continues until all VM pairs are placed or until there is no possible path that can accommodate the bandwidth of a VM pair.

The significance of the Approximation Algorithm is that it is an approximation algorithm, but more importantly that it has a performance guarantee. The total number of satisfiable requests by this approximation algorithm is at least $B * M^{\frac{1}{B+1}}$ times the maximum number of satisfiable requests in the optimal solution. This is provable and is proved in great detail in "Algorithm Design" by J. Kleinberg [9].

In addition to the previous three algorithms a fourth algorithm will analyzed that was created by other authors. This algorithm is called the BlockingIsland algorithm and works similarly to the most bandwidth first algorithm. The way this algorithm works is as follows: it sorts all the VM pairs by maximum minimum-available bandwidth edge in descending order. If there are ties, shortest path first. If there are still ties, highest bandwidth demand first. It then places each VM pair along its shortest path, choose one randomly if there are multiple.

To calculate the algorithmic complexity, it is useful to break up the possible shortest paths into 3 different cases. That is how many shortest paths between two PMs i and j if they are under the same edge switch, the same pod, and different pod. When two PMs are under the same edge switch, it has only one shortest path. Otherwise, when two PMs are under the same POD, it has $\frac{k^2}{4}$ shortest paths, as there are $\frac{k}{2}$ edge switches and $\frac{k}{2}$ aggregation switches, and every edge switch connects to every aggregation switch in a POD. Otherwise,

when two PMs are under different PODs, it has $\frac{k^2}{4}$ shortest paths, as there are $\frac{k^2}{4}$ core switches and each one sits on a shortest path connecting two PMs from different PODs.

Using a combination of the methods mentioned above. The first algorithm will place VM pairs by the Shortest Path First Left Most Edge First. The pseudocode is given below:

Algorithm: SP_LF: Shortest Path, Left Most Edge First
Input: all virtual machine pairs
Output: places all virtual machine pairs

```
1. sort all vm pairs by shortest path O(nlogn)
2. for (all virtual machine pairs)
3.     if (edge can accommodate bandwidth)
4.         Place current VM pair on edge
5.         Update bandwidth on current edge
6.     end if;
7. end for;
```

Fig. 4 Pseudocode for SP_LF Algorithm

When finding the path for each VM pair, it just finds the first available shortest path maintained for this pair. As there are n VM pairs, and each pair will look for at most $\frac{k^2}{4}$ shortest paths, then the time complexity of Fig. 4 is $\frac{nk^2}{4}$.

The second algorithm will place vm pairs by the Shortest Path First on the edge that has the most available bandwidth. The pseudocode is given below:

Algorithm: SP_MBF: Shortest Path, Most Available Bandwidth First
Input: all virtual machine pairs
Output: places all virtual machine pairs

1. sort all vm pairs by shortest path O(nlogn)
2. **for** (all virtual machine pairs)
3. **if** (edge can accommodate bandwidth and edge has most available bandwidth of any available edge)
4. Place current VM pair on edge
5. Update bandwidth on current edge
6. **end if;**
7. **end for;**

Fig. 5 Pseudocode for SP_MBF Algorithm

Similarly, to the previous algorithm. When finding the path for each VM pair, it just finds the first available shortest path maintained for this pair. As there are n VM pairs, and each pair will look for at most $\frac{k^2}{4}$ shortest paths, then the time complexity of Fig. 5 is $\frac{nk^2}{4}$.

For each VM pair to be satisfied, if they are located under different edge switches (either in the same POD or not), then there are $\frac{k^2}{4}$ shortest paths between those two PMs they are located in. It then checks all of these $\frac{k^2}{4}$ shortest paths and chooses one whose minimum-available-bandwidth edge has the maximum available bandwidth among all the $\frac{k^2}{4}$ shortest paths. The time complexity again is $\frac{nk^2}{4}$.

The third algorithm will place the lowest total weighted pairs first where every utilized edge has weight that is updated every cycle by the relationship

$$W_f = M^{\frac{1}{B+1}}$$

The pseudocode is given below:

Algorithm: Approximation Algorithm
Input: all virtual machine pairs
Output: places all virtual machine pairs
Notations: $W_f = M^{\frac{1}{B+1}}$

1. **for** (all virtual machine pairs)
2. search for lowest weight pair
3. **if** (all edges can accommodate required bandwidth)
4. place lowest total weight pair
5. update all used edges with current VM pairs bandwidth
6. update all used edges with W_f of current VM pair
7. **end if;**
8. **end for;**

Fig. 6 Pseudocode for Approximation Algorithm

Theorem: The Approximation Algorithm is a $2BM^{\frac{1}{B+1}}$ approximation algorithm. That is, the total number of satisfiable requests by GDP is at least $2BM^{\frac{1}{B+1}}$ times of the maximum number of satisfiable requests in optimal solution. A proof of this is given in detail in Maximizing Number of Satisfiable Routing Requests in Static Ad Hoc Networks [11].

Assume you will n as number of VM pairs. As there are at most n rounds (each round it satisfies one VM pair). In each round it finds among at most n VM pairs one minimum weighted VM pair that can be satisfied. To calculate minimum weight for each VM pair, it takes O(|E| + |V| log |V|). Thus, total time complexity is n^2(|E| + |V| log |V|). Next let's find |E|, number of edges in the network and |V|, number of nodes in the network.

Edges between PMs and edge switches: $\frac{k^3}{4}$

Edges between edge and aggregation switches: $\frac{k^3}{4}$

Edges between core and aggregation switches: $\frac{k^2}{4} * k = \frac{k^3}{4}$

Total edges = $\frac{3k^3}{4}$

Number of nodes |V| is $\frac{k^2}{4}$ core switches + $\frac{k^2}{4}$ aggregation switches + $\frac{k^2}{4}$edge switches + $\frac{k^3}{4}$

PMs = $\frac{3k^2}{4} + \frac{k^3}{4}$

Therefore the total time complexity is n^2*($\frac{3k^3}{4} * \frac{3k^2}{4} + \frac{3k^3}{4}$ log ($\frac{3k^2}{4} + \frac{k^3}{4}$))= O($n^2$ * (k^3+ k^3 log k^3)

= O(n^2 * (k^3 * log(k^3))

The Pseudocode for the BlockingIsland algorithm is given below:

Algorithm: BI: BlockingIsland
Input: all virtual machine pairs
Output: places all virtual machine pairs

1. Sorts all the VM pairs by maximum minimum-available bandwidth edge in descending order. If there are ties, shortest path first. If there are still ties, highest bandwidth demand first. O(nlogn)
2. **for** (all virtual machine pairs)
3. **if** (path is shortest path)
4. Place current VM pair on edge
5. Update bandwidth on current edge
6. **end if;**
7. **else if (**multiple paths choose path randomly**)**
8. Place current VM pair on edge
9. Update bandwidth on current edge
10. **end if**;
11. **end for;**

Fig. 7 Pseudocode for BlockingIsland Algorithm

When finding the path for each VM pair Sorts all the VM pairs by maximum minimum-available bandwidth edge in descending order. If there are ties, shortest path first. If there are still ties, highest bandwidth demand first. As there are n VM pairs, and each pair will look for at most $\frac{k^2}{4}$ paths, then the time complexity of Fig. 7 is $\frac{nk^2}{4}$.

For each VM pair to be satisfied, if they are located under different edge switches (either in the same POD or not), then there are $\frac{k^2}{4}$ shortest paths between those two PMs they are located in. It then checks all of these $\frac{k^2}{4}$ shortest paths and chooses one whose minimum-available-bandwidth edge has the maximum available bandwidth among all the $\frac{k^2}{4}$ shortest paths. The time complexity again is $\frac{nk^2}{4}$.

CHAPTER 5

PERFORMANCE EVALUATION

When evaluating the performance of the three different algorithms a number of different situations were examined. The simulation has five different parameters, k, the number of VM pairs, the minimum bandwidth per pair, the maximum bandwidth per pair, and the bandwidth capacity for each edge; These parameters are denoted by k, n, b_{min}, b_{max}, and b, respectively. To examine the effectiveness of each algorithm the amount of VM pairs that are accommodated into the network will be evaluated with different sets of parameters.

