EFFICIENT VIRTUAL MACHINE REPLICATION AND SERVER CONSOLIDATION USING MINIMUM COST FLOW AND BIN PACKING

_____

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

Alexander Toneff

Spring 2020

PROJECT: EFFICIENT VIRTUAL MACHINE REPLICATION AND SERVER CONSOLIDATION USING MINIMUM COST FLOW AND BIN PACKING

AUTHOR: ALEXANDER TONEFF

APPROVED:

_____

Bin Tang, Ph.D.

Project Committee Chair

_____

Jack Han, Ph.D.

Committee Member

_____

Mohsen Beheshti, Ph.D.

Department Chair, Committee Member

ACKNOWLEDGEMENTS

This work is built entirely on Professor Bin Tang's work. Thank you to Dr. Tang for patiently explaining the concepts to me. Thank you to Dr. Han, Dr. Beheshti, for taking the time to offer feedback and support me through this process. An additional thank you to all the professors and wonderful staff at the CSUDH Computer Science Department who make learning a joy.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This project shows a complete solution for efficient VM replication in a fat tree data center, modeled as a minimum cost flow problem for optimal replication flow, and then modeled as a bin packing problem for optimal server consolidation. Previously, both optimizations have not been shown in a single work. Using Python 3.6+ and the Google OR-Tools linear solver, results show the optimal algorithms outperform the best heuristics. Future work indicates a model where flow and consolidation can be optimized in a single algorithm, as well as a need for examination of non-linear factors (sensitivity analysis) and the use of machine learning for energy optimization.

Chapter 1: INTRODUCTION

With the explosive growth of cloud infrastructure and worldwide network usage, minimizing the cost of virtual machine replication and data center energy usage has been a topic of interest for practical optimization. In data centers, the problem of moving around data efficiently has been modeled as a minimum cost flow problem [7].

The problem of efficient virtual machine replication in data centers is typically approached by considering the following sub-problems: minimizing flow costs, maximizing server consolidation, and fulfilling service level agreements.

In order to minimize flow cost (i.e., transferring copies of a virtual machine across a network), the topology of the data center is modeled as a graph problem. Like others [7], this paper assumes the fat tree topology for the model.

Maximizing server consolidation involves converting the network topology into a graph as well, and then solving it as a bin packing problem. Both of these types of optimization problems can be solved using linear programming. Linear optimization problems are a large class of problems which are most widely solved by applications of Dantzig's simplex algorithm, devised in 1947 [6].

Besides minimum cost flow and bin packing, other interesting applications of the simplex algorithm include: routing problems (i.e. traveling salesman or delivery optimization), assignment, scheduling, and large classes of linear, constraint, and integer optimizations. Examples of these can be found at [5].

The format of this paper first discusses related work, and why they are different from the

solution being addressed here. Then the 7 algorithms used are explained. There are two stages to the optimization, some algorithms concern the first stage where sum of network flow is minimized (Algorithms 1, 2, 3), and the other algorithms are used in the second stage where server consolidation is maximized (Algorithms 4, 5, 6, 7). To clarify, an optimal solution would only require Algorithm 3 (minimum cost flow) and Algorithm 7 (bin packing with constraints). The others are for comparison to the optimal.

There are often multiple optimal flow solutions, from these we use algorithms to maximize server consolidation. In the two-stage optimization design, the set of consolidation solutions is dependent on the set of optimal flow solutions. In other words, the second stage is searching through flow solutions which are equivalent in flow-cost to the minimum cost flow solution, in order to find an optimal flow solution which uses the least number of servers (maximizes consolidation).

In each of these two stages (flow, consolidation), the optimal algorithms are compared to greedy heuristics. Algorithms (1, 2, 4, 5, 6) are greedy heuristics meant for comparison against the optimal algorithms (3, 7). Greedy heuristics are simpler to understand and implement, but only the optimal algorithms are guaranteed to produce the best solution every time.

After the analysis of the algorithms, we look at implementation. The Implementation section explains the program logic. Each critical piece of logic is discussed, in the order that it executes in the program. The complete code is attached in the appendix. The results show why and how optimal outperforms heuristic, and at the very end there is discussion on direction of future work.

