TRAFFIC PRIORITY MAXIMIZATION IN POLICY ENABLED TREE BASED DATA CENTERS

A Thesis

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

Alexander Ing

Summer 2018

ACKNOWLEDGEMENTS

I would like to thank the department of computer science for sharing their knowledge and teaching me the necessary material to make this thesis. A special thanks to my committee chair, Dr. Bin Tang, for his passion and influence on me and this research. I am also grateful to my committee members, Dr. Han and Dr.Beheshti, for their input and guidance.

TABLE OF CONTENTS

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

ABSTRACT

Increased internet service usage places increased demand on data centers posing new problems such as degrading performance and increased security risks. Middleboxes such as load balancers and intrusion detection systems are one way of addressing the issues facing data centers. A policy chain is a sequence of middleboxes that traffic must traverse in a specified order, policy driven data centers enforce policy chains to insure data integrity and improve network performance. In situations where a data center is impacted and receives more traffic demand than available bandwidth then a decision must be made as to which requests to satisfy. This thesis examines the maximization of traffic priority in policy driven data centers using tree topologies, while adhering to bandwidth constraints. Three heuristic algorithms are proposed to address the maximization problem. A dynamic programming approach that outperforms the heuristic algorithms is proposed when the data center is under additional constraints.

CHAPTER 1

INTRODUCTION

The increase of internet activities and data processing has led to an increase of data center traffic and usage. This increase in activity has led to increased demand for Infrastructure as a Service (IaaS) and cloud data center services to provide the resources needed to run online services such as video on demand, social media applications, and data processing (Bhardwaj, 2010). The influx of activity has led to new issues within cloud data centers such as security, quality assurance, traffic control, and resource allocation (Buyya, 2010). Data centers use a variety of techniques and tools to address these issues, an example of such tools are network appliances called middleboxes. These middleboxes help improve network performance, security, and many other aspects of the data center. Middleboxes are computer networking devices that perform specialized tasks on data center traffic, examples of such tasks are packet inspection, firewalls, load balancing, and intrusion detection (Carpenter, 2002).

Network appliances such as middleboxes are highly used in enterprise environments as they provide much needed network services. Many data centers rely on middleboxes to perform crucial tasks such as ensuring their data is secure and to help in data forwarding. Due to the specialization and complexity of the tasks performed by middleboxes, they are oftentimes installed in data centers using specialized proprietary hardware. The use of proprietary hardware makes installing and managing middleboxes time intensive and costly.

The virtualization technologies were introduced to data centers in order to address the limitations and inflexibility of traditional network functions. Network function virtualization (NFV) and software defined networks (SDN) are two technologies that are being introduced into data centers to help optimize their processes (Han, 2015). SDNs and their controllers receive information from throughout the data center, allowing for a global view of the status of the network, unlike traditional off the shelf network components. The global view allows for SDN enabled data centers the opportunity to perform optimizations within the network such as improving traffic control, load balancing, and network policy enforcement (Markiewicz, 2014). The opportunity for optimizations provided by SDNs and NFV has increased interest and improved adoption rate of such technologies, it is estimated that 44% of traffic within data centers will be supported by SDNs and NFV technology by 2020 (Cisco 2016).

The introduction of IaaS and virtualization of hardware resources has led to the use of virtual machines (VMs) in data centers. The VMs provide customers with computing resources in an isolated environment that the customer can then manipulate to perform their desired task (Beloglazov, 2010). The VMs also provide a monetization strategy for the datacenter as the VMs and their resources can be rented to customers. VM communication within the data center make up the majority of traffic on the data center network, by 2020 it is estimated that communication within the data center itself will make up 77% of its network traffic (Cisco, 2016).

In a policy driven data center, traffic flowing through the data center will need to adhere to the network’s policy. These policies establish rules that network traffic must follow such as the order of middleboxes traffic must traverse. An example of such a policy would be for each VM communication pair generating traffic, that traffic would need to pass through a firewall for security then pass through a WAN optimizer. The traffic in the example would need to follow the policy and go through the specified middleboxes in their assigned order before getting delivered to its destination. The ordering of the middleboxes is called a policy chain or service chain, as each packet in the traffic flow must follow the chain of middleboxes in the policy (Sallam, 2018).

Data center topologies describe the structure of the network and how the network components are setup. The data center topology can determine how many paths are available to transport data flow from server to server. There are many data center topologies in use today such as Fat Tree, Bcube, and Elastic Trees each with their own advantages and disadvantages (Heller, 2010).

This paper will discuss how to maximize VM communication traffic priority within the data center network that are limited to using a tree topology while adhering to network policy. Data center traffic must be processed by every middlebox type in the specified order, as well as satisfying bandwidth constraints. We call the problem priority maximization problem. A literature review is conducted to show similar work already done in this field. The priority maximization problem is formally formulated and special cases are addressed. We show that in special cases the priority maximization problem can be modeled as the knapsack problem which can be solved using dynamic solving. A few heuristic algorithms are proposed they are lowest demand first, highest priority first, and highest average priority first. The heuristic algorithms are compared to the dynamic programming solution in the special cases to show the heuristics performance under specific restrictions. The heuristics are then run in the general case and compared with each other to determine their performance.

Background Survey

Many studies have investigated traffic optimization in data center networks such as Charikar et al. who have characterized a multicommodity flow problem in a general graph, with demand on flows and capacity constraints on edges (Charikar, 2018). They also introduce a new constraint where flow must be processed by compute nodes hosting middleboxes in a particular order determined by a policy applied to all flow. They attempt to optimize middlebox placement among the available compute nodes as well as optimize traffic steering, and routing paths. The proposed solution to their multicommodity flow problem is the use of linear programming which are used to solve the optimization problems.

Vazirani proposed an approximation algorithm for maximizing flow in a tree graph, he accomplished this by solving multi cut and integer multicommodity flow (Vazirani, 2003). They formulate a model with source and sink nodes at the leaves of the tree with weighted edges. Their proposed solution is a primal dual algorithm that results in a 2 approximation algorithm. This paper however does not consider middleboxes or policy in its formulation. Gouveia also solves the multicommodity flow problem in data center networks but adds a new constraint to flows (Gouveia, 1996). The new constraint is to limit the amount of hops a flow can take, this is similar to the acceptable amount of delay a flow can have to meet quality of service standards. The solution they propose is a linear programming approach that provides a lower bound to the amount of hop constraint. They solve the constraint in both directed and undirected graphs, but they do not take into consideration middleboxes or policy enforcement. Nguyen et al. propose an optimization framework called OFFICER that is a heuristic algorithm that attempts to maximize traffic in a network, keeping into consideration endpoint policy (Nguyen, 2015). The proposed algorithm performs in polynomial time, however does not take into consideration middlebox policy chains.

Policy enforcement in data centers has also been a hot topic in research due to the amount of optimizations and complexity of the problems that need addressing. Qazi et al. propose a SDN based traffic steering model to direct traffic in data centers such that network policy is enforced (Qazi, 2013). The model which they call SIMPLE, showed that SDNs can be used to not only monitor traffic but to steer it toward ordered middleboxes in the policy chain. Fayazbakhsh et al. also propose a solution to policy enforcement using what they call FlowTags (Fayazbakhsh, 2013). They recognize that network wide policy enforcement with middleboxes can be complex, and without proper implementation can be a cause for errors. Their solution is FlowTags, traffic that are processed by a middlebox are tagged upon completion. Joseph et al. proposed a policy aware switching layer that they called Player (Joseph, 2008). Player is a layer-2 switching layer that consist of policy aware switches called pswitches, middleboxes are then connected to the pswitches. These pswitches can then forward traffic to each other to enforce policy.

Of the works mentioned above, only Charikar et al. addresses policy chain enforcement and traffic optimization under constraints of edge capacity and traffic demand. However, their work differs from ours as they do not take into consideration communication priority and instead focus on flow maximization. They also focus heavily on middlebox placement within the compute nodes in the data center, while we assume the middleboxes are already installed and will not be moved after installation.

CHAPTER 2

PROBLEM DESCRIPTION AND METHODOLOGY

Data centers may not be able to serve all of the communication request, especially during peak hours of internet service usage. It is shown that data flow within the data center itself is a major component of all traffic serviced by its network (Mahimkar, 2011). When data centers are overloaded a decision must be made as to which VM requests to satisfy, and which VMs to move to a different network. VM communication pairs may have higher priority than others such as certain services such as VoIP or video streaming depending on the network configurations. The problem facing data centers is which virtual machine requests should be fulfilled to insure network congestion does not interfere with data flow and optimize the value of the VMs being serviced.

Network Model

The data center is modeled as an undirected graph G (V, E). Where $V = V_P \cup V_S$, the union of physical machines V_P and the network switches V_S. E is the set of edges in the graph G, each edge representing a connection in the network, connecting a switch to another switch or switch to physical machine. Each network component V is connected by C connections, where $1 \leq C$. The data center network will contain L communication pairs; each communication pair consist of a VM pair. The set of communicating VM pairs $P = \{(vm_1, vm_1`), (vm_2, vm_2`), \ldots, (vm_i, vm_i`)\}$ where each pair $vm_i$ and $vm_i$ ` $(1 \leq i \leq L)$, are the communication source VM and a destination VM respectively. Each physical machine V_P can store multiple VMs, it is possible for the same physical machine to host both vm_i and vm_i`. Middleboxes M can be installed on switches V_S within the datacenter network.

VM Pair Model

Each VM pair P consist of a source and destination VM, the flow of traffic travels from the source VM to its corresponding destination VM. Each VM pair in our model has three properties: communication frequency, communication priority, and demand. Pair (V_i, V_i`) communicates at frequency $F_i$, where $F_i$ is a random number from [1, F]. The second VM pair property is priority $T_i$, each pair ($V_i$, $V_i$`) is given a priority which is a random number from [1, T]. The total value of a VM pair is given by the product of T_i and F_i. The last property associated with each VM pair (V_i, V_i`) is demand D_i, the demand is the amount of bandwidth used by the communicating pair. The demand for a given for a pair is a random number from [1, D].

Edge Model

Each edge E in graph G has a capacity to the available bandwidth, the bandwidth capacity determines how much demand an edge can support. Edge (u,v) has capacity K(u,v), indicating the bandwidth available at (u,v). The total demand D across an edge must not exceed capacity, thus $D \leq K$. Total demand supported by a given edge E is the sum of demand from all VM communication pairs transmitting over E. Each edge connects at most two network switches vertices V_S or one V_S to a physical machine vertex V_P.