As well as the ability of each algorithm to place VM pairs inside of the network the algorithms energy consumption will also be tested. The energy consumption is defined as the number of hops needed to facilitate the connection between the virtual machine pair. This amount will vary based on how many pairs are placed and the length of the path needed to connect the virtual machine pair.

For the first simulation the independent parameter will be k and the dependent variable is the percentage of VM pairs that can be accommodated by each algorithm. The capacity per edge in this simulation is 10 Gbps and the bandwidth per VM pair is 500 Mbps. The number of physical machines depends on the value of k for the tree and is given by the expression $\frac{k^3}{4}$. To properly load the network structure the number of VM pairs spawned depends on the number of physical machines and is given by the expression $\frac{10k^3}{4}$. So, for each physical machine there are ten VM pairs.

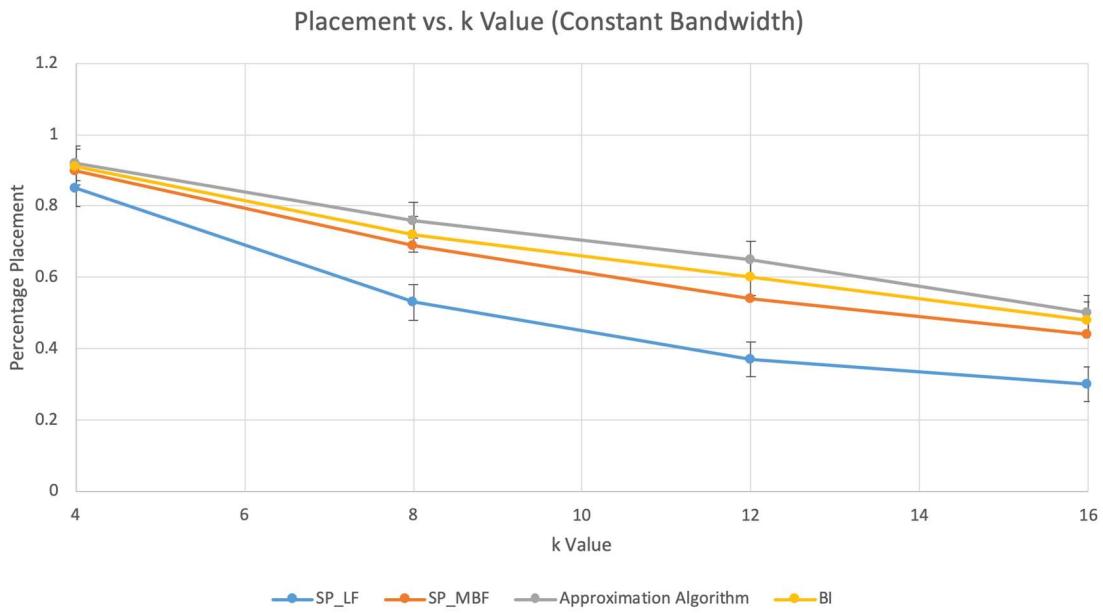

Fig. 8 VM pair placement vs k for a 500 Mbps bandwidth per pair

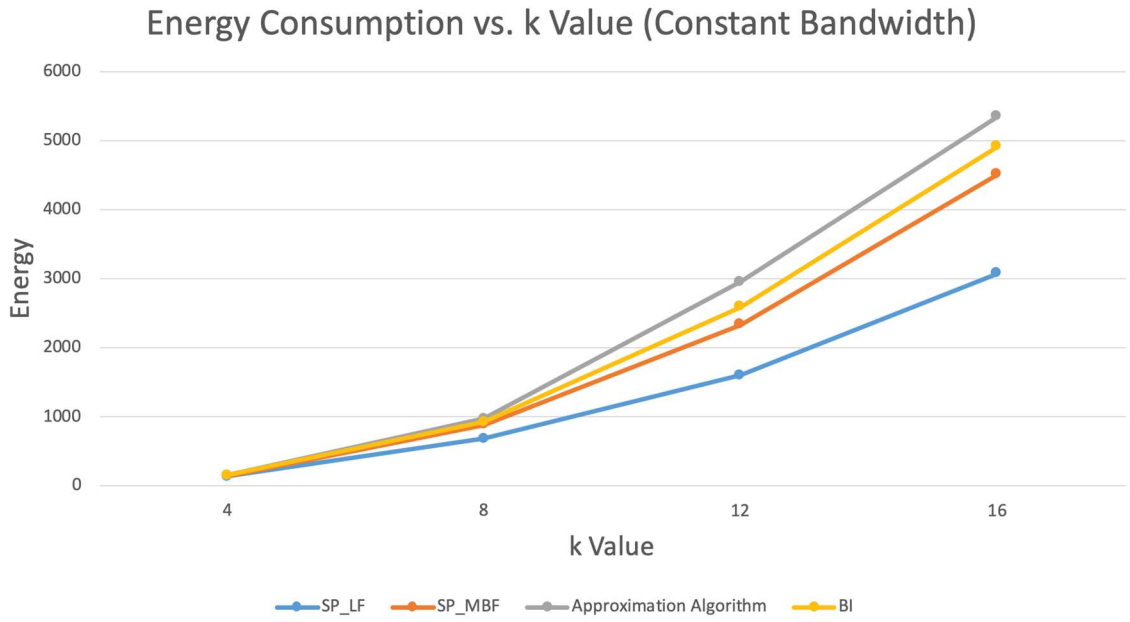

Fig. 9 Energy Consumption for VM pair placement vs k for a 500 Mbps bandwidth per pair

The first simulation is described via Fig. 8. The number of VM pairs in this simulation depends on the value of k. The relationship $\frac{10^{\ 3}}{4}$ describes how many VM pairs are attempted to be placed by each algorithm. As one would expect has the value of k goes

up and the number of VM pairs increases each algorithms ability to accommodate pairs into the network is reduced significantly. What is interesting to note however is that the SP_LF algorithm consistently cannot accommodate as many pairs as the other two algorithms. Due to the increased complexity of the Approximation algorithm it makes sense that it out performs all other algorithms at every value of k. The Blocking Island algorithm performs better than the most bandwidth first algorithm, but less than the approximation algorithm.

The energy consumption for these simulation conditions performs as expected due to the fact that the number of virtual machine pairs placed scales exponentially with the size of k. However, in spite of this the approximation algorithm consumes the most amount of energy as it consistently places the largest number of virtual machine pairs and each of these pairs as a longer than average path length.

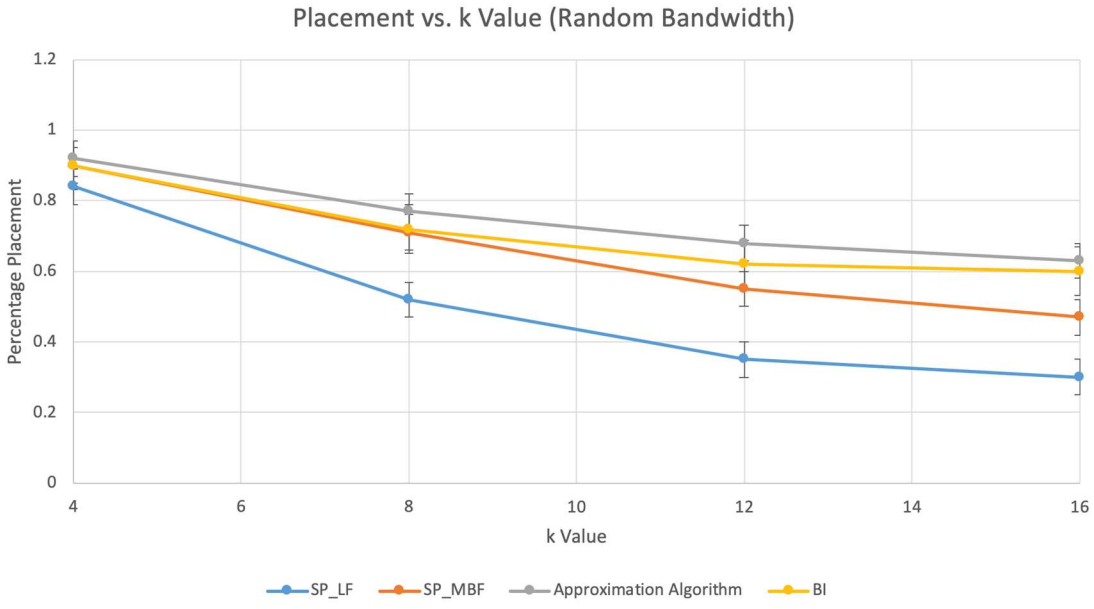

Fig. 10 VM pair placement vs. value of k with a random bandwidth ranging from 1 Mbps to 500 Mbps