Chapter 2: RELATED WORK

In "Power-efficient virtual machine replication in data centers," P. Khani, B. Tang, J. Han, and M. Beheshti showed the details of the graph transformation of a fat tree topology data center, in order to model virtual machine replication as a minimum cost flow problem with an optimal solution [7]. For reference, a figure diagramming an example fat tree topology where k=4 (k is the arity of the fat tree and determines the magnitude of its structure):

Review of Fat Tree Topology, where k=4

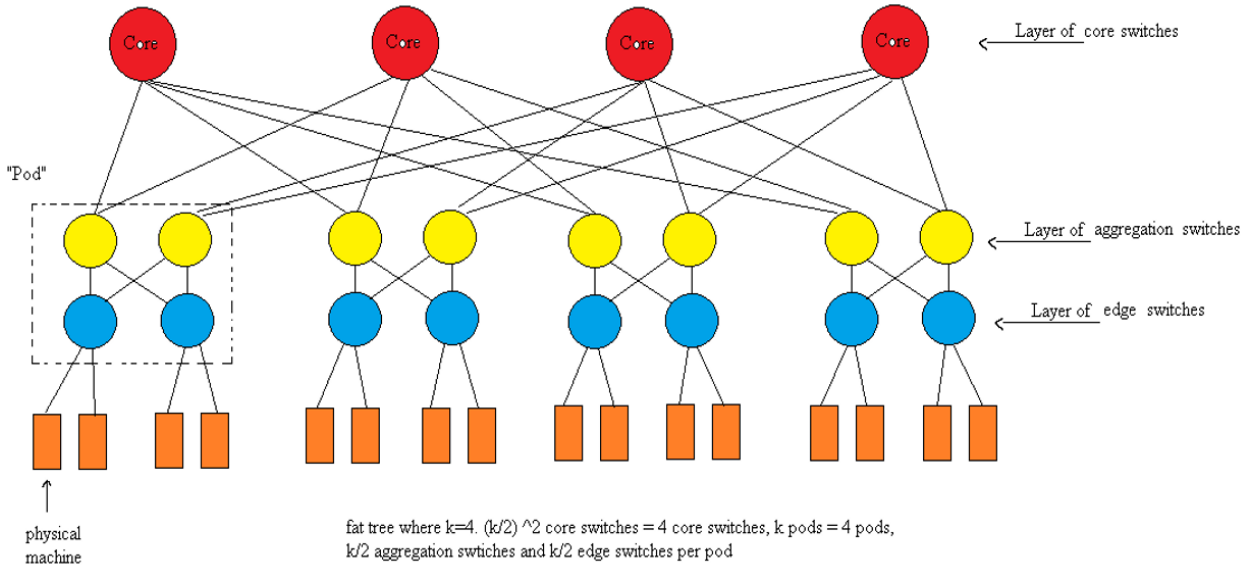

Figure 1 Review of fat tree topology

More information on how the fat tree structure is generated from ‘k’ and why it is a useful network topology structure can be found outside this paper.

The “efficient virtual machine replication” problem was formulated as follows: there are several virtual machines sitting on physical machines in the fat tree topology. Each

‘original’ virtual machine wants to send R replica copies of itself to other physical machines, using the shortest network routing paths possible. Minimizing the sum of flow cost of all selected routing paths was the primary objective, subject to the following constraints: for safety, no two copies of a virtual machine could ever occupy the same physical machine; physical machines were modeled to have a limited space capacity.