Middlebox Model

The data center is modeled such that the set M middleboxes are placed within the network, where $M = \{m_1 \ldots m_x\}$, x being the total number of middleboxes placed in the data center. In this model $x \geq 2$ as policy enforcement is one of the factors in this study. However, it should be noted that the proposed heuristic algorithms will still perform should $M < 2$ be the case. The middleboxes are placed randomly on a switch V_S within the data center network, an example is shown in figure 1. The path needed to traverse the ordered middleboxes in the policy chain make up what we call the 'spine'. The spine S is $S \subseteq E$ where $S_i = E(u_q, v_{q+1})$, q and q+1 indicate the middlebox in the set M. The term spine is used as the VM communication pairs must traverse the predetermined path once the flow reaches the first middlebox in the policy chain. Figure 1 shows an example of the spine which is colored in orange. In the example below, the policy to be enforced is as follows {MB1, MB2, MB3} and the spine consist of the edges E(MB1, MB2) and E(MB2, MB3). The switch containing the first middlebox in the policy chain is labeled the ingress switch as it is the switch letting flow into the spine. Conversely, the switch hosting the last middlebox in the policy is labeled as the egress switch, as flow is leaving the spine through said switch.

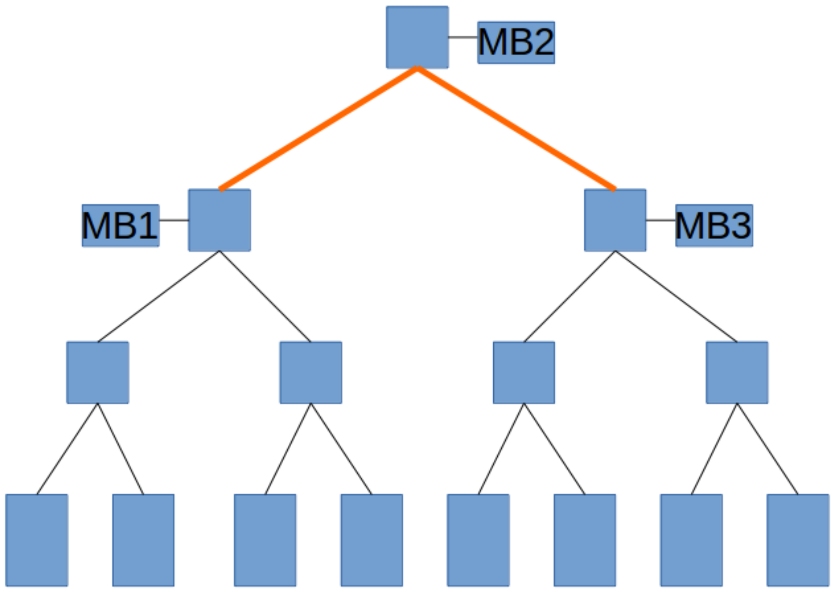


Figure 1. Example tree data center topology with highlighted spine path

Data Center Topology

The data center topology in this study is modeled as a tree, an example of such a topology is shown in figure 1. The leaf nodes of the tree represent physical machines with capabilities and resources to host VMs. All non-leaf nodes represent OpenFlow enabled switches with sufficient resources to install and support middleboxes. OpenFlow is a standardized protocol used by SDN controllers to interact with switchs using the protocol (McKeown, 2008). The tree data center topology was chosen because of the property of having a single path to and from any node. This makes routing flows from a VM in a communication pair to its destination simple, allowing us to focus on the VM satisfiability and communication priority maximization. Despite the lack of redundancy, trees are still used in data center topologies. One common case for tree use in data centers is when virtual lans (VLANs) are implemented, they are often structured as spanning trees of the network. An example of this is Cisco's suggestion of implementing Remote Span (RSPAN) VLANs, citing that the redundancy for such usage is unnecessary (Cisco, 2005). Even a physical 3-layer network can have a logical 2-layer topology forming a tree that can be created using a spanning tree algorithm of the network and the use of VLANs (Meng, 2010).

CHAPTER 3

THE PROPOSED SOLUTIONS

This section of the thesis will discuss the proposed algorithms to maximize communication priority given edge capacity and communication demand constraints. First a feasibility study is proposed to determine if the pending communication can all be satisfied by network. Next, four algorithms are proposed and their functionality are shown and explained.

Feasibility Study

Before optimizations algorithms are considered, we must first determine if they are even necessary. The feasibility study is used to check if the data center has the resources necessary to satisfy all of the VM communication pairs. If there is available bandwidth to sustain all of the communication flows, then there is no need to decide which VMs to choose in order to optimize priority as we can simply choose all of the VMs. The tree topology makes the feasibility study simple as there is only one path to consider for each pair, thus to determine feasibility we can check the path of each allocated communication pair and calculate remaining bandwidth available. Every VM pair will subtract its demand T from the edge capacity K of edges along their path to the spine, through the spine, and to the destination VM. while checking every communication, if any edge no longer has any bandwidth, $K < 0$, then the feasibility study shows that not every VM can be satisfied and a decision must be made to which VMs to choose in order to maximize priority.

Algorithms

Lowest Demand First

The Lowest Demand First algorithm performs the following tasks. First the VM communication pairs P are ordered in terms of lowest demand D. VM IDs are stored using a list in non-decreasing order, the IDs will be used to keep track of which VM to check. Once the list is created, the first VM in the list, will identify all the edges on its path toward the ingress switch. The selected edges will then be check to determine their bandwidth capacity K, if $K < D$ then the VM is not considered further and the Lowest Demand First will move to the next VM. If $K \geq D$ then the most traveled edge in the spine is checked to see its bandwidth availability, if not then the VM is not considered. If there is available bandwidth, then the edges from the egress switch of the spine to the destination VM is selected and the bandwidth is checked, if not enough bandwidth then the VM is not considered. If there is available bandwidth, then the whole path from source and destination of the communication pairs can support the demand of the pair. When the demand has been shown to have available bandwidth to support the VM, it will send the communication over the checked path. The time complexity for sorting the VMs would be $O(V^2)$ and each VM would need a tree traversal to check bandwidth along its path which is $O(n)$, resulting in a time complexity of $O(V^2 * n)$. Figure 2 below shows the pseudocode of the Lowest Demand First algorithm.

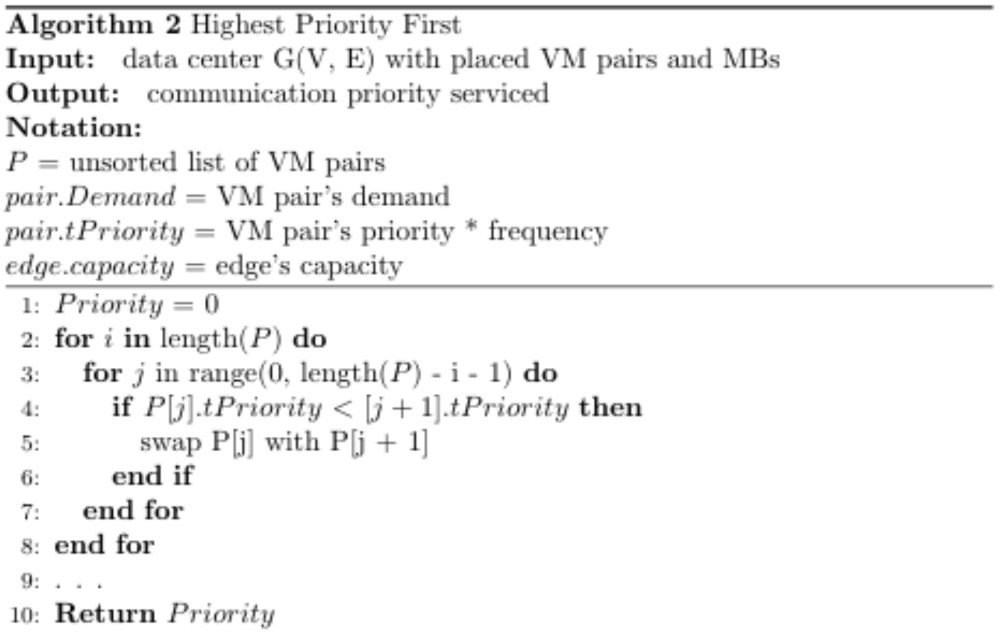
Algorithm 1 Lowest Demand First
Input: data center G(V, E) with placed VM pairs and MBs
Output: communication priority serviced
Notation:
P = unsorted list of VM pairs
$pair.Demand$ = VM pair's demand
$pair.tPriority$ = VM pair's priority * frequency
$edge.capacity$ = edge's capacity

```
Priority = 0
for i in length(P) do
    for j in range(0, length(P) - i - 1) do
        if $P[j].Demand > P[j+1].Demand$ then
            swap P[j] with P[j + 1]
        end if
    end for
end for
for pair in P do
    $vm1, vm2 \leftarrow$ VMs in pair
    $fullFlag = 0$
    $ES \leftarrow$ edge set from $vm1$ to $vm2$ following policy
    for edge in ES do
        if $pair.Demand > edge.capacity$ then
            $fullFlag = 1$
            break
        end if
    end for
    if $fullFlag == 0$ then
        establish pair connection
        update edges capacity $ES$
        $Priority \mathrel{+}= pair.tPriority$
    end if
end for
Return Priority
```

Figure 2. Lowest demand first pseudocode

Highest Priority First

The Highest Priority First algorithm receives a list of VM communication pairs. The communication pairs are sorted by the highest priority, a list of VM ids are stored to keep track of order. The algorithm then performs the same bandwidth check described in the Lowest Demand First algorithm, checking edges to the ingress switch, the spine, then from the egress switch to the destination VM. If all the edges on the communication path have available bandwidth, then the checked VM communication is permitted. If there is not enough bandwidth, then the VM is not considered. This algorithm shares the same structure of sorting then tree traversal as the lowest demand first algorithm, thus the time complexity is O(V^2 * n). Figure 3 below shows the pseudocode of Highest Priority First algorithm, from line nine onwards refer to the first algorithm as they are identical.