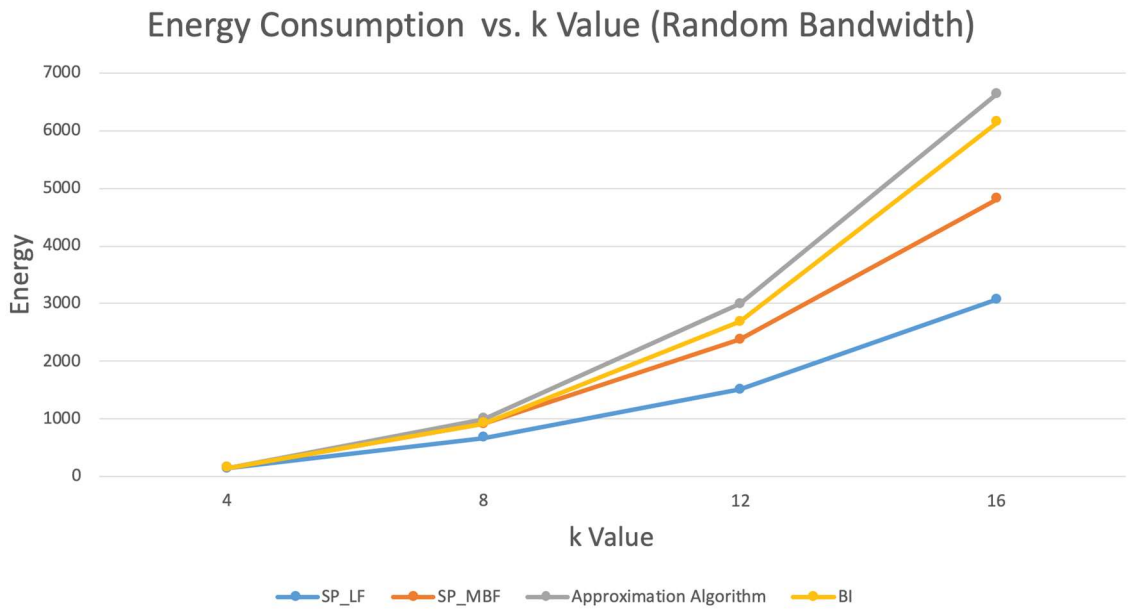

Fig. 11 Energy Consumption for VM pair placement vs. value of k with a random bandwidth ranging from 1 Mbps to 500 Mbps

Examine Fig. 10. Similar to Fig. 8, except the difference here is that the bandwidth per VM pair is varied randomly between 1 Mbps and 500 Mbps. Again, the SP_LF algorithm consistently performs the worst. Again, the Approximation algorithm seems to out performs the other two. The difference between the algorithms appears to be very linear in this situation and evolves as one would expect as the value of k increases.

Similarly, to the last simulation the approximation algorithm consumes the most amount of energy as it consistently places the most amount of virtual machine pairs and for the pairs that it places the average path length is longer. Again the Blocking Island algorithm performs better than the most bandwidth first algorithm, but less effectively than the approximation algorithm.

For the next set of simulations, it makes sense to observe how the algorithms behave when different parameters are modulated. Next the case where the bandwidth per

pair, will be modulated with the value of k being 12, the bandwidth capacity being 10 Gbps, and the number of VM pairs given by $\frac{10k^3}{4}$

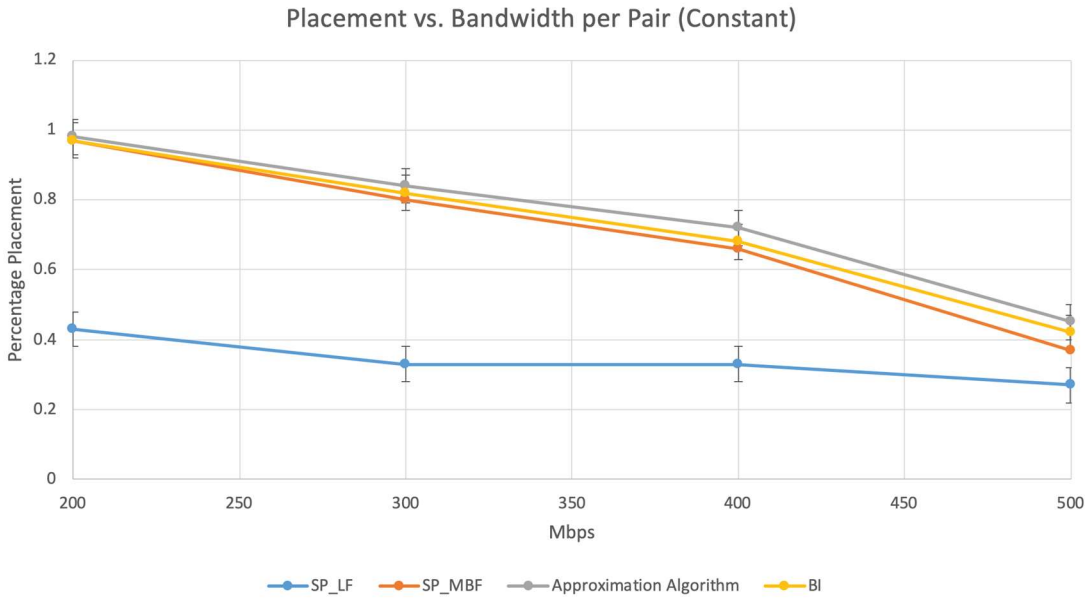

Fig. 12 VM pair placement vs. Bandwidth per Pair (Constant)

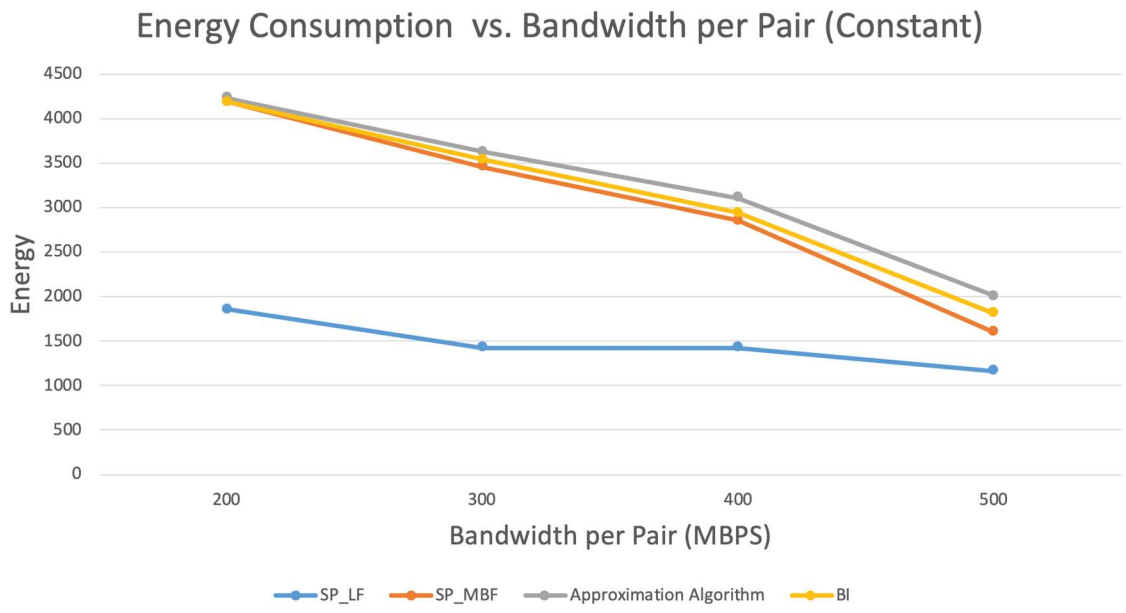

Fig. 13 Energy Consumption for VM pair placement vs. Bandwidth per Pair (Constant)