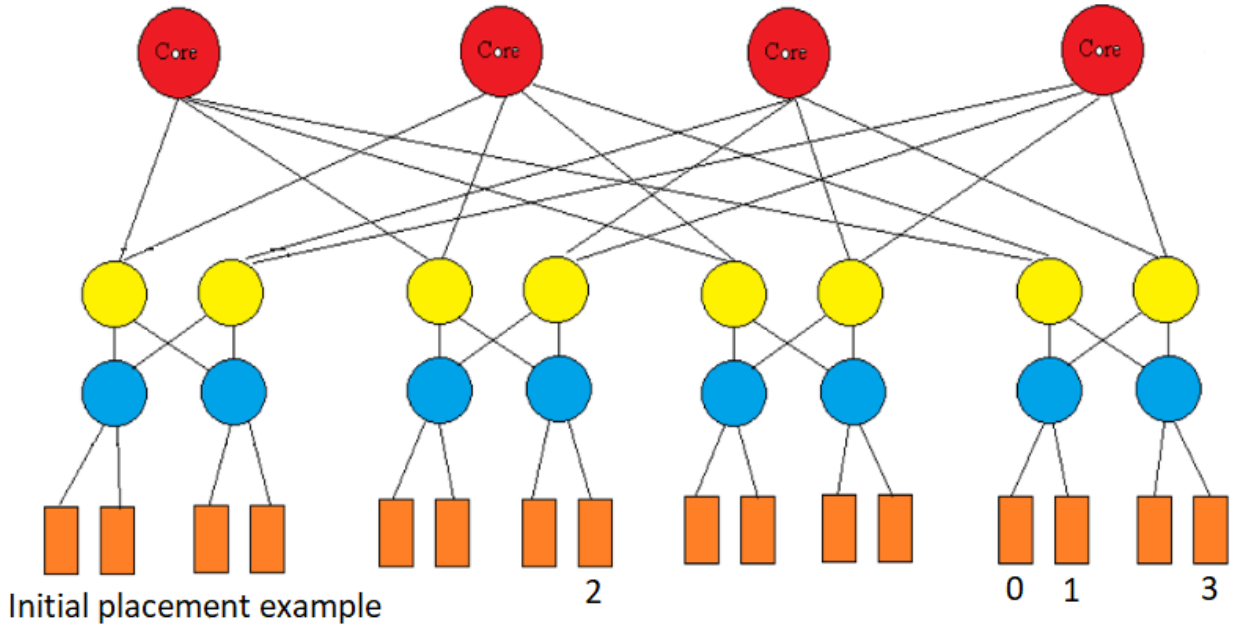

Figure 2 Initial VM placement example

In the illustration above, the original virtual machines 0, 1, 2, 3 are initially located at physical machines 12, 13, 7, 15 respectively (physical machines are labeled 0 - 15).

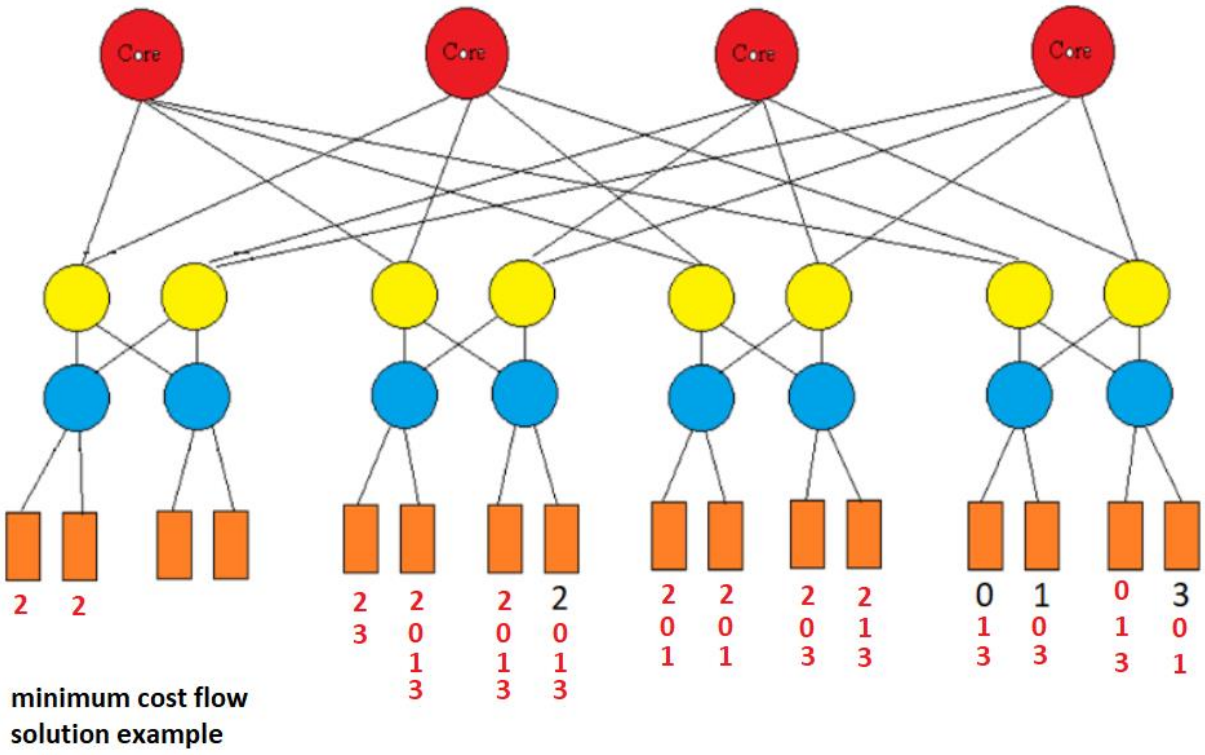

Figure 3 Minimum cost flow solution example

In the illustration above, we see an example minimum cost flow solution for the previous figure. Each VM replicates 9 times (red numbers). Virtual machine replica copies are indicated by the red numbers corresponding with their original virtual machine id: 0, 1, 2, or 3. The black numbers 0, 1, 2, 3 are the original virtual machines which never moved. For a detailed explanation of replica choices, please see Figure 17 on page 16.

Khani et al.'s results compare the optimal flow cost of the minimum cost flow solution to some simple greedy heuristics, such as a heuristic where each virtual machine copy 'chooses' its cheapest routing path. The heuristic is quite good, performing almost as well as optimal in most cases, but it is not as good as optimal because the order in which each virtual machine copy can pick is arbitrary. In more complex conditions,

sometimes one virtual machine copy will greedily choose a destination physical machine that could have been used more efficiently by a different virtual machine copy.