Algorithm 2 Highest Priority First
Input: data center G(V, E) with placed VM pairs and MBs
Output: communication priority serviced
Notation:
P = unsorted list of VM pairs
$pair.Demand$ = VM pair's demand
$pair.tPriority$ = VM pair's priority * frequency
$edge.capacity$ = edge's capacity

```
1: $Priority = 0$
2: for $i$ in length($P$) do
3:    for $j$ in range(0, length($P$) - i - 1) do
4:       if $P[j].tPriority < [j+1].tPriority$ then
5:          swap P[j] with P[j + 1]
6:       end if
7:    end for
8: end for
9: . . .
10: Return $Priority$
```

Figure 3. Highest priority first pseudocode

Highest Average Priority First

The Highest Average Priority First algorithm receives a list of VM communication pairs. The VM pairs each have the attributes of demand D and priority T, the average priority is calculated by T / D. Once the average priority is calculated the VMs are sorted in order of highest average priority first. The checking and allocating of bandwidth used after the ordering is established is the same as the previous algorithms.

This algorithm shares the same structure as the previous algorithms, resulting in the same time complexity of O(V^2 * n). Figure 4 below shows the pseudocode of the Highest Average Priority First algorithm, from line nine onwards refer to the first algorithm as they are identical.

```
Algorithm 3 Highest Average Priority First
Input:  data center G(V, E) with placed VM pairs and MBs
Output:  communication priority serviced
Notation:
$P$ = unsorted list of VM pairs
$pair.Demand$ = VM pair's demand
$pair.tPriority$ = VM pair's priority * frequency
$edge.capacity$ = edge's capacity
 1: $Priority = 0$
 2: for $i$ in length($P$) do
 3:   for $j$ in range(0, length($P$) - i - 1) do
 4:     if $P[j].tPriority$ / $P[j].Demand$ < $P[j+1].tPriority$ / $P[j+1].Demand$ then
 5:       swap P[j] with P[j + 1]
 6:     end if
 7:   end for
 8: end for
 9: . . .
10: Return $Priority$
```

Figure 4. Highest average priority first pseudocode

Dynamic Programming

The tree data center with edges E that have uniform bandwidth capacity and VMs are placed under special conditions can be modeled such that dynamic programming can be used to select VM communication pairs. The special conditions for the VM communication pairs is the source VM in in the pairs must be hosted in a physical machine that exist in the subtree G`, where the root of G` is the ingress switch to the spine. An example of the subtree G` can be seen in figure 5, in this example MB1 is the first middlebox in the policy so the subtree G` is represented in red. Another special condition that must be met to allow for dynamic programming is the destination VMs in the communication pairs must exist in the subtree G``, where the root of G`` is the egress switch to the spine. The example shown in figure 5 has MB3 as the last middlebox in the policy, the subtree with the egress switch as the root is shown by the color green.

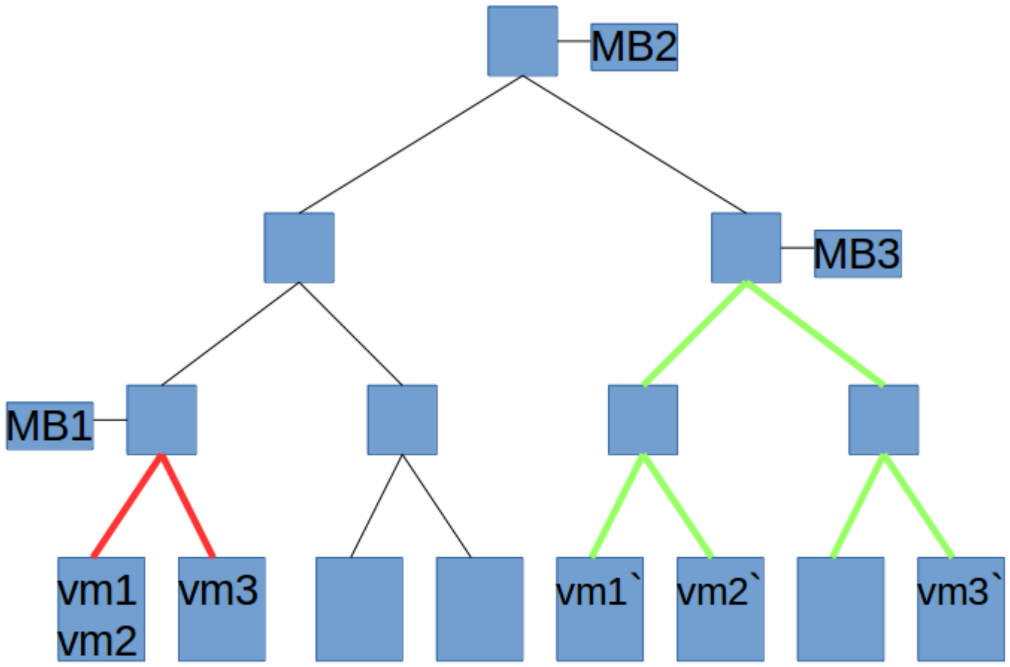


Figure 5. Tree topology with highlighted subtree under ingress and egress switch

If the source VMs are located in the subtree with the spine ingress as the root and the destination VMs located in the subtree with the spine egress as the root, then the maximization of priority in the data center can be modeled as the 1/0 knapsack problem. The knapsack problem is an optimization problem where a knapsack with total available capacity C is given and we are given [1 . . . n] items i, each item has the two attributes of weight W and value V (Khuri, 1994). Assuming that $\sum_{i=1}^{n} W_i > C$ and that items cannot be split resulting in an all or nothing approach per item selected, then a decision must be made for which items to take in order to maximize the value. The data center can be modeled to reflect the description above, the total available capacity in the model is calculated by the identifying the most traversed edge in the spine. Once identified, the edge capacity divided by the amount of times traversed will result in the total available capacity that this network can support. The items to choose from in this model are the communication pairs, the weight of each pair is modeled as the demand they require to send their flow, the value of each pair is modeled as the communication priority. The time complexity of dynamic programming for the knapsack problem is O(V * C) where V is the VMs and C is the maximum capacity the network can support. A tree traversal is required to determine the max capacity C, which is O(n), resulting in a total time complexity of O(V * C + n).

Dynamic programming example

Given the data center shown in figure 5 shown above, assume the following is true for this data center:

1. Edges have K = 2.
2. VM pair 1 and 2 have D = 1 and T * F = 1.
3. VM pair 3 has D = 2 and T * F = 3.

With the given information we can model the data center to resemble the 1/0 knapsack problem. In the example given all the edges in the spine are traversed once, resulting in the first edge being selected. Since all edges in this data center have K = 2, the total available capacity C for the knapsack problem would be K/traversals which is 2. VM pair 1 and 2 both are modeled to have weight 1 and value 1, while VM pair 3 has weight 2 and value 3. Table 1 below shows the results of the dynamic programming using the parameters described.

Table 1

Dynamic Programming Table Creation

Items \ Weight	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2
3	0	1	3

Using the generated table, the dynamic programming algorithm can determine that item three will provide the most value for the weight available, thus selecting VM pair 3. When VM pair 3 is selected, using the table we can see that the edge capacity no longer has any available capacity, so no other VM pair is selected. After VM pair 3 is selected the connection will be made and the edge capacities within the data center would be updated. In the provided example we show that the Dynamic Programming approach can be used to solve the priority maximization problem effectively.

CHAPTER 4

RESULTS AND ANALYSIS

This section will first discuss the performance of the heuristic algorithms, Highest Priority First, Lowest Demand First, Highest Average Priority First, dynamic programming approach, and random VM selection. First the case were VMs are placed in the subtree of the egress and ingress MBs to the spine is analyzed. Then the results of the heuristic algorithms will be analyzed in the general VM placement case.

The parameters used for VM attributes in the simulations are the same across all tests. Communication pair priority is randomly assigned upon creation; the range of priority is 1 - 100. The frequency of each communication pair is random, the range for communication frequency is 1 - 10. The demand cost of a communication pair is also random and assigned on creation, the range for demand is also 1 - 10. The topology used in the simulation is a tree data center with 84 nodes, each node having four children, excluding leaf nodes. This setup results in 21 switches in the data center connecting 64 physical machines as shown by figure 6. We will now explore the effects of varying parameters for the amount of middleboxes, available link capacity, and amount of VMs to examine the performance of the proposed algorithms.

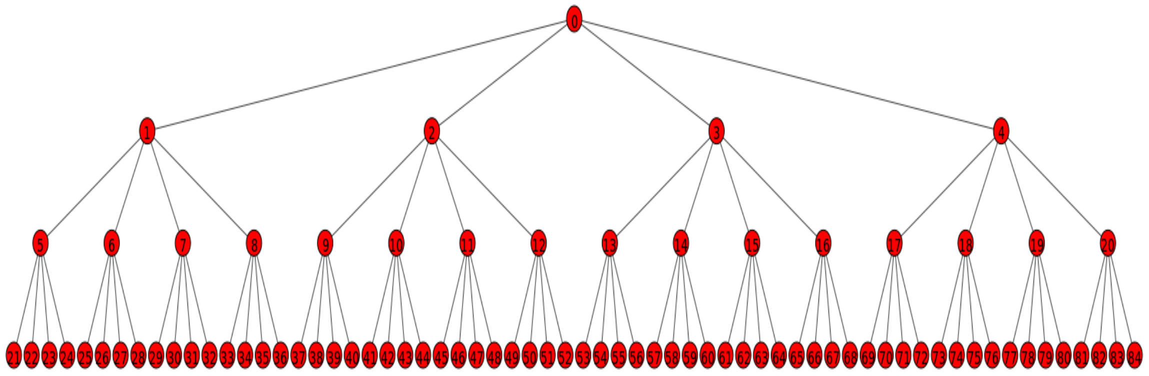


Figure 6. Data center topology used in simulations

First the heuristic algorithms and the dynamic programming algorithm are tested to see how they perform with varying amounts of middleboxes in the data center policy under the special conditions mentioned earlier. The parameters used for the simulation is link capacity equaling to 200. The amount of communication pairs attempting to establish data flow is 200, for a total of 200 source VMs and 200 destination VMs. Figure 7 shows the performance of the proposed algorithms when the network has three, five, and eight middleboxes in the policy. In all three middlebox cases we observe that the Dynamic Programming approach performs the best with an average of 3247.82 priority serviced when three MBs where placed in the data center. The worst performing heuristic algorithm was the Highest Priority First with an average of 1852.98 priority serviced with three MBs. We believe a potential reason for this underperformance is the algorithm does not take into consideration the demand of the communications, while the other higher performing algorithms, all consider demand when choosing communication pairs.