Examine Fig. 12. For this simulation the value of k is 12 and the value of n is $\frac{10k^3}{4} = 4320$. Similarly, the SP_LF algorithm performs significantly worse than the other two algorithms. The SP_MBF appears to perform better when the bandwidth per pair is held constant and worse when the bandwidth per pair is modulated randomly. In a similar fashion the bandwidth per pair will be modulated except it will now be random within a range as opposed to simply being constant. All algorithms perform similarly, with the approximation algorithm consistently performing the best.

The energy consumption behaves similarly to the placement. Due to the fact that the number of virtual machine pairs is fixed it follows closely to the number of pairs placed by each algorithm with the exception of the approximation algorithm with a slightly higher than expected energy consumption due to its larger than average path length and higher number of placed pairs.

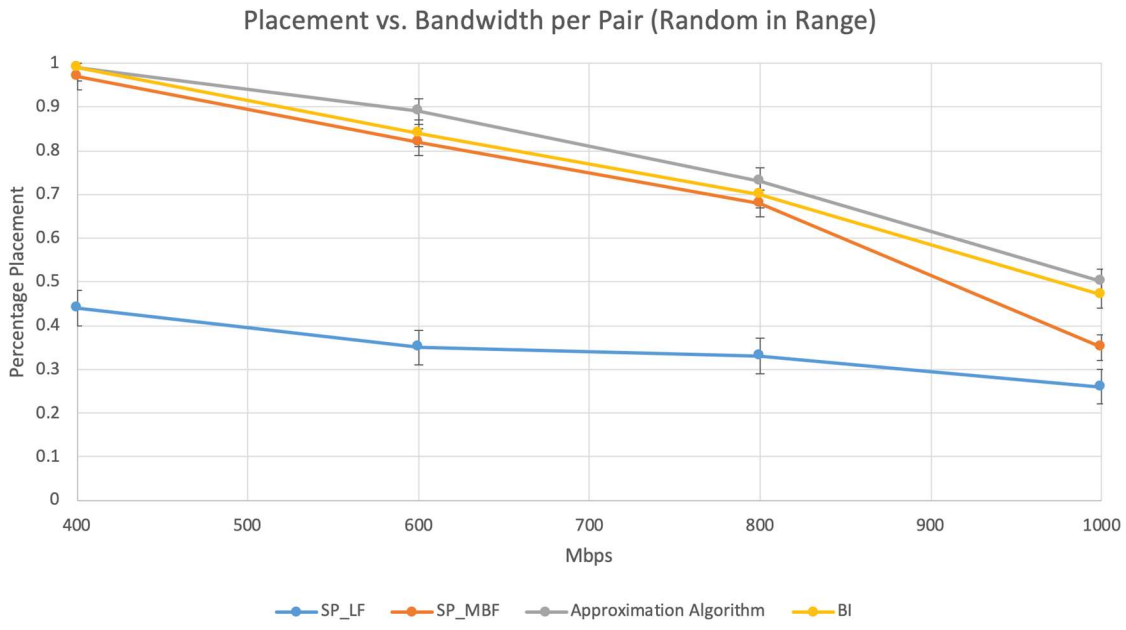

Fig. 14 VM pair placement vs. Bandwidth per Pair (Random between 1 and value in Mbps)

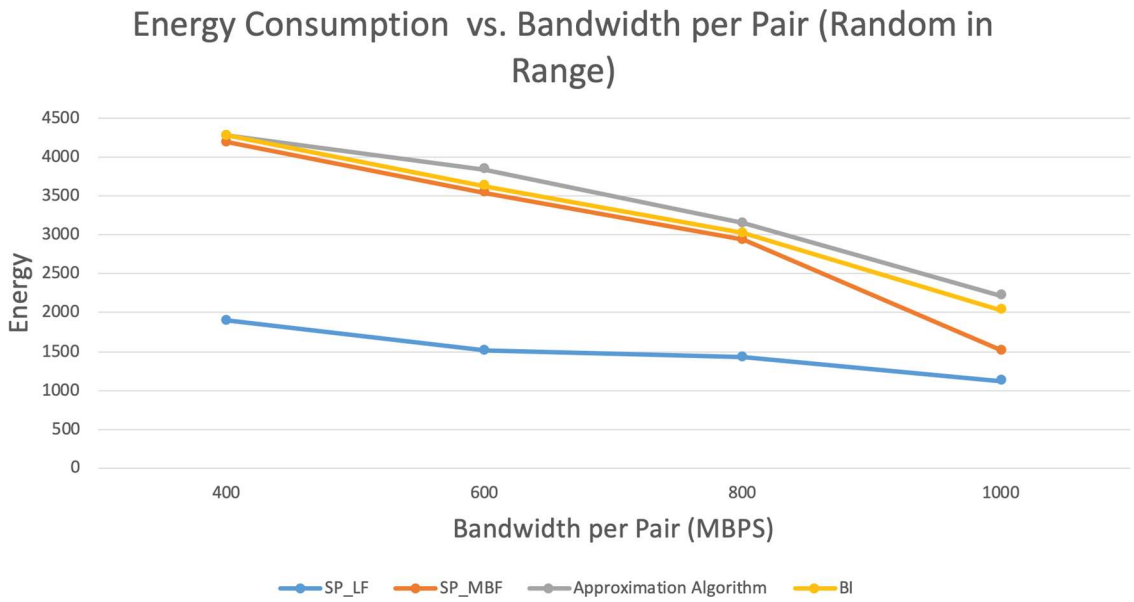

Fig. 15 Energy Consumption for VM pair placement vs. Bandwidth per Pair (Random between 1 and value in Mbps)

Examine Fig. 14. For this simulation the value of k is 12 and the value of n is $\frac{10k^3}{4} =$ 4320. Again, the SP_LF algorithm performs consistently worse than the other two and all algorithms perform worse as the average bandwidth per pair increases. What is interesting however, is that the Approximation Algorithm consistently performs better than the SP_MBF algorithm. In fact, when the value of bandwidths is random within a range the effectiveness of the two algorithms is linear but diverge. That is as the random bandwidth per pair increase the increased effectiveness of the Approximation Algorithm is proportionate to the increased bandwidth. Like in the previous simulation the placement of all the algorithms is fairly close, but the Approximation Algorithm performs consistently better.

Again, the approximation algorithm consumes the most amount of energy because it places the largest number of pairs as well as having a slightly than higher average path length.

Next the number of VM pairs will be modulated and the effectiveness of the three algorithms will be evaluated in similar fashion. The value of k will be set to 12 and the bandwidth will be set to 400 Mbps with the capacity for each edge being held at 10 Gbps. With constant bandwidth being examined before a random bandwidth range.