Khani et al. ran into a problem when it came to server consolidation: the minimum cost flow solution gave them an arbitrary number of destination physical machines [7]. For example, imagine a minimum flow cost solution where total flow cost is 60, and 10 physical machines end up holding virtual machines. Compare this to a solution where total flow cost is still 60, but 8 physical machines end up being used. Even though a solution using fewer physical machines is more desirable (because unused machines can be slept to save power), the minimum cost flow algorithm would arbitrarily choose any solution which minimized flow cost. This is due to the way the problem was modeled, see Figure 4. There was no modeled cost representing activation of a server.

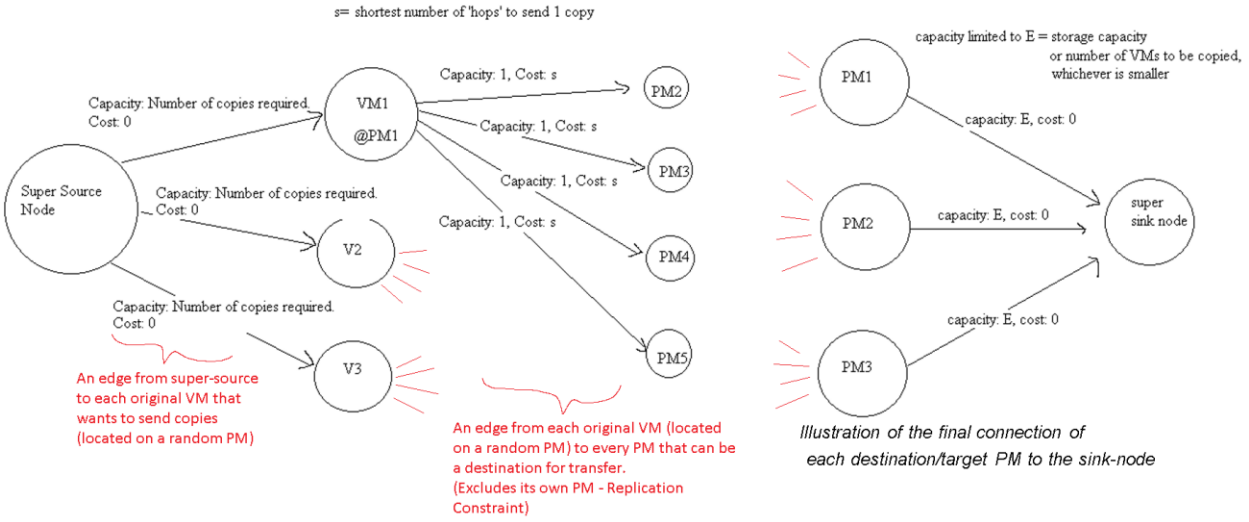

Figure 4 Khani, Tang, et. al graph transformation

As a side note, when modeling assignment (assign each virtual machine copy to a destination physical machine) as a minimum cost flow problem, it is common to use a "super sink" and a "super source" node to aggregate all the supply and demand into just

1 source node and 1 sink node. This makes the problem compatible with a larger class of solvers than those which can handle multiple source and sink nodes.

As the illustration of their graph transformation shows, there is no cost on the final set of edges from each physical machine to the super sink node, effectively making it ‘free’ to use an arbitrary number of servers in the solution. While this graph transformation allows a solver to minimize flow cost, it does not address server consolidation.

After the minimum cost flow algorithm produces an optimal flow cost solution, Khani et al. then used a second ‘consolidation stage’ to try to consolidate servers used by reexamining choices made by the minimum cost flow solution. Their consolidation algorithms were greedy heuristics which were not optimal, thus leaving room for improvement. However, this problem can be solved optimally by being modeled as a bin packing problem with constraints, which is what this paper does.

H. Goudarzi and M. Pedram have the most cited work concerning efficient virtual machine replication, in their paper "Energy-Efficient Virtual Machine Replication and Placement in a Cloud Computing System," (2012) [2]. Their paper focuses on fulfilling service level agreements in a cloud environment, which means ensuring the correct amount of CPU and memory are allocated to each virtual machine. In addition, they focus on server consolidation of virtual machines. Their work does not consider using virtual machine replicas as backup copies for safety, nor is flow cost ever considered.

X. Dai, J. M. Wang and B. Bensaou, in "Energy-Efficient Virtual Machines Scheduling in Multi-Tenant Data Centers," (2016) [9] have a paper similar to Goudarzi and Pedram concerning the fulfillment of service level agreements and server consolidation in cloud computing. They concluded an optimal solution would take too long to be practical, and they compared their greedy heuristic algorithms for server consolidation to an optimal one achieved by Gurobi (a privatized linear solver). Again, their work does not consider flow cost.