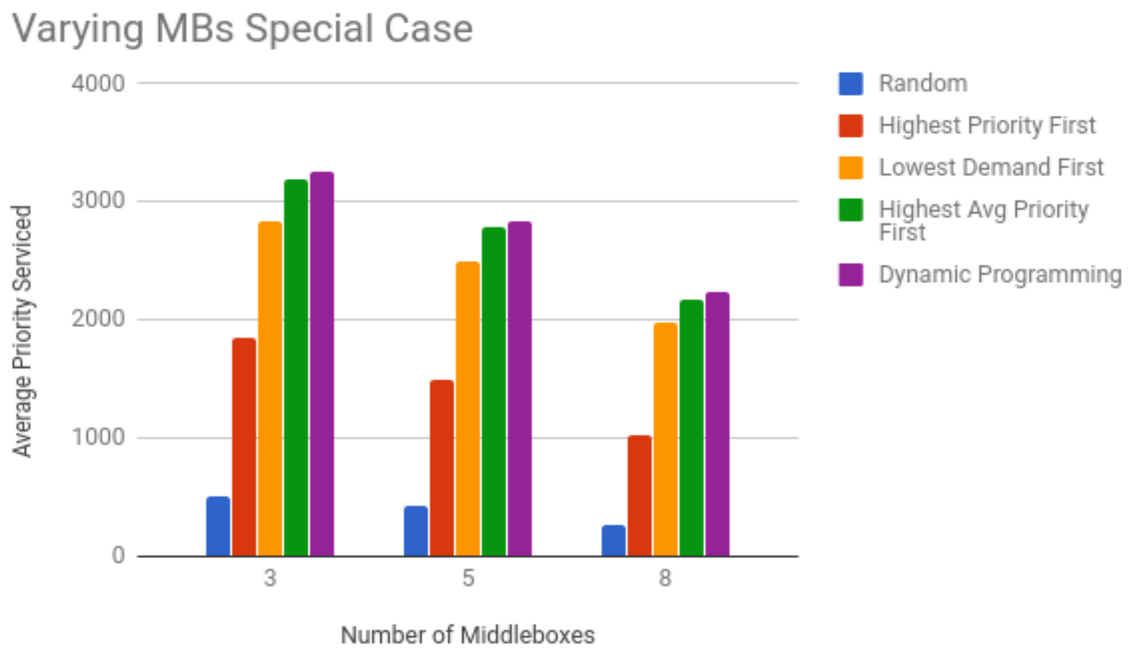


Figure 7. Algorithm performance with varying MBs, special case

Next the heuristic algorithms are tested to observe their performance with varying amounts of middleboxes in the data center policy in the general case. The parameters used for the simulation are the same as when the varying middleboxes were tested in the special case, once again testing three, five, and eight middleboxes. The difference now is the VM communication pairs are now be placed randomly on any physical machine in the data center instead of limiting to subtrees of the ingress and egress switches. Figure 8 shows the average priority from the results of the simulations, we observe that the highest performing heuristic algorithm is the Highest Average Priority First. The lowest performing heuristic, once again being the Highest Priority First.

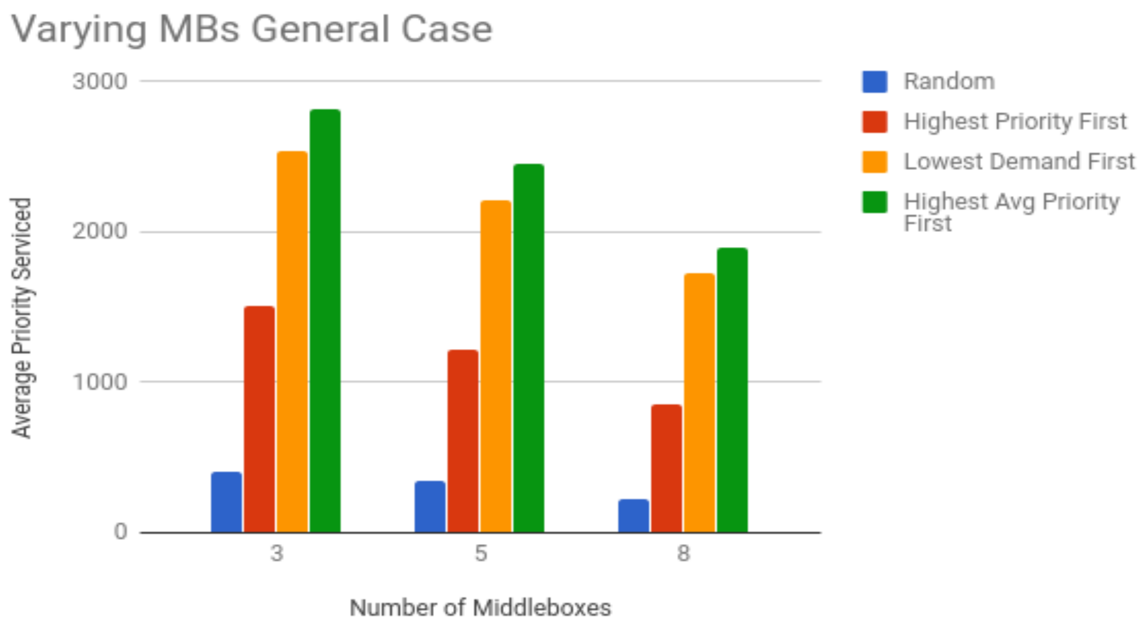


Figure 8. Algorithm performance with varying MBs, general case

Next we test what effect varying link capacity has on the proposed algorithms, fist we test the special case allowing for the Dynamic Programming algorithm to be used. The parameters used for the simulation is the number of middleboxes that are placed in the data center amounts to five. The amount of communication pairs attempting to establish data flow is 200, for a total of 200 source VMs and 200 destination VMs. The varying link capacities, which will determine how much demand they can sustain, used in the test is 100, 300, and 500. The results show the Dynamic Programming algorithm performs the best in all link capacities. The algorithm that performed the second best is the Highest Average Priority First. We can see from the results shown in Figure 9 that as the link capacity increases the more demand can be satisfied, and the better all of the algorithms perform.

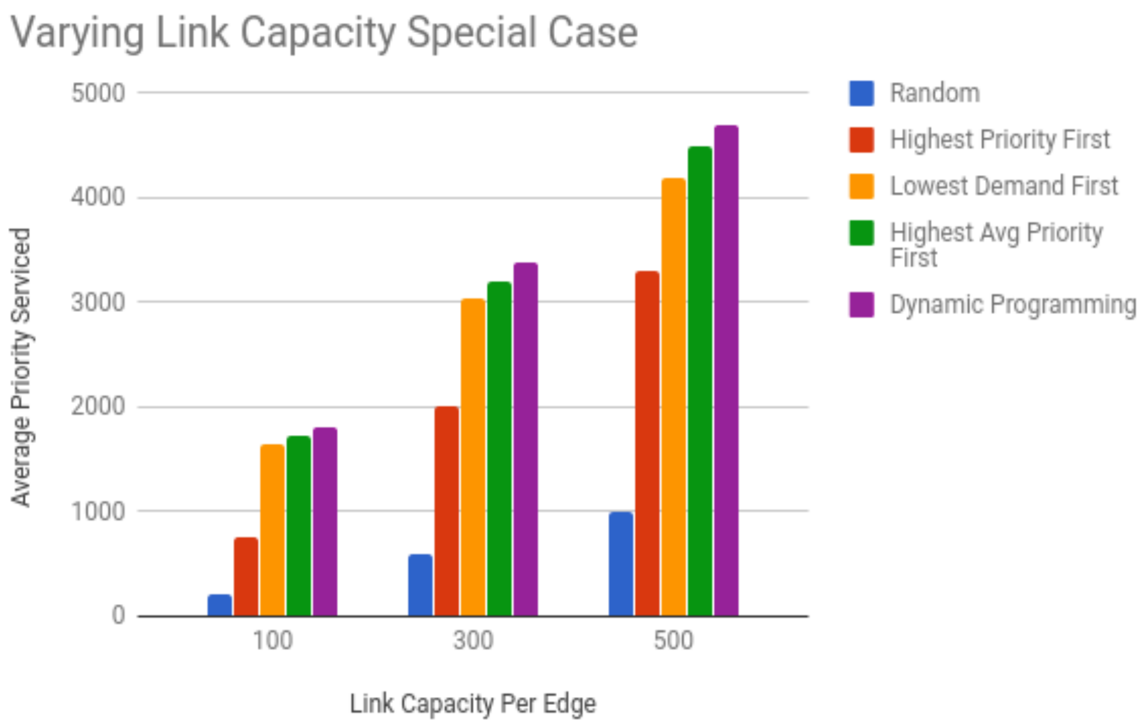


Figure 9. Algorithm performance, varying link capacity, special case

Next we test the heuristic algorithm's performance with varying link capacities in the general case. The parameters of the simulation testing were the same as the special case, but with random placement of the VMs from the communication pairs. The Highest Average Priority First performs the best out of the heuristic algorithms, with Highest Priority First only performing better than Random. The results shown in Figure 10 show that as the amount of capacity per edge increased the performance of Highest Average Priority First algorithm also increased.