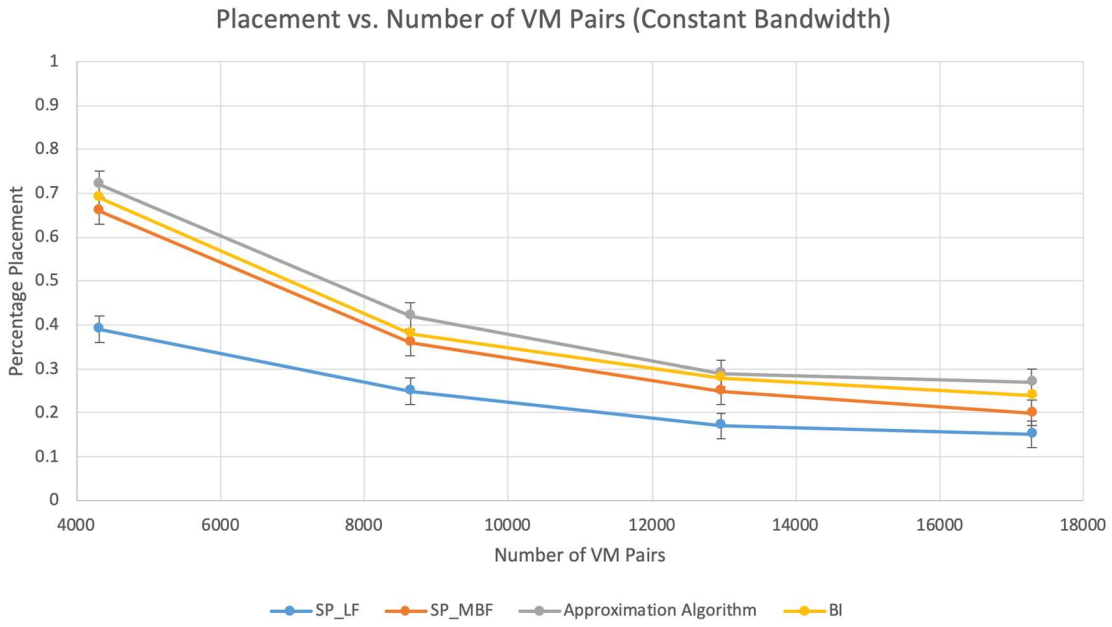

Fig. 16 Placement vs. Number of VM Pairs (Constant Bandwidth 400 Mbps)

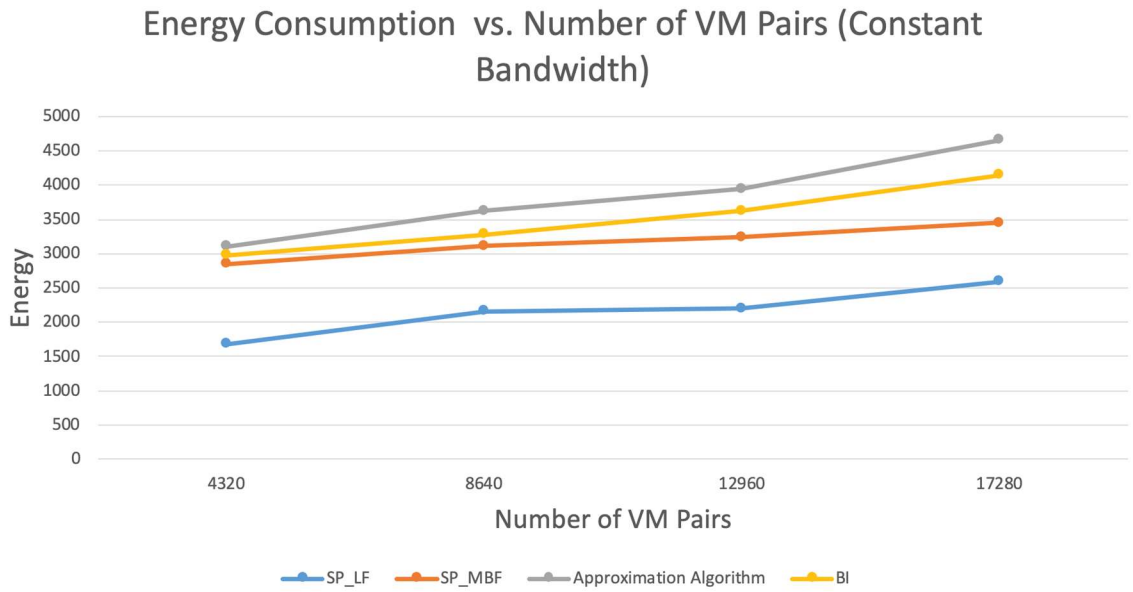

Fig. 17 Energy Consumption for Placement vs. Number of VM Pairs (Constant Bandwidth 400 Mbps)

Examine Fig. 16. The SP_LF algorithm performs significantly worse than the other two and similarly all algorithms perform worse as the number of attempted accommodated VM Pairs increases. Like in the other simulations with constant bandwidth the SP_MBF and Approximation algorithm perform similarly with the Approximation Algorithm performing marginally better. The situation where each pair is randomly assigned bandwidth within a certain range will be examined next.

The energy consumption of this plot is more interesting because as percentage of placed pairs decreases the number placed pairs increases. What is most interesting is as the number of potential pairs increases the average energy consumption increases. Perhaps this is due to the fact that each algorithm can place more desirable pairs.

In the next simulation the parameters will be similar to the previous with the exception that the bandwidth range will be distributed randomly between 1 and 800 Mbps.

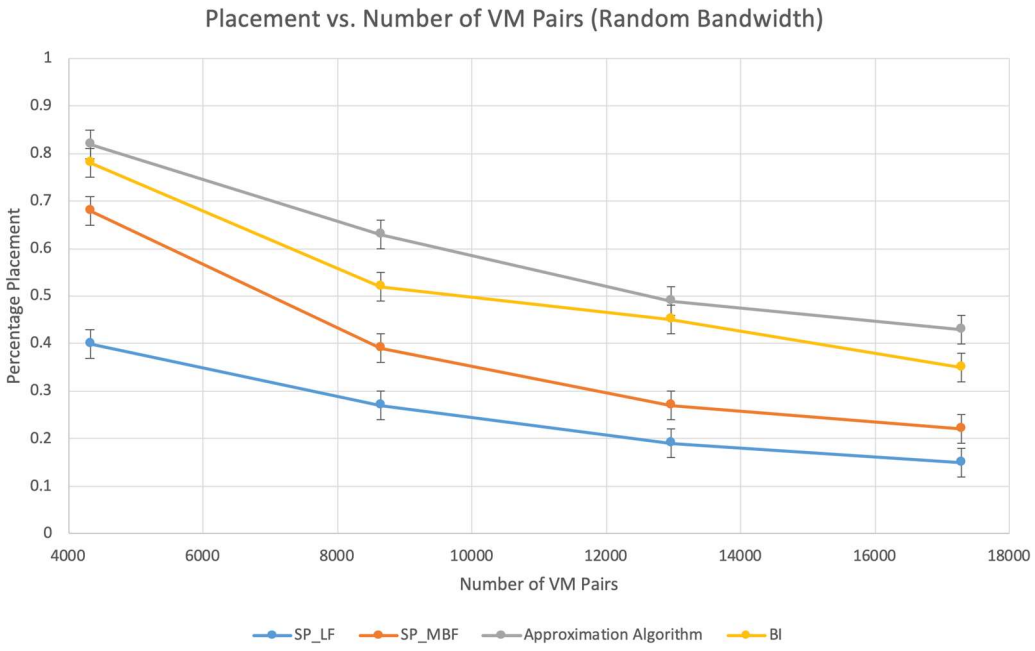

Fig. 18 VM pair placement vs Number of VM pairs (Random Bandwidth between 1-800 Mbps)