In “Improved Filtering for the Bin-Packing with Cardinality Constraint” [8], Derval, R’egin, Schaus show how to prune the search tree of bin packing solutions, using a “too-big”, “too-small” concept, reducing computation time spent on impossible assignments. There is no mention of network flow cost or efficient VM replication, thus the scope of their paper is different than this one. However, their work has potential to reduce the bin packing time calculation.

In “Minimum Cost Maximum Flow Algorithm for Dynamic Resource Allocation in Clouds” [10], Hadji and Zeghlache simplify VM assignment by assuming all costs are known by the cloud provider. The big idea of their paper is to predict future cloud resource allocation based on past usage. There are no mentions of network flow cost at all. They claim their algorithm “matches the global optimum most of the time”, which is a contradiction, as an optimal algorithm should guarantee the best solution, every time, all sets of cases considered. They use a simplified cost function where cost is inversely proportional to current usage of that PM. This means no concrete results data on PM

space capacity, etc., as VMs are only classified by “type”, ex: small, medium, large. They exclusively use minimum cost flow or bin packing to solve a simplified graph transformation of the problem. This project uses both MCF and bin packing in sequence.

Currently, it seems that no single research paper produces both optimal network flow and optimal server consolidation together. There may also be a lack of organized data on sensitivity analysis: which factors are truly significant in optimizing data center energy usage? This would involve analysis of whether it is worth the cost of running optimal solutions against simpler greedy heuristics, frequency of recalculation (depending on how dynamic the environment is), actual measurable amount of energy saved, and whether there are other important factors to consider.

This paper details an experiment with an optimal minimum cost flow solution and an optimal server consolidation solution used together. There is some discussion at the end on how to combine these two stages into one, for future work.

Chapter 3: ALGORITHMS AND THEIR ANALYSIS

Algorithm 1: First Fit. Each replica VM goes to the lowest ID available PM without considering flow cost, which is why it achieves maximum consolidation. Available means there is both enough PM space and there are no other copies of this VM already on the target PM. First Fit algorithm is used as a benchmark for maximum consolidation, without any consideration of flow. Pseudocode:

```
For each originating VM:
        needtoplace = number of replica copies
        For each PM:
                If needtoplace ==0:
                        break
                If (this PM has enough space AND this PM does not contain this VM or any copy of this VM):
                        place a replica of this VM on this PM
                        needtoplace = needtoplace - 1
                Else
                        continue
        If needtoplace != 0:
                print "Not all copies were placed"
```

Figure 5 Pseudocode for Algorithm 1

First Fit time complexity: assuming feasibility, in the worst case, each replica VM must search through all the PMs, checking: if each PM already contains any copy of that VM, if that PM has enough space remaining. This means O(numReplicaCopies * numPMs) * (the constant which represents the time it takes to perform the checks for space and any other copies of this VM).

Algorithm 2: Greedy. A greedy heuristic where each VM chooses the lowest flow-cost available destination PM, in arbitrary order. Pseudocode:

```
for each replica VM:
        calculate flow cost to each PM
        for each PM, in order of ascending flow cost:
                If (PM has space AND PM does not contain this VM or any of its copies)
                        place a replica of this VM on this PM
        Check that replica was placed
```

Figure 6 Pseudocode for Algorithm 2

Greedy performance often matches or comes close to minimum cost flow solutions in terms of flow-cost, but it has no consideration of consolidation, and it is not an optimal flow-cost solution due to allowing the replica VMs to choose in arbitrary order, this can lead to one replica VM arbitrarily preventing a later replica VM's optimal choice[1].

Greedy time complexity: each replica VM must sort the list of PMs by ascending flow-cost. It must check each PM for space and whether that PM contains any other copies of this VM. Worst case, this leads to O(numReplicaCopies * (numPM * log numPM) * numPM), where each replicaVM has to sort the list of PMs by flow-cost and doesn't place until the last PM on its sorted list.

Algorithm 3: Minimum Cost Flow. This is the optimal flow algorithm which can guarantee the minimum total flow cost every time. The minimum cost flow algorithm is implemented in Google's OR-Tools in C++, using Goldberg's method, with O(V^2 * E). In this graph transformation, the number of vertices will be numReplicaCopies + numPM , plus 2 (the supersource and supersink). The number of edges is numReplicaCopies + numPM + numReplicaCopies * (numPM-1). See the graph transformation figures.