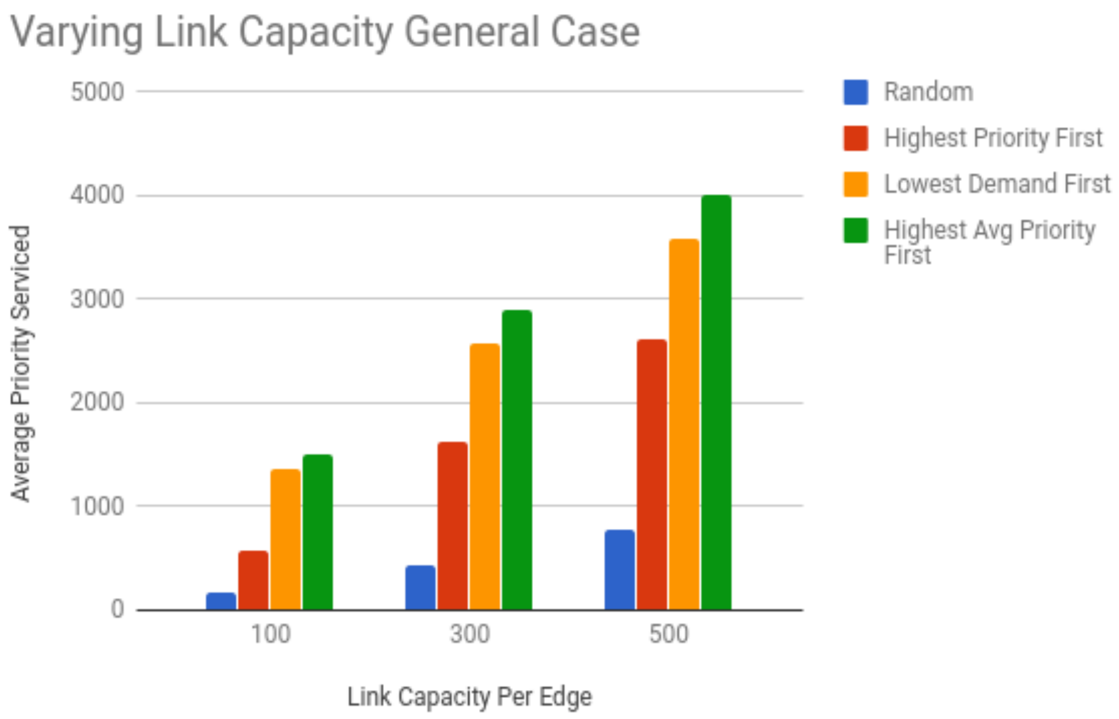


Figure 10. Algorithm performance, varying link capacity, general case

The last group of testing done on the algorithms is with varying amounts of VM communication pairs placed within the data center, first the special case is examined along with the Dynamic Programming approach. The parameters used for the simulation is the number of middleboxes that are placed in the data center amounts to five. The amount of link capacity for every edge is 200. The varying amount of VM communication pairs attempting to establish connections for data flow were tested using 100, 300, and 500 VM pairs. Once again the Dynamic Programming approach outperformed the other algorithms as shown in Figure 11, with the Highest Average Priority First performing second best. An interesting note is the average priority improved for all of the algorithms, with the exception of random, as the number of VM pairs increased as shown by table 2, despite the amount of bandwidth available was static at 200. A possible explanation for this occurrence is that as the amount of VM communication pairs increase, the algorithms now have a larger pool of communication pairs with preferable demand, frequency, and priority ratios.

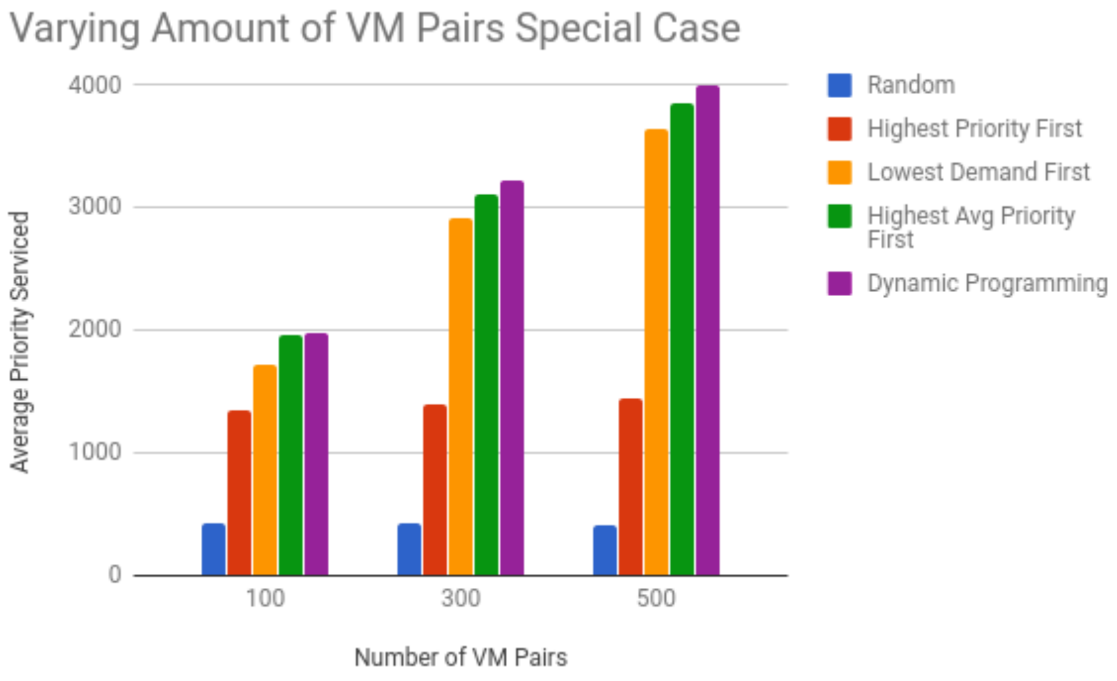


Figure 11. Algorithm performance, varying amount of VMs, special case

Table 2

Average Priority Varying Amount of VMs

Algorithms / Number of VMs	100	300	500
Random	420	421.2	411.82
Lowest Demand First	1725.1	2910.58	3649.14
Highest Avg Priority First	1963.08	3101.92	3853.14
Highest Demand First	1346.5	1400.76	1439.68
Dynamic Programming	1970.74	3224.24	3994.22

Lastly, the heuristic algorithms are compared with each other in the general case with varying amounts of VM communication pairs. The parameters of the simulation testing were the same as the special case, but with random placement of the VMs from the communication pairs instead of only in the subtree of the ingress and egress switches. The Highest Average Priority First algorithm performs the best of the heuristics with Lowest Demand First performing second best. Once again we observe from Figure 12 that as the amount of VM pairs increases so does the average priority satisfied, excluding random. Another interesting note is Highest Priority First algorithm does increase in performance with the amount of VM pairs available, but not as much as the other heuristic algorithms.

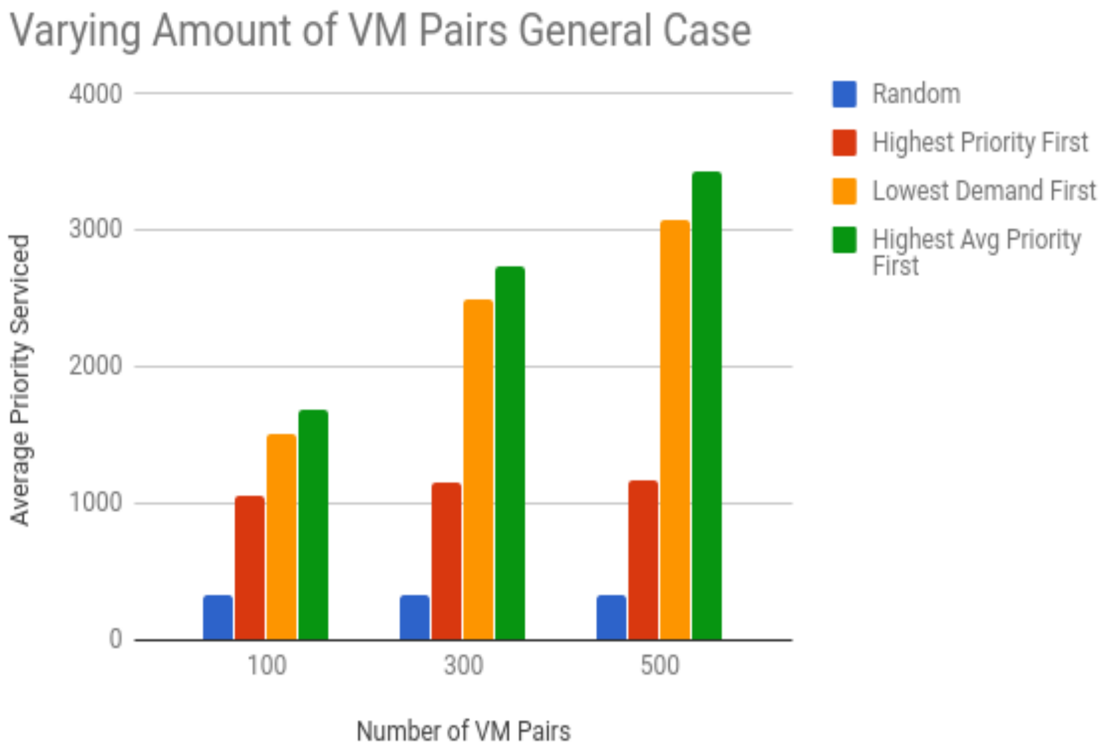


Figure 12. Algorithm performance, varying amount of VMs, general case

CHAPTER 5

CONCLUSION AND FUTURE WORK

Future work that could extend the research presented in this study would include development and testing of a general solution for the priority maximization in a policy enforced tree data center. This thesis looked into multiple middleboxes with only a single instance each, further research can be done to propose a solution to the multiple middlebox with multiple instances policy enforcement problem. The addition of other NFV technologies and techniques, such as VM replication or middlebox placement, can be factored into the optimization of the data center in tandem with the algorithms proposed in this paper. Further future work includes testing the proposed algorithms in an emulated data center environment, the testing was done via simulation in this report, but implementation via emulation would improve the validity of the findings.

In this thesis we proposed four algorithms to maximize communication priority of VM communication pairs in policy aware data centers using tree topology under bandwidth capacity constraints. To address the problem, three heuristic algorithms were proposed they are highest priority first, lowest demand, and highest average priority. Under specific conditions we show the priority maximization problem can be modeled as a 1/0 knapsack problem, which can then be solved using dynamic programming. Rigorous simulation showed that the dynamic programming out performs the three heuristic algorithms under the specific conditions, with the Highest Average Priority First performing second best. In the general case with VMs randomly distributed across the physical machines, the Highest Average Priority First performed the best with Lowest Demand First performing second best and all heuristics performing better then random choice. The proposed algorithms show that there are still many optimizations that can be made within data centers that can improve performance and quality of service without the need for purchasing new hardware.

REFERENCES

REFERENCES

Beloglazov, A., & Buyya, R. (2010, May). Energy efficient allocation of virtual machines in cloud data centers. In Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on (pp. 577-578). IEEE.

Bhardwaj, S., Jain, L., & Jain, S. (2010). Cloud computing: A study of infrastructure as a service (IAAS). International Journal of engineering and information Technology, 2(1), 60-63.

Buyya, R., Beloglazov, A., & Abawajy, J. (2010). Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges. arXiv preprint arXiv:1006.0308.

Carpenter, B., & Brim, S. (2002). Middleboxes: Taxonomy and issues (No. RFC 3234).

Charikar, M., Naamad, Y., Rexford, J., & Zou, X. K. (2018). Multi-commodity flow with in-network processing. arXiv preprint arXiv:1802.09118.

Cisco, C. (2005). Server Farm Security in the Business Ready Data Center Architecture v2. (pp. 2.1 - 2.16)

Cisco Index, C. G. C. (2016). Forecast and methodology, 2015-2020 white paper. Retrieved 1st June.