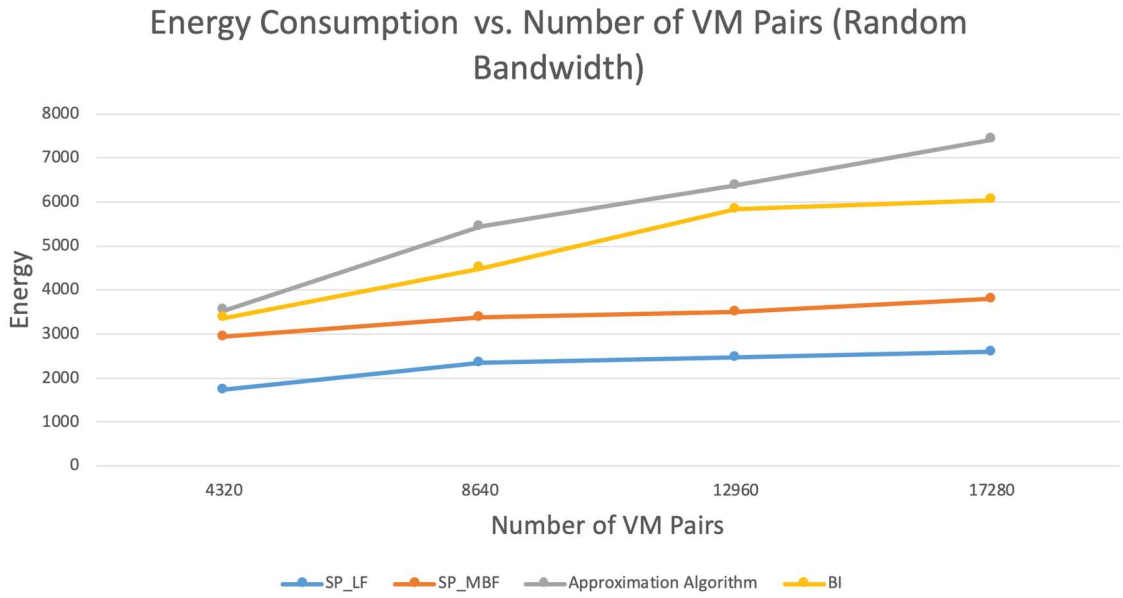

Fig. 19 Energy Consumption for VM pair placement vs Number of VM pairs (Random Bandwidth between 1-800 Mbps)

Examine Fig. 18. Again, the SP_LF algorithm consistently performs the worst. The other two algorithms perform similarly at the lower end of the scale, but as the number of VM pairs increases the effectiveness of algorithms two and three start to diverge. The

magnitude of the divergence increases as the number of VM pairs increases, which seems to be similar to the other simulated parameter sets involving random bandwidth per VM pair.

Similarly, to the previous simulation the approximation algorithm has the highest energy consumption, but as the percentage of placed pairs goes down the energy consumption goes up. This may be due to the fact that having more pairs allows the more desirable pairs to be placed on average increases the total effectiveness and energy consumption of the network.

CHAPTER 6

CONCLUSION AND FUTURE WORK

We have explored different methods of accommodating virtual machine pairs inside physical machines in a k-ary fat tree topology and have found that there are many algorithms that solve this problem, however the Approximation Algorithm performed the best on average. This algorithm seemed to be less optimum when the bandwidth allocated per pair was constant. It is important to note however that the algorithmic complexity of the Approximation Algorithm is given by $O(n^5)$ where the complexity of the other two algorithms is given by $O(n^{\frac{8}{3}})$.

It is also worth noting that as the size of k-ary tree grows the number of 6 hop cases increases. This relationship is illustrated via Fig. 2. Sorting by the path length becomes less and less significant as the tree grows larger. This is shown through our analysis of different simulation parameters with the Approximation Algorithm outperforming the SP_MBF algorithm more significantly as the size of the tree grows.

For the future, we would like to examine the same structure, but with more realistic conditions. Perhaps limiting the computation capacity of the physical machines or applying additional restrictions in our network for example, having to visit different dedicated middle boxes at different locations in the network.

REFERENCES

[1] P. Khani, B. Tang, J. Han and M. Beheshti, "Power-efficient virtual machine replication in data centers," *2016 IEEE International Conference on Communications (ICC)*, Kuala Lumpur, 2016, pp. 1-7.

[2] Al-Fares, Mohammad & Radhakrishnan, Sivasankar & Raghavan, Barath & Huang, Nelson & Vahdat, Amin. (2010). Hedera: Dynamic Flow Scheduling for Data Center Networks.. 281-296.

[3] Ghemawat, Sanjay & Gobioff, Howard & Leung, Shun-Tak. (2003). The Google File System. ACM SIGOPS Operating Systems Review. 37. 29-43.

[4] Benson, Theophilus & Anand, Ashok & Akella, Aditya & Zhang, Ming. (2009). Understanding Data Center Traffic Characteristics. Computer Communication Review - CCR. 40. 65-72.

[5] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. 2005. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*(NSDI'05), Vol. 2. USENIX Association, Berkeley, CA, USA, 15-28.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. 2008. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.* 38, 4 (August 2008), 63-74. DOI: https://doi.org/10.1145/1402946.1402967

[7] X. Meng, V. Pappas, and L. Zhang. 2010. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the 29th conference on Information communications* (INFOCOM'10). IEEE Press, Piscataway, NJ, USA, 1154-1162.

[8] Alicherry, Mansoor & Lakshman, T.V.. (2013). Optimizing Data Access Latencies in Cloud Systems by Intelligent Virtual Machine Placement. Proceedings - IEEE INFOCOM. 647-655. 10.1109/INFCOM.2013.6566850.

[9] J. Kleinberg and E. Tardos, “Algorithm Design”, Addison Wesley, 2005.

[10] https://www.theatlas.com/charts/E1Wxox0c

[11] Z. Sumpter, L. Burson, B. Tang, and X. Chen. 2013. Maximizing Number of Satisfiable Routing Requests in Static Ad Hoc Networks. IEEE Global Communications Conference (GLOBECOM)

APPENDICES

THE SP_LF ALGORITHM CODE

```
public static double shortestPathOpenEdge() {
    cleanEdgeBandwidth();

    //System.out.println("Shortest Pair First Open Edge");
    Integer[][] VmPairHops = new Integer[1][2];

    for (int i = 0; i < 1; i++) {
        VmPairHops[i][0] = i;
        VmPairHops[i][1] = hopsPerPair(i);
    }

    //sorts by number of hops
    Arrays.sort(VmPairHops, new Comparator<Integer[]>() {
        @Override
        public int compare(Integer[] entry1, Integer[] entry2) {
            Integer time1 = entry1[1];
            Integer time2 = entry2[1];
            return time1.compareTo(time2);
        }
    });

    int count = 0;
    ArrayList<Integer> path;
    for (int i = 0; i < 1; i++) {
        path = findOpenSwitchPath(VmPairHops[i][0]);
        if (path == null) {

        } else {
            count++;

        }

    }

    double percentagePlaced = (double) count / 1;

    return percentagePlaced;
}
```

THE SP_MBF ALGORITHM CODE

```
public static double shortestPathBandwidthEdge() {
    cleanEdgeBandwidth();

    //System.out.println("Shortest Pair First Bandwidth Edge");
    Integer[][] VmPairHops = new Integer[l][2];

    for (int i = 0; i < l; i++) {
        VmPairHops[i][0] = i;
        VmPairHops[i][1] = hopsPerPair(i);
    }

    //sorts by number of hops
    Arrays.sort(VmPairHops, new Comparator<Integer[]>() {
        @Override
        public int compare(Integer[] entry1, Integer[] entry2) {
            Integer time1 = entry1[1];
            Integer time2 = entry2[1];
            return time1.compareTo(time2);
        }
    });

    int count = 0;
    ArrayList<Integer> path;
    for (int i = 0; i < l; i++) {
        path = findPath(VmPairHops[i][0]);
        if (path == null) {

        } else {
            count++;

        }

    }
    double percentagePlaced = (double) count / l;

    return percentagePlaced;
}
```