Algorithm 4: RFF, or ReplacementFirstFit, is a consolidation heuristic which limits each replica VM to a set of target PMs that are equivalent in flow cost to the minimum cost flow solution. This ensures the consolidation solution always has the same flow cost as

MCF's solution(Algorithm 3). Replica VMs are placed in the lowest-id available PM, where "available" means: flow cost to target PM is the same as MCF, PM has enough space, and no other copy of that VM is on the target PM. Pseudocode:

```
for each replica VM:
        for each PM that has the same flow cost as the MCF solution PM, in order of ascending PM ID:
                If (PM has enough space AND this PM does not contain this VM or any copies)
                        place a replica of this VM on this PM
        Check that this replica was placed
```

Figure 7 Pseudocode for Algorithm 4

ReplacementFirstFit's time complexity: Since optimal flow costs are already given as a result of the MCF solution, each replica copy only searches for the lowest-ID-available-PM from a list of target PMs which are equivalent in flow cost to the MCF solution, while also considering whether the target PM has enough space and whether there is already a replica copy of that VM on the target PM. The worst case is O(numReplicaCopies * numPMs).

Algorithm 5: MCF-Placed-FirstFit, just like Algorithm 4 RFF, is a consolidation heuristic which starts from the minimum cost flow solution and examines each PM to try to move any VMs to the lowest-id available PM, while respecting equivalent flow-costs. The key difference is that Algorithm 5 keeps the MCF placement as a starting point and attempts to transfer to lower ID PMs, whereas RFF redoes all placements. Pseudocode:

```
for each replica VM:
        for each PM that has the same flow cost as the MCF solution PM, in order of ascending PM ID:
                If (PM is the same as the MCF solution placement OR this PM was not activated in MCF solution)
                        continue
                If (PM has enough space AND this PM does not contain this VM or any copies)
                        transfer replica VM from the MCF solution PM to this PM
        Check that this replica was placed
```

Figure 8 Pseudocode for Algorithm 5

Algorithm 6: Khani et al.'s best consolidation heuristic is described as follows:

(1) Look at each PM holding exactly 1 VM, in arbitrary order.
(2) For each of these PMs, check if each of its contained VMs can be moved elsewhere (if the move is equivalent in cost to the MCF solution).
(3) If all contained VMs can be moved, move them all. If not, don't move any.
(4) Then repeat steps 1-3 with PMs holding 2 VMs, then repeat steps 1-3 with PMs with 3 VMs, ...until you reach the maximum possible number of VM per PM.

Algorithm 6 outperforms Algorithms 4 and 5, because it starts by trying to consolidate the PMs holding the lowest number of VMs; intuitively, PMs containing the fewest VMs are the most likely to be consolidated. They are the "low-hanging fruit" for consolidation. At worst case, the algorithm will run numReplicaCopies times over all PMs that do not contain an original VM, checking each replicaVM for valid alternative PM choices (numReplica * numPM). This means a worst possible runtime of O (numReplicaCopies * (numReplicaVMs * numPMs)). In the above implementation, to try to speed up the algorithm, PMs containing an original VM are excluded from VM examination, and the remaining list of PMs is sorted ascending by number of contained VMs.

Algorithm 7: the optimal bin packing solution has *no* known polynomial time algorithm. It is based on Dantzig's simplex algorithm, the most efficient algorithm commonly used to solve linear programming problems. There are different variations of implementations of Dantzig's simplex algorithm, however these can always be shown to have a class of problems for which the runtime is exponential.

When setting stress is relatively low, there is usually no way for the optimal flow and

consolidation algorithms to outperform the heuristics, as choices are not constrained enough for a greedy heuristic to be forced into a local optimum while missing the global optimum of flow and consolidation.

Additionally, when setting stress is relatively high, it becomes so difficult for a solution to meet the constraints that there may be no room left over for any intelligent decision making in which optimal outperforms the heuristics.

We will see that in low or high stress conditions, optimal does not offer any improvement over Algorithm 6. There is a ‘sweet spot’ of stress where conditions are stressful enough that heuristics make mistakes, but not so stressful that there is no room for intelligent decision making to prevail after meeting the constraints. This ‘sweet spot’ of stress is where optimal most notably outperforms heuristics. This observation applies to both the flow-optimization-stage and the server-consolidation stage.

Chapter 4: IMPLEMENTATION

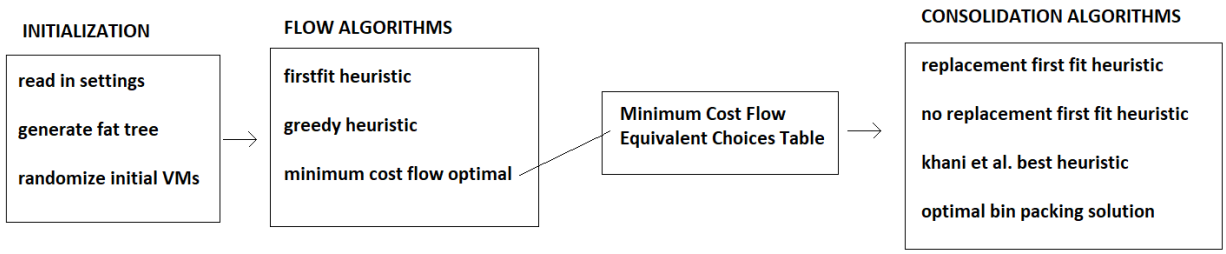

Figure 9 High level program flow

Above is a high level flow chart of the program. Initialization involves reading in the settings.