Fayazbakhsh, S. K., Sekar, V., Yu, M., & Mogul, J. C. (2013, August). Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (pp. 19-24). ACM.

Gouveia, L. (1996). Multicommodity flow models for spanning trees with hop constraints. European Journal of Operational Research, 95(1), 178-190.

Han, B., Gopalakrishnan, V., Ji, L., & Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. IEEE Communications Magazine, 53(2), 90-97.

Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S., & McKeown, N. (2010, April). Elastictree: Saving energy in data center networks. In Nsdi (Vol. 10, pp. 249-264).

Joseph, D. A., Tavakoli, A., & Stoica, I. (2008, August). A policy-aware switching layer for data centers. In ACM SIGCOMM Computer Communication Review (Vol. 38, No. 4, pp. 51-62). ACM.

Khuri, S., Bäck, T., & Heitkötter, J. (1994, April). The zero/one multiple knapsack problem and genetic algorithms. In Proceedings of the 1994 ACM symposium on Applied computing (pp. 188-193). ACM.

Mahimkar, A., Chiu, A., Doverspike, R., Feuer, M. D., Magill, P., Mavrogiorgis, E., ... & Yates, J. (2011, November). Bandwidth on demand for inter-data center communication. In Proceedings of the 10th ACM Workshop on Hot Topics in Networks (p. 24). ACM.

Markiewicz, A., Tran, P. N., & Timm-Giel, A. (2014, October). Energy consumption optimization for software defined networks considering dynamic traffic. In Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on (pp. 155-160). IEEE.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

Meng, X., Pappas, V., & Zhang, L. (2010, March). Improving the scalability of data center networks with traffic-aware virtual machine placement. In INFOCOM, 2010 Proceedings IEEE (pp. 1-9). IEEE.

Nguyen, X. N., Saucez, D., Barakat, C., & Turletti, T. (2015, April). OFFICER: A general optimization framework for OpenFlow rule allocation and endpoint policy enforcement. In Computer Communications (INFOCOM), 2015 IEEE Conference on (pp. 478-486). IEEE.

Qazi, Z. A., Tu, C. C., Chiang, L., Miao, R., Sekar, V., & Yu, M. (2013, August). SIMPLE-fying middlebox policy enforcement using SDN. In ACM SIGCOMM computer communication review (Vol. 43, No. 4, pp. 27-38). ACM.

Sallam, G., Gupta, G. R., Li, B., & Ji, B. (2018). Shortest Path and Maximum Flow Problems Under Service Function Chaining Constraints. arXiv preprint arXiv:1801.05795.

Vazirani, V. V. (2003). Multicut and Integer Multicommodity Flow in Trees. In Approximation Algorithms (pp. 145-153). Springer, Berlin, Heidelberg.

APPENDICES

APPENDIX A

LOWEST DEMAND FIRST CODE

```
class LowCostFirst:
    def __init__(self, topo, vmPairs):
        self.topo = topo
        self.vmPairs = vmPairs
        self.middleboxes = topo.get_middleboxes()
        self.cap_flag = 0
        self.dropped_pairs = 0

    def allocate(self):
        print("Lowest Freq first start: -running-")
        # after cost is calc'd for each pair, order: lowest cost first
        temp_arr = self.vmPairs
        for index in range(1, len(self.vmPairs)):
            value = self.vmPairs[index]
            i = index - 1
            while i >= 0:
                if value.comm_frequency() < self.vmPairs[i].comm_frequency():
                    temp_arr[i + 1] = self.vmPairs[i]
                    temp_arr[i] = value
                    i -= 1
                else:
                    break
        self.vmPairs = temp_arr


    def cap_check(self, pair):
        mb_counter = 0
        path = []

        vm1, vm2 = pair.get_vms()
        if len(self.middleboxes) != 0:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name()
            ))
        else:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
                vm2.get_parent().get_name()
            ))

        if len(self.middleboxes) > 1:
            mb_counter = mb_counter + 1  # start at 1
            while mb_counter < len(self.middleboxes):
                path.extend(self.topo.get_path(
                    self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                    self.middleboxes[mb_counter].get_parent_switch().get_name()
                )[1:])
                mb_counter = mb_counter + 1
            path.extend(self.topo.get_path(
                self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                vm2.get_parent().get_name()
            )[1:])
        master_edge_seq = self.topo.es
        path_edges = []
        count = 0
        for l in path:
            for e in master_edge_seq:
                if count < len(path) - 1:
                    if (str(e.tuple[0]) == str(l) and str(e.tuple[1]) == str(path[count + 1]))
    \
                        or (str(e.tuple[0]) == str(path[count + 1]) and str(e.tuple[1]) ==
    str(l)):
                      path_edges.append(e)
                      count = count + 1
        for single in path_edges:
            checker = single['capacity'] - pair.comm_frequency()
```

```
                single['capacity'] = single['capacity'] - pair.comm_frequency()
                if checker < 0:
                    self.cap_flag = 1
            for single_revert in path_edges:
                single_revert['capacity'] = single_revert['capacity'] +
pair.comm_frequency()
            # used to print the path, used for debug
            # print("path: " + str(path))

        def run_alg(self):
            pair_cost = 0
            priority = 0
            # once pairs are ordered, calc cost
            for pair in self.vmPairs:
                self.cap_check(pair)
                vm1, vm2 = pair.get_vms()
                # print("pair " + str(vm1.get_name()) + " " + str(vm2.get_name()) + " pri:
"
                #       + str(pair.get_priority()) + " cost: " + str(pair.run_alg_calc())
                #       + " freq: " + str(pair.comm_frequency()))

                if self.cap_flag == 0:
```

```
            mb_counter = 0
            priority += pair.get_priority()
            if len(self.middleboxes) != 0:
                pair_cost += self.topo.get_distance_new(
                    vm1.get_parent().get_name(),
                    self.middleboxes[mb_counter].get_parent_switch().get_name(),
                    pair.comm_frequency()
                )
                mb_counter = mb_counter + 1
                while mb_counter < len(self.middleboxes):
                    pair_cost += self.topo.get_distance_new(
                        self.middleboxes[mb_counter -
1].get_parent_switch().get_name(),
                        self.middleboxes[mb_counter].get_parent_switch().get_name(),
                        pair.comm_frequency()
                    )
                    mb_counter = mb_counter + 1
                pair_cost += self.topo.get_distance_new(
                    self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                    vm2.get_parent().get_name(),
                    pair.comm_frequency()
                )
```

```
        else:
            pair_cost += self.topo.get_distance_new(
                vm1.get_parent().get_name(),
                vm2.get_parent().get_name(),
                pair.comm_frequency()
            )
        self.cap_flag = 0
    else:
        self.dropped_pairs = self.dropped_pairs + pair.comm_frequency()
        self.cap_flag = 0
# print("lowest freq first, dropped packets: " + str(self.dropped_pairs))
# print("lowest freq first, Value: " + str(priority))
return priority
```

APPENDIX B

HIGHEST PRIORITY FIRST CODE

```
class HiPriFirst:
    def __init__(self, topo, vmPairs):
        self.topo = topo
        self.vmPairs = vmPairs
        self.middleboxes = topo.get_middleboxes()
        self.dropped_pairs = 0
        self.cap_flag = 0

    def allocate(self):
        print("Highest priority first start: -running-")
        temp_arr = self.vmPairs

        for index in range(1, len(self.vmPairs)):
            value = self.vmPairs[index]
            i = index - 1
            while i >= 0:
                if value.get_priority() > self.vmPairs[i].get_priority():
                    temp_arr[i + 1] = self.vmPairs[i]
                    temp_arr[i] = value
                    i -= 1
                else:
                    break
        self.vmPairs = temp_arr

    def cap_check(self, pair):
        mb_counter = 0
        path = []

        vm1, vm2 = pair.get_vms()
        if len(self.middleboxes) != 0:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name()
            ))
        else:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
                vm2.get_parent().get_name()
            ))

        if len(self.middleboxes) > 1:
            mb_counter = mb_counter + 1  # start at 1
            while mb_counter < len(self.middleboxes):
```

```
        path.extend(self.topo.get_path(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name()
        )[1:])
        mb_counter = mb_counter + 1
    path.extend(self.topo.get_path(
        self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
        vm2.get_parent().get_name()
    )[1:])
master_edge_seq = self.topo.es
path_edges = []
count = 0
for l in path:
    for e in master_edge_seq:
        if count < len(path) - 1:
            if (str(e.tuple[0]) == str(l) and str(e.tuple[1]) == str(path[count + 1])) \
                    or (str(e.tuple[0]) == str(path[count + 1]) and str(e.tuple[1]) == str(l)):
                path_edges.append(e)
                count = count + 1
for single in path_edges:
    checker = single['capacity'] - pair.get_communication_frequency()
    single['capacity'] = single['capacity'] - pair.get_communication_frequency()
```

```
            if checker < 0:
                self.cap_flag = 1
        for single_revert in path_edges:
            single_revert['capacity'] = single_revert['capacity'] +
pair.get_communication_frequency()
        # used to print the path, used for debug
        # print("path: " + str(path))


    def run_alg(self):
        pair_cost = 0
        priority = 0
        # once pairs are ordered, calc cost


 for pair in self.vmPairs:
            self.cap_check(pair)
            vm1, vm2 = pair.get_vms()
            # print("pair " + str(vm1.get_name()) + " " + str(vm2.get_name()) + " pri: "
            #       + str(pair.get_priority()) + " cost: " + str(pair.run_alg_calc())
            #       + " freq: " + str(pair.get_communication_frequency()))


            if self.cap_flag == 0:
                mb_counter = 0
```

```
priority += pair.get_priority()
if len(self.middleboxes) != 0:
    pair_cost += self.topo.get_distance_new(
        vm1.get_parent().get_name(),
        self.middleboxes[mb_counter].get_parent_switch().get_name(),
        pair.get_communication_frequency()
    )
    mb_counter = mb_counter + 1
    while mb_counter < len(self.middleboxes):
        pair_cost += self.topo.get_distance_new(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name(),
            pair.get_communication_frequency()
        )
        mb_counter = mb_counter + 1
    pair_cost += self.topo.get_distance_new(
        self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
        vm2.get_parent().get_name(),
        pair.get_communication_frequency()
    )
else:
    pair_cost += self.topo.get_distance_new(
                    vm1.get_parent().get_name(),
                    vm2.get_parent().get_name(),
                    pair.get_communication_frequency()
                )
            self.cap_flag = 0
        else:
            self.dropped_pairs = self.dropped_pairs +

pair.get_communication_frequency()
            self.cap_flag = 0
    # print("highest pri first, dropped packets: " + str(self.dropped_pairs))
    # print("highest pri first, Value: " + str(priority))
    return priority
```