THE APPROXIMATION ALGORITHM CODE

```
public static double approximationAlgorithm() {
    cleanEdgeBandwidth();
    PriorityQueue<VirtualMachine> orderedVms = new PriorityQueue<>();
    int count = 0;

    //finding initial paths without adding bandwidth
    ArrayList<Integer> path;
    for (int i = 0; i < l; i++) {
        VirtualMachine v = new VirtualMachine(i, allEdges, vmBandwidth[i]);
        v.setPath(findPath3(i));
        orderedVms.offer(v);
    }

    //placing the VMs, adding the bandwidth, and reordering after every placement
    for (int i = 0; i < l; i++) {
        if (orderedVms.peek().getPath() == null) {
            orderedVms.poll();
            continue;
        }

        if (orderedVms.poll().useEdges()) {
            for (VirtualMachine v : orderedVms) {
                v.checkPath();
            }
            count++;
        }
    }

    return (double) count / l;
}
public static Stack<Integer> findPath3(int input) {
    double currentBestWeight = Double.MAX_VALUE;
    Stack<Integer> currentBestSolution = null;

    int[] locations = findVmLocation(input);

    if (locations[0] == locations[1]) {
        return new Stack<>();
    }

    ArrayList<Stack<Integer>> on = new ArrayList<>();
    ArrayList<Stack<Integer>> off = new ArrayList<>();

    Stack<Integer> firstValue = new Stack<>();
    firstValue.push(locations[0]);
    on.add(firstValue);
```

```
    while (!on.isEmpty()) {

        Stack<Integer> currentNodeStack = on.remove(0);

        //adds all nodes the current node is connected to
        for (Edge e : allEdges) {
            if (currentNodeStack.peek() == e.getNode1() || currentNodeStack.peek() == e.getNode2()
                    && currentNodeStack.size() < MAX_PATH_LIMIT && !offStackContains(off, currentNodeStack.peek())) {
                if (e.getNode1() != currentNodeStack.peek() && !currentNodeStack.contains(e.getNode1())) {

                    Stack<Integer> tempStack = (Stack<Integer>) currentNodeStack.clone();
                    tempStack.push(e.getNode1());
                    on.add(tempStack);
                } else if (e.getNode2() != currentNodeStack.peek() && !currentNodeStack.contains(e.getNode2())) {
                    Stack<Integer> tempStack = (Stack<Integer>) currentNodeStack.clone();
                    tempStack.push(e.getNode2());
                    on.add(tempStack);
                }
            }
        }

        off.add(currentNodeStack);

        //checks to see if current stack is better solution than previous stack
        if (currentNodeStack.contains(locations[1]) && currentNodeStack.contains(locations[0])) {

            double currentWeight = checkPathWeight(currentNodeStack);
            if (currentWeight < currentBestWeight) {
                currentBestSolution = currentNodeStack;
                currentBestWeight = currentWeight;
            }

        }

    }

    return currentBestSolution;
}
```

THE BLOCKINGISLAND CODE

```
public static double[] blockingIsland() {
    cleanEdgeBandwidth();
    PriorityQueue<VirtualMachine> orderedVms = new PriorityQueue<>(new BandwidthCompVM());
    ArrayList<VirtualMachine> vms = new ArrayList<>();


    int count = 0;
    int energyConsumption = 0;

    //finding initial paths without adding bandwidth
    ArrayList<Integer> path;
    for (int i = 0; i < l; i++) {
        VirtualMachine v = new VirtualMachine(i, allEdges, vmBandwidth[i]);
        v.setPath(findPathTest(i, v.getBandwidth(), new BandwidthComp()));
        orderedVms.offer(v);
    }

    //placing the VMs, adding the bandwidth, and reordering after every placement
    for (int i = 0; i < l; i++) {
        VirtualMachine tempvm = orderedVms.peek();

        if (orderedVms.peek().getPath() == null) {
            orderedVms.poll();
            continue;
        }


        if (orderedVms.poll().useEdges()) {
            energyConsumption += (hopsPerPair(tempvm.getPairNumber()) - 2);

            //System.out.println("Worked on this one... " + orderedVms.poll());
            for (VirtualMachine v : orderedVms) {
                v.checkPathImp();
            }
            count++;

        }
    }

    double[] returnArray = new double[2];
    returnArray[0] = (double) count / l;
    returnArray[1] = (double) energyConsumption / 2;

    return returnArray;
}
```

K-ARY TREE CONSTRUCTION CODE

The initializeEdges function creates all of the connections inside of the tree. This is one of the three functions that creates all the static objects that will be used in the simulation. This is not a trivial task as there are many specific rules for which objects (switches and physical machines) are connected to each other. It also creates the edge objects which have their own class and many internal variables.

The initializeValue function creates all of the objects that the edges connect to. This is one of the three functions that creates all the static objects that will be used in the simulation. These objects include the three different switch types and the physical machines that hold the virtual machines. These objects all have their own properties and variables that define their behavior.

The initializeVirtualMacine function creates all of the virtual machines and places them inside physical machines and assigns them the bandwidth required per pair. This is one of the three functions that creates all the static objects that will be used in the simulation.

```
public static void initializeEdges() {
    //simple constants to make loops appear cleaner
    int a = cube(k) / 4;
    int b = a + square(k) / 2;
    int c = square(k) / 2;
    int d = k / 2;
    int numEdges = 3 * cube(k) / 4;

    //initialize PM to Edge switch edges
    for (int i = 0; i < a; i++) {
        allEdges[i] = new Edge(i, a + (i / d), bandwidthCapacity, numEdges);
    }

    //initialize Edge switch to Aggregation switch edges
    int count = a;
    int count2 = 0;
    for (int i = a; i < b; i++) {
        for (int j = 0; j < d; j++) {
            allEdges[count] = new Edge(i, b + j + count2, bandwidthCapacity, numEdges);
            count++;
        }
        if ((i + 1) % d == 0) {
            count2 += d;
        }
    }

    //initialize Aggregation switch edges to Core switch edges
    count2 = 0;
    for (int i = b; i < a + square(k); i++) {
        for (int j = 0; j < d; j++) {
            allEdges[count] = new Edge(i, a + square(k) + count2, bandwidthCapacity, numEdges);
            count++;
            count2++;
        }
        if ((i + 1) % d == 0) {
            count2 = 0;
        }
    }
}
```

```
public static void initializeValues() {

    //initialize elements array with total number of elements
    allNetworkElements = new Object[((5 * square(k)) / 4) + (cube(k)) / 4];

    //initialize edge array with total number of edges
    allEdges = new Edge[3 * cube(k) / 4];

    for (int i = 0; i < (cube(k) / 4); i++) {
        allNetworkElements[i] = new PhysicalMachine();
    }

    for (int i = (cube(k) / 4); i < (cube(k) / 4 + square(k) / 2); i++) {
        allNetworkElements[i] = new Switch("edge");
    }

    for (int i = (cube(k) / 4 + square(k) / 2); i < (cube(k) / 4 + square(k)); i++) {
        allNetworkElements[i] = new Switch("aggregation");
    }

    for (int i = (cube(k) / 4 + square(k)); i < allNetworkElements.length; i++) {
        allNetworkElements[i] = new Switch("core");
    }

    printParameters();

}
```

```
public static void initializeVirtualMachines() {
    int numberOfPhysicalMachines = (cube(k)) / 4;
    int whichPhysicalMachine;
    PhysicalMachine element;

    //initalizing the vm pairs to random PMs
    for (int i = 0; i < l; i++) {

        //first of the pair
        whichPhysicalMachine = (int) (Math.random() * numberOfPhysicalMachines);
        element = (PhysicalMachine) allNetworkElements[whichPhysicalMachine];
        element.addVm(i);

        whichPhysicalMachine = (int) (Math.random() * numberOfPhysicalMachines);
        element = (PhysicalMachine) allNetworkElements[whichPhysicalMachine];
        element.addVm(i);

    }

    //intialize bandwidths with number of vm pairs
    vmBandwidth = new double[l];

    //setting the bandwidth for each pair
    for (int i = 0; i < vmBandwidth.length; i++) {
        double bandwidth = bMin + Math.random() * (bMax - bMin);
        vmBandwidth[i] = bandwidth;
    }

}
```