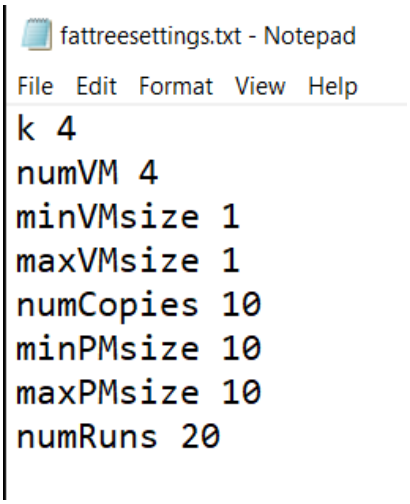

Figure 10 Initial settings text file

In the settings, k is the arity of the fat tree, which can be any positive even integer. A k=4 fat tree has 16 PMs, a k=8 fat tree has 128 PMs. Generalized, the fat tree will have ((k^3) / 4) PMs. numVM is the number of original VMs, which have randomized placement. The min and max VM size are the allowed inclusive range for randomized VM size. numCopies is the total number of copies of each VM, including the original and

replica copies. Min and max PM size are the allowed inclusive range for randomized PM space capacity. numRuns is used by the write-only version (no monitor output) of the program, used for comparing X number of runs results in a table which is outputted in fattreeresults.txt.

As the high level flow chart indicates, the minimum cost flow solution is turned into a table representing equivalent flow-cost PM choices for each replica VM copy. This table of choices is used as input to each of the various consolidation algorithms, which are described later in this paper.

In order to model these types of problems, some environment setup is required. The least amount of setup is using Google's OR-Tools, a free, open-source set of linear solvers that can handle a wide variety of optimization problems. Google's OR-Tools can be installed with python "version 3.5+ on Linux, or 3.6+ on Mac OS or Windows" [3]. A 64 bit system is required. Once Python 3.6+ and pip are installed, the easiest way is to pip install using the command:

```
python -m pip install --upgrade --user ortools
```

More information detailing the installation process as well as OR-Tools solving capabilities can be found on google's developer website [3][4]. Example optimization problems are included with explanations of how to solve them using OR-Tools. OR-Tools also has the option to be built from source with any other linear solver engine placed on top of it: OR-tools, Gurobi, SCIP, GLPK.

Once Python 3.6+ and OR-Tools are successfully installed, in order to input the

topology into the minimum cost flow solver, we will need to define the graph as follows: (start-node, end-node, capacity, unit-cost) for each arc(a.k.a. edge) in the graph. The unit-cost, a.k.a. flow-cost, can be easily calculated by observing the following properties about the fat tree structure: for any positive even integer k, PMs (physical machines) which share the same edge switch are 2 network 'hops' apart.

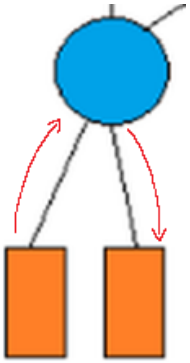

Figure 11 Two hop example

Above, illustration of a 2-hop path, through edge switch, in k=4 fat treePMs which share the same pod, but not the same edge switch, are always 4 network 'hops' apart:

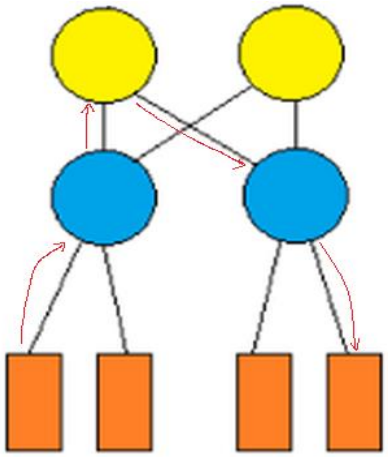

Figure 12 Four hop example

Above, example of a 4-hop path, k=4, using aggregate switch (yellow).

PMs which do not share the same edge switch and do not share the same pod are always 6 ‘hops’ apart, and must route through a core switch. This observation allows us to define every possible routing cost from any PM to any other PM by knowing ‘k’.

Labeling each PM starting from 0, some index math allows us to identify whether PMs are connected most closely by the same edge switch, aggregate switch, or require a core switch, and thus assign either 2, 4, or 6 network ‘hops’ as their respective routing cost. These are compiled into a dictionary (aka hash map) structure for faster lookups. Key = (pm1 ID, pm2 ID). Value = Flow-cost between the two PMs.