APPENDIX C

HIGHEST AVERAGE PRIORITY FIRST CODE

```
class HighAveFirst:
    def __init__(self, topo, vmPairs):
        self.topo = topo
        self.vmPairs = vmPairs
        self.middleboxes = topo.get_middleboxes()
        self.dropped_pairs = 0
        self.cap_flag = 0

    def allocate(self):
        print("Highest Avg priority first start: -running-")
        temp_arr = self.vmPairs

        for index in range(1, len(self.vmPairs)):
            value = self.vmPairs[index]
            i = index - 1
            while i >= 0:
                if (value.get_priority()/value.get_communication_frequency()) > \

(self.vmPairs[i].get_priority()/self.vmPairs[i].get_communication_frequency()):
                    temp_arr[i + 1] = self.vmPairs[i]
                    temp_arr[i] = value
                i -= 1
            else:
                break
        self.vmPairs = temp_arr


    def cap_check(self, pair):
        mb_counter = 0
        path = []


        vm1, vm2 = pair.get_vms()
        if len(self.middleboxes) != 0:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name()
            ))
        else:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
                vm2.get_parent().get_name()
            ))


        if len(self.middleboxes) > 1:
```

```
    mb_counter = mb_counter + 1  # start at 1
    while mb_counter < len(self.middleboxes):
        path.extend(self.topo.get_path(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name()
        )[1:])
        mb_counter = mb_counter + 1
    path.extend(self.topo.get_path(
        self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
        vm2.get_parent().get_name()
    )[1:])
master_edge_seq = self.topo.es
path_edges = []
count = 0
for l in path:
    for e in master_edge_seq:
        if count < len(path) - 1:
            if (str(e.tuple[0]) == str(l) and str(e.tuple[1]) == str(path[count + 1])) \
                    or (str(e.tuple[0]) == str(path[count + 1]) and str(e.tuple[1]) == str(l)):
                path_edges.append(e)
                count = count + 1
for single in path_edges:
```

```
            checker = single['capacity'] - pair.get_communication_frequency()
            single['capacity'] = single['capacity'] - pair.get_communication_frequency()
            if checker < 0:
                self.cap_flag = 1
        for single_revert in path_edges:
            single_revert['capacity'] = single_revert['capacity'] +
pair.get_communication_frequency()
        # used to print the path, used for debug
        # print("path: " + str(path))


    def run_alg(self):
        pair_cost = 0
        priority = 0
        # once pairs are ordered, calc cost
        for pair in self.vmPairs:
            self.cap_check(pair)
            vm1, vm2 = pair.get_vms()
            # print("pair " + str(vm1.get_name()) + " " + str(vm2.get_name()) + " pri: "
            #       + str(pair.get_priority()) + " cost: " + str(pair.run_alg_calc())
            #       + " freq: " + str(pair.get_communication_frequency()))

            if self.cap_flag == 0:
```

```
mb_counter = 0
priority += pair.get_priority()
if len(self.middleboxes) != 0:
    pair_cost += self.topo.get_distance_new(
        vm1.get_parent().get_name(),
        self.middleboxes[mb_counter].get_parent_switch().get_name(),
        pair.get_communication_frequency()
    )
    mb_counter = mb_counter + 1
    while mb_counter < len(self.middleboxes):
        pair_cost += self.topo.get_distance_new(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name(),
            pair.get_communication_frequency()
        )
        mb_counter = mb_counter + 1
    pair_cost += self.topo.get_distance_new(
        self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
        vm2.get_parent().get_name(),
        pair.get_communication_frequency()
    )
else:
```

```
            pair_cost += self.topo.get_distance_new(
                vm1.get_parent().get_name(),
                vm2.get_parent().get_name(),
                pair.get_communication_frequency()
            )
        self.cap_flag = 0
    else:
        self.dropped_pairs = self.dropped_pairs + pair.get_communication_frequency()
        self.cap_flag = 0
# print("highest pri first, dropped packets: " + str(self.dropped_pairs))
# print("highest pri first, Value: " + str(priority))
return priority
```

APPENDIX D

DYNAMIC PROGRAMMING CODE

```
class DynamicProg:
    def __init__(self, topo, vmPairs):
        self.topo = topo
        self.vmPairs = vmPairs
        self.middleboxes = topo.get_middleboxes()
        self.lowest_spine = 0
        self.path_edges = []
        self.path = []
        self.cache = {}
        self.vms = ()
        self.max_cap = 0

    def allocate(self):
        print("Dynamic Prog: -running-")
        mb_counter = 0
        lowest_spine = 0
        first_pass = 1
        path = []
        if len(self.middleboxes) > 1:
            mb_counter = mb_counter + 1  # start at 1
            while mb_counter < len(self.middleboxes):
```

```
        if first_pass == 1:
            path.extend(self.topo.get_path(
                self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name()
            ))
            first_pass = 0
        else:
            path.extend(self.topo.get_path(
                self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name()
            )[1:])
        mb_counter = mb_counter + 1
else:
    lowest_spine = 0


master_edge_seq = self.topo.es
path_edges = []
count = 0
for l in path:
    for e in master_edge_seq:
        if count < len(path) - 1:
            if (str(e.tuple[0]) == str(l) and str(e.tuple[1]) == str(path[count + 1])) \
                    or (str(e.tuple[0]) == str(path[count + 1]) and str(e.tuple[1]) == str(l)):
                path_edges.append(e)
                count = count + 1
                # print("edge link: " + str(e.tuple[0]) + "-" + str(e.tuple[1]) +
str(e.attributes()))

        self.path_edges = path_edges
        self.path = path

    def spine_checker(self, path_edges, path):
        # how many times does each edge get crossed in spine
        # cap of links / # of times crossed
        # return the lowest
        counter = 0
        path_tuple_counter = 0
        dest_index = 1
        flag = 0
        path_tuple = []
        for single in path_edges[:len(path_edges)]:
            z = path[counter], path[dest_index], single['capacity'], 1
            for tuple_single in path_tuple:
                if (tuple_single[0] == z[0] and tuple_single[1] == z[1]) \
                    or (tuple_single[1] == z[0] and tuple_single[0] == z[1]):
                  tuple_single = tuple_single[0], tuple_single[1], tuple_single[2],
(tuple_single[3] + 1)
                  flag = 1
                if flag == 1:
                  path_tuple[path_tuple_counter] = tuple_single
                path_tuple_counter += 1
              if flag == 0:
                path_tuple.append(z)
              flag = 0
              path_tuple_counter = 0
              counter += 1
              dest_index += 1
            lowest_available_cap = int(path_tuple[0][2] / path_tuple[0][3])
            for tup in path_tuple:
              if int(tup[2] / tup[3]) < lowest_available_cap:
                lowest_available_cap = int(tup[2] / tup[3])
            self.max_cap = lowest_available_cap + 1
            return lowest_available_cap

          def dyn_setup(self):
            t = self.spine_checker(self.path_edges, self.path)
```

```
    @staticmethod
    def total_value(vms, max_weight):
        return sum([x[2] for x in vms]) if sum([x[1] for x in vms]) < max_weight else 0


    def solve(self, vms, max_weight):
        if not vms:
            return ()
        if (vms, max_weight) not in self.cache:
            tail = vms[1:]
            head = vms[0]


            chose = (head,) + self.solve(tail, max_weight - head[1])
            dont_chose = self.solve(tail, max_weight)
            if self.total_value(chose, max_weight) > self.total_value(dont_chose,
max_weight):
                item_choice = chose
            else:
                item_choice = dont_chose
            self.cache[(vms, max_weight)] = item_choice
        return self.cache[(vms, max_weight)]
```

```
def pair_dyn_setup(self):
    dyn_pairs = ()
    for pair in self.vmPairs:
        vm1, vm2 = pair.get_vms()
        x = str(vm1.get_name()) + " (" + str(vm1.get_parent().get_label()) + ")" \
            + str(vm2.get_name()) + " (" + str(vm2.get_parent().get_label()) + ")", \
            pair.get_communication_frequency(), pair.get_priority()
        dyn_pairs = (x,) + dyn_pairs
    return dyn_pairs


def cap_check(self, pair):
    mb_counter = 0
    path = []

    vm1, vm2 = pair.get_vms()
    if len(self.middleboxes) != 0:
        path.extend(self.topo.get_path(
            vm1.get_parent().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name()
        ))
    else:
        path.extend(self.topo.get_path(
            vm1.get_parent().get_name(),
            vm2.get_parent().get_name()
        ))

    if len(self.middleboxes) > 1:
        mb_counter = mb_counter + 1  # start at 1
        while mb_counter < len(self.middleboxes):
            path.extend(self.topo.get_path(
                self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name()
            )[1:])
            mb_counter = mb_counter + 1
        path.extend(self.topo.get_path(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            vm2.get_parent().get_name()
        )[1:])
    master_edge_seq = self.topo.es
    path_edges = []
    count = 0
    for l in path:
        for e in master_edge_seq:
            if count < len(path) - 1:
                    if (str(e.tuple[0]) == str(l) and str(e.tuple[1]) == str(path[count + 1])) \
                            or (str(e.tuple[0]) == str(path[count + 1]) and str(e.tuple[1]) == str(l)):
                        path_edges.append(e)
                        count = count + 1
            for single in path_edges:
                checker = single['capacity'] - pair.get_communication_frequency()
                single['capacity'] = single['capacity'] - pair.get_communication_frequency()
                if checker < 0:
                    self.cap_flag = 1
            for single_revert in path_edges:
                single_revert['capacity'] = single_revert['capacity'] +
    pair.get_communication_frequency()
            # used to print the path, used for debug
            # print("path: " + str(path))


        def run_alg(self):
            pair_cost = 0
            priority = 0
            # once pairs are ordered, calc cost
            for pair in self.vmPairs:
                self.cap_check(pair)
                vm1, vm2 = pair.get_vms()
```

```
# print("pair " + str(vm1.get_name()) + " " + str(vm2.get_name()) + " pri: "
#       + str(pair.get_priority()) + " cost: " + str(pair.run_alg_calc())
#       + " freq: " + str(pair.get_communication_frequency()))
if self.cap_flag == 0:
    mb_counter = 0
    priority += pair.get_priority()
    if len(self.middleboxes) != 0:
        pair_cost += self.topo.get_distance_new(
            vm1.get_parent().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name(),
            pair.get_communication_frequency()
        )
        mb_counter = mb_counter + 1
        while mb_counter < len(self.middleboxes):
            pair_cost += self.topo.get_distance_new(
                self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name(),
                pair.get_communication_frequency()
            )
            mb_counter = mb_counter + 1
        pair_cost += self.topo.get_distance_new(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            vm2.get_parent().get_name(),
            pair.get_communication_frequency()
        )
    else:
        pair_cost += self.topo.get_distance_new(
            vm1.get_parent().get_name(),
            vm2.get_parent().get_name(),
            pair.get_communication_frequency()
        )
    self.cap_flag = 0
else:
    self.dropped_pairs = self.dropped_pairs + pair.get_communication_frequency()
    self.cap_flag = 0
# print("Random, dropped packets: " + str(self.dropped_pairs))
# print("Random, Value: " + str(priority))
return priority
```