PATHFINDING CODE

The findPath functions input is a virtual machine pair and the output is the path that connects that virtual machine pair through the network. The behavior for this path finding function is described in detail in the algorithm section, but it an implementation of the greedy pathfinding algorithm.

```
public static ArrayList<Integer> findPath(int input) {
    // l is the number of vm pairs
    //find the path from vm0 to vm0
    int[] locations = findVmLocation(input);
    int hops = hopsPerPair(input);

    ArrayList<Integer> path = new ArrayList<>();
    ArrayList<Integer> on = new ArrayList<>();

    //for 0 hops
    if (hops == 0) {
        path.add(locations[0]);
        return path;
    }

    //handling the case for when there is only one hop
    if (hops == 1) {
        on.add(locations[0]);
        path.add(locations[0]);

        for (Edge allEdge : allEdges) {
            if (allEdge.getNode1() == on.get(0)) {
                on.add(allEdge.getNode2());
                //adding bandwidth contribution
                if (!allEdge.addBandwidth(vmBandwidth[input])) {
                    return null;
                }

                path.add(allEdge.getNode2());
                on.remove(0);
                break;
            }
        }
```

```
        for (Edge e : allEdges) {
            if (e.getNode2() == on.get(0)) {

                path.add(locations[l]);
                break;
            }
        }

        for (Edge e : allEdges) {
            if (e.getNode1() == locations[l]) {
                //adding bw contribution for return edge
                if (!e.addBandwidth(vmBandwidth[input])) {
                    return null;
                }
                break;
            }

        }

        return path;
    }
    //handling the 3 hop case

    if (hops == 3) {
        on.add(locations[0]);
        path.add(locations[0]);

        //finding the first switch
        for (Edge e : allEdges) {
            if (e.getNode1() == on.get(0)) {
                on.add(e.getNode2());
                path.add(e.getNode2());
                on.remove(0);
```

```
            //adding bandwidth contribution
            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }
            break;
        }
    }
    //finding the second switches
    for (Edge e : allEdges) {
        if (e.getNode1() == on.get(0)) {
            on.add(e.getNode2());
        }
    }
    int previousSwitch = on.remove(0);
    int currentLowestBwSwitch = findLowestBandwidthSwitchChoice(on, previousSwitch, input);

    //adding bandwidth for 2-3 switch
    for (Edge e : allEdges) {
        if (e.getNode1() == previousSwitch && e.getNode2() == currentLowestBwSwitch) {
            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }
        }
    }

    //take the switch with the lowest bandwidth route
    on.clear();
    on.add(currentLowestBwSwitch);
    path.add(currentLowestBwSwitch);

    //find the path to the switch that connects to the other VM in the pair
    int finalSwitch = -1;
    for (Edge e : allEdges) {
        if (e.getNode1() == locations[1]) {
            finalSwitch = e.getNode2();
        }
    }
```

```
    //now connect 2nd to last switch with last switch
    for (Edge e : allEdges) {
        if (e.getNode2() == on.get(0) && e.getNode1() == finalSwitch) {
            path.add(e.getNode1());

            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }
        }
    }
    //add location of final PM
    //add final path bandwidth
    for (Edge e : allEdges) {
        if (e.getNode1() == locations[1]) {
            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }
        }

    }
    path.add(locations[1]);

    return path;
}

//handling 5 hop case
if (hops == 5) {
    on.add(locations[0]);
    path.add(locations[0]);

    //getting first switch
    for (Edge allEdge : allEdges) {
        if (allEdge.getNode1() == on.get(0)) {
            on.add(allEdge.getNode2());
            path.add(allEdge.getNode2());
            on.remove(0);
```

```
            if (!allEdge.addBandwidth(vmBandwidth[input])) {
                return null;
            }

            break;
        }
    }

    //getting second switch
    //finding the second switches
    for (Edge e : allEdges) {
        if (e.getNode1() == on.get(0)) {
            on.add(e.getNode2());
        }
    }
    int previousSwitch = on.remove(0);
    //find which switch has the lowest bandwidth to our previous switch

    int currentLowestBwSwitch = findLowestBandwidthSwitchChoice(on, previousSwitch, input);
    //couldn't find switch
    if (currentLowestBwSwitch == -1) {
        return null;
    }

    //adding bandwidth from 2-3
    for (Edge e : allEdges) {
        if (e.getNode1() == previousSwitch && e.getNode2() == currentLowestBwSwitch) {
            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }
        }
    }

    //take the switch with the lowest bandwidth route (Aggregate switch)
    on.clear();
    on.add(currentLowestBwSwitch);
    path.add(currentLowestBwSwitch);
```

```
//find the core switch with the lowest bandwidth
//first finding all the core switches our aggregate switch connects to
for (Edge e : allEdges) {
    if (e.getNode1() == on.get(0)) {
        on.add(e.getNode2());
    }
}

previousSwitch = on.remove(0);
//find our lowest bandwidth choice AS - CS
currentLowestBwSwitch = findLowestBandwidthSwitchChoice(on, previousSwitch, input); //its failing in here?

//couldn't find switch
if (currentLowestBwSwitch == -1) {
    return null;
}

//adding bandwidth from 3-4
for (Edge e : allEdges) {
    if (e.getNode1() == previousSwitch && e.getNode2() == currentLowestBwSwitch) {
        if (!e.addBandwidth(vmBandwidth[input])) {
            return null;
        }
    }
}

on.clear();
on.add(currentLowestBwSwitch);
path.add(currentLowestBwSwitch);

//find the lowest bandwidth choice from CS - appropriate AS
//which pod does it connect to?
int podNumber = ((4 * locations[1]) / square(k));

//is our cs in first half or second half?
int a = currentLowestBwSwitch;
```

```
int min = -1;
int max = -1;

if (a >= cube(k) / 4 + square(k) && a < cube(k) / 4 + 9 * square(k) / 8) {
    //first half
    //what range of pod numbers to consider?
    min = cube(k) / 4 + square(k) / 2 + (podNumber * (k / 2));
    max = cube(k) / 4 + square(k) / 2 + (podNumber * k / 2) + k / 4; //not inclusive
}

if (a >= cube(k) / 4 + 9 * square(k) / 8 && a < cube(k) / 4 + 5 * square(k) / 4) {
    //second half
    //what range of pod numbers to consider?
    min = cube(k) / 4 + square(k) / 2 + (podNumber * k / 2) + k / 4;
    max = cube(k) / 4 + square(k) / 2 + (podNumber * k / 2) + k / 2;// not inclusive
}

previousSwitch = on.remove(0);

on.clear();

//adding possible AS to consider
for (int i = min; i < max; i++) {
    on.add(i);
}

//lowest bandwidth choice from CS - AS
currentLowestBwSwitch = findLowestBandwidthSwitchChoiceDown(on, previousSwitch, input);

//couldn't find switch
if (currentLowestBwSwitch == -1) {
    return null;
}
```

```
    //adding bandwidth from 4-5
    for (Edge e : allEdges) {
        if (e.getNode2() == previousSwitch && e.getNode1() == currentLowestBwSwitch) {
            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }
        }
    }

    path.add(currentLowestBwSwitch);
    //to find choice from AS - ES go backwards from bottom
    //find the path to the switch that connects to the other VM in the pair
    int finalSwitch = -1;
    for (Edge e : allEdges) {
        if (e.getNode1() == locations[1]) {
            finalSwitch = e.getNode2();
        }
    }

    //now connect 2nd to last switch with last switch
    for (Edge e : allEdges) {
        if (e.getNode2() == on.get(0) && e.getNode1() == finalSwitch) {
            path.add(e.getNode1());
            if (!e.addBandwidth(vmBandwidth[input])) {
                return null;
            }

        }

    }

    path.add(locations[1]);

        //add bandwidth for last
        for (Edge e : allEdges) {
            if (e.getNode1() == locations[1]) {
                if (!e.addBandwidth(vmBandwidth[input])) {
                    return null;
                }
            }
        }

        return path;
    }

    return path;
}
```