APPENDIX E

RANDOM ALGORITHM CODE

```
class RandomAlg:
    def __init__(self, topo, vmPairs):
        self.topo = topo
        self.vmPairs = vmPairs
        self.middleboxes = topo.get_middleboxes()
        self.dropped_pairs = 0
        self.cap_flag = 0

    def allocate(self):
        t = self.middleboxes
        print("Random Running" + str(t[0]))

    def cap_check(self, pair):
        mb_counter = 0
        path = []

        vm1, vm2 = pair.get_vms()
        if len(self.middleboxes) != 0:
            path.extend(self.topo.get_path(
                vm1.get_parent().get_name(),
        self.middleboxes[mb_counter].get_parent_switch().get_name()
    ))
else:
    path.extend(self.topo.get_path(
        vm1.get_parent().get_name(),
        vm2.get_parent().get_name()
    ))

if len(self.middleboxes) > 1:
    mb_counter = mb_counter + 1  # start at 1
    while mb_counter < len(self.middleboxes):
        path.extend(self.topo.get_path(
            self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
            self.middleboxes[mb_counter].get_parent_switch().get_name()
        )[1:])
        mb_counter = mb_counter + 1
    path.extend(self.topo.get_path(
        self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
        vm2.get_parent().get_name()
    )[1:])
master_edge_seq = self.topo.es
path_edges = []
```

```
        count = 0
        for l in path:
            for e in master_edge_seq:
                if count < len(path) - 1:
                    if (str(e.tuple[0]) == str(l) and str(e.tuple[1]) == str(path[count + 1])) \
                            or (str(e.tuple[0]) == str(path[count + 1]) and str(e.tuple[1]) == str(l)):
                        path_edges.append(e)
                        count = count + 1
        for single in path_edges:
            checker = single['capacity'] - pair.get_communication_frequency()
            single['capacity'] = single['capacity'] - pair.get_communication_frequency()
            if checker < 0:
                self.cap_flag = 1
        for single_revert in path_edges:
            single_revert['capacity'] = single_revert['capacity'] +
pair.get_communication_frequency()
        # used to print the path, used for debug
        # print("path: " + str(path))


    def run_alg(self):
        pair_cost = 0
        priority = 0
```

```
# once pairs are ordered, calc cost
for pair in self.vmPairs:
    self.cap_check(pair)
    vm1, vm2 = pair.get_vms()
    # print("pair " + str(vm1.get_name()) + " " + str(vm2.get_name()) + " pri: "
    #       + str(pair.get_priority()) + " cost: " + str(pair.run_alg_calc())
    #       + " freq: " + str(pair.get_communication_frequency()))
    if self.cap_flag == 0:
        mb_counter = 0
        priority += pair.get_priority()
        if len(self.middleboxes) != 0:
            pair_cost += self.topo.get_distance_new(
                vm1.get_parent().get_name(),
                self.middleboxes[mb_counter].get_parent_switch().get_name(),
                pair.get_communication_frequency()
            )
            mb_counter = mb_counter + 1
            while mb_counter < len(self.middleboxes):
                pair_cost += self.topo.get_distance_new(
                    self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                    self.middleboxes[mb_counter].get_parent_switch().get_name(),
                    pair.get_communication_frequency()
                )
                mb_counter = mb_counter + 1
            pair_cost += self.topo.get_distance_new(
                self.middleboxes[mb_counter - 1].get_parent_switch().get_name(),
                vm2.get_parent().get_name(),
                pair.get_communication_frequency()
            )
        else:
            pair_cost += self.topo.get_distance_new(
                vm1.get_parent().get_name(),
                vm2.get_parent().get_name(),
                pair.get_communication_frequency()
            )
        self.cap_flag = 0
    else:
        self.dropped_pairs = self.dropped_pairs + pair.get_communication_frequency()
        self.cap_flag = 0
# print("Random, dropped packets: " + str(self.dropped_pairs))
# print("Random, Value: " + str(priority))
return priority
```

APPENDIX F

TOPOLOGY GENERATION CODE

```
import igraph

import math as m

import random as r

import NetworkComponents as networks



class topoGen:


    def __init__(self, nodes, children, numPairs, numMB, physicalCap=None,

link_cap=None):

        self.nodes = nodes

        self.children = children

        self.numPairsairs = numPairs

        self.n_middleboxes = numMB

        self.max_vm_size = 1


        self.hosts, self.edgeSwitches = [], [], [], []

        self.middleboxes = []

        self.master_graph = None


        self.physicalCap = physicalCap
        self.link_cap = link_cap
        self.es = None
        self.freq = 1

    def create_topology(self):
        tree = igraph.Graph.Tree(self.nodes, self.children)
        host_count = 0
        create_count = 0
        e_count = 0
        pod_number, pod_index = 0, 0
        vertical_number = 1
        for g in tree.vs:
            g["label"] = create_count
            g["host_flag"] = 0
            g["edgeSwitch"] = "None"
            g["host"] = "None"
            g["name"] = "None"
            flag = 0
            for e in tree.es:
                if create_count == e.tuple[0] or create_count == e.tuple[1]:
                    flag = flag + 1
            if flag == 1:
```

```
    g["host_flag"] = 1
    host_count = host_count + 1
    host_name = "host_" + str(host_count)
    if self.physicalCap is None:
        host_capacity = int(m.ceil((self.numPairsairs * 2.0 * self.max_vm_size)))
    else:
        host_capacity = self.physicalCap
    host = networks.PhysicalMachine(host_capacity, host_name, g["label"])
    g["host"] = tree.vs.select()
    g["name"] = str(host_name)
    self.hosts.append(host)
else:
    e_count = e_count + 1
    edgeSwitch_name = "edgeSwitch_" + str(e_count)
    edgeSwitch = networks.PhysicalSwitch(edgeSwitch_name)
    edgeSwitch.set_label(g["label"])
    edgeSwitch.set_p(pod_number)
    edgeSwitch.set_v(vertical_number)
    edgeSwitch.set_h(2)
    pod_index += 1
    vertical_number += 1
    g["edgeSwitch"] = tree.vs.select()
```

```
            g["name"] = edgeSwitch_name
            self.edgeSwitches.append(edgeSwitch)

        create_count = create_count + 1

    self.master_graph = tree
    self.es = igraph.EdgeSeq(self.master_graph)
    self.es["weight"] = 1
    self.es["capacity"] = 0
    self.es["capacity_val"] = 0
    for edge in self.es:
        edge["capacity"] = self.link_cap  # randomize here
        edge["capacity_val"] = self.link_cap  # randomize here

def create_middleboxes(self):
    # make first and last middlebox an edge switch here

    middlebox_host = []
    self.randomEdgeSwitch()
    for i in range(1, self.n_middleboxes + 1):
        if self.n_middleboxes > 1:
            if i == 1 or i == self.n_middleboxes:
```

```
            new_middlebox = networks.MiddleBox("MiddleBox_" + str(i))
            random_parent_switch = self.randomEdgeSwitch()

            while random_parent_switch.get_name() in middlebox_host:
                random_parent_switch = self.randomEdgeSwitch()

            new_middlebox.set_parent_switch(random_parent_switch)
            middlebox_host.append(random_parent_switch.get_name())
            self.middleboxes.append(new_middlebox)
        else:
            new_middlebox = networks.MiddleBox("MiddleBox_" + str(i))
            random_parent_switch = self.randomSwitch()

            while random_parent_switch.get_name() in middlebox_host:
                random_parent_switch = self.randomSwitch()

            new_middlebox.set_parent_switch(random_parent_switch)
            middlebox_host.append(random_parent_switch.get_name())
            self.middleboxes.append(new_middlebox)

def randomSwitch(self):
    #layer_check = r.randint(0, 3)
    switches = self.edgeSwitches
    return switches[r.randint(1, len(switches) - 1)]

def randomEdgeSwitch(self):
    switches = []
    index = []
    for z in self.master_graph.vs:
        if z['host_flag'] == 1:
            index.append(self.master_graph.neighbors(z)[0])
    for x in self.edgeSwitches:
        if x.get_label() in index:
            switches.append(x)
    return switches[r.randint(1, len(switches) - 1)]

def get_hosts(self):
    return self.hosts

def getGraph(self):
    return self.master_graph

def getMiddleboxes(self):
    return self.middleboxes
    def setMiddleboxes(self, mbs):
        self.middleboxes = mbs


    def get_path(self, node1, node2):
        path = self.master_graph.get_shortest_paths(
            node1, to=node2, weights='weight', mode=igraph.OUT, output='vpath')
        return path[0]


    def get_distance_new(self, node1, node2, freq):
        # self.master_graph.es[0]["capacity"] = 1
        self.freq = freq
        path = self.master_graph.get_shortest_paths(
            node1, to=node2, weights='weight', mode=igraph.OUT, output='vpath')
        # print("p: " + str(path[0])) # prints path, uncomment to debug
        counter = 0
        link_full = False
        while counter < len(path[0]) - 1:
            for e in self.es:
                if (e.tuple[0] == path[0][counter] and e.tuple[1] == path[0][counter + 1]) or
(e.tuple[1] ==
                                                        path[0][counter] and e.tuple[0] == path[0][counter
+ 1]):
                if e["capacity"] <= 0:
                    link_full = True
                e["capacity"] = e["capacity"] - self.freq
                if e["capacity"] <= 0:
                    e["weight"] = 999999
                    # self.master_graph.delete_edges(e)
            counter = counter + 1
        if link_full:
            # if full set to -1 but for now the same
            # cost = -1
            cost = len(path[0]) - 1
        else:
            cost = len(path[0]) - 1
        return cost
```