We define the super source and super sink nodes, giving them an index number that does not conflict with the assigned PM ids, and set their supply and demand to be opposites of each other. The supply will be the sum of sizes of all replica copies being transferred, since original VMs are never moved. The demand will be supply * -1.

Following the transformation in Khani et al., we generate arcs from the super source node to each original VM, accounting for each VM’s size. Next we generate an arc from each source PM to any other PM that can be a potential destination PM (excluding itself): The capacity of each of these arcs is set to 1, ensuring that no PM ever receives more than 1 copy of a given VM, satisfying the replication constraint.

Finally, each potential destination PM is connected to the super sink node. The capacity of each of these arcs is set to that destination PM’s remaining space capacity, satisfying the PM capacity constraint.

From here it is just a matter of letting OR-Tools do all the work in computing a solution

and organizing its output into human readable form: which arcs were used, to exactly what capacity, along with the total flow-cost of the solution. After processing, we want to see which PMs now contain copies of which VMs.

```
Minimum Cost Flow Solution: Pre-Consolidation
Minimum cost: 184

PM-  | PM-space | PM-max-   | Assigned-
ID   | used     | capacity | VM(s)
-------------------------------------------------
 0       1         30          2
 1       1         30          2
 2       0         30
 3       0         30
 4       2         30          2    3
 5       4         30          2    0    1    3
 6       4         30          2    0    1    3
 7       4         30          2    0    1    3
 8       3         30          2    0    1
 9       3         30          2    0    1
 10      3         30          2    0    3
 11      3         30          2    1    3
 12      3         30          0    1    3
 13      3         30          1    0    3
 14      3         30          0    1    3
 15      3         30          3    0    1

4 PMs originally used. 14 PMs used after Minimum Cost Flow: pre-consolidation.
```

Figure 13 Sample MCF output

While we have found an optimal flow-cost solution using the minimum cost flow solver, we have not yet optimized server consolidation. Like Khani et al., we can reexamine the choices made by the optimal minimum cost flow solution and come up with a list of equivalent cost choices for each VM replica copy that was moved. Once again, we leverage our observation of how routing costs in the fat tree structure will always predictably be 2, 4, or 6 network ‘hops’ and, once more, we make use of the cost dictionary we compiled earlier.

Doing so allows us to form a pre-consolidation table detailing each replica VM copy’s

journey, where we can see: the originating VM id of that replica copy, the destination PM it was assigned by the minimum cost flow solution, the routing/flow cost of that assignment (always 2, 4, or 6 'hops'), and a list of destination PMs that would be the same cost as the MCF solution (including the one chosen by the MCF solution):

```
Pre-consolidation table:

vmID origPM PM   cost PMchoices:same-flow-cost,includes MCFs arbitrary choice
0    12     13   2    [13]
0    12     14   4    [14, 15]
0    12     15   4    [14, 15]
0    12     5    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
0    12     6    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
0    12     7    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
0    12     8    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
0    12     9    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
0    12     10   6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
1    13     12   2    [12]
1    13     14   4    [14, 15]
1    13     15   4    [14, 15]
1    13     5    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
1    13     6    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
1    13     7    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
1    13     8    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
1    13     9    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
1    13     11   6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
2    7      6    2    [6]
2    7      4    4    [4, 5]
2    7      5    4    [4, 5]
2    7      0    6    [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
2    7      1    6    [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
2    7      8    6    [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
2    7      9    6    [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
2    7      10   6    [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
2    7      11   6    [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
3    15     14   2    [14]
3    15     12   4    [12, 13]
3    15     13   4    [12, 13]
3    15     4    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3    15     5    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3    15     6    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3    15     7    6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3    15     10   6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
3    15     11   6    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Figure 14 Pre-consolidation table, MCF result

In the above example, note how for some VM replica copies, there is only 1 choice. This

is the case where the minimum cost flow solution had a PM send a replica copy to its edge-switch-neighbor PM, at a cost of 2 network hops. In a k=4 fat tree, each edge switch only holds 2 PMs: therefore, there can be no other equivalent cost choice for that replica copy (sending a copy to itself is never an option). In a k=8 fat tree, each edge switch holds 3 PMs, therefore the minimum number of choices would be 2 instead of 1.

As dictated by the minimum cost flow solution, the “medium length” (2 choices, in the above example) list of choices comes from choosing a destination PM in the same pod, but not the same edge-switch-neighbor, at a cost of 4 hops. The maximum length list of choices comes from being unable to choose an edge-switch-neighbor nor a pod-neighbor, being forced to route through a core switch at a cost of 6 hops.

Now that we have our list of minimum flow cost equivalent PM choices for each VM replica copy, we can use this as input to Khani et al.’s best consolidation heuristic and as input for our own optimal bin packing solution.

In order to use a linear solver for our optimal bin packing solution, we must define the variables and constraints, using the minimum cost flow solution to help us define our choices. Each VM copy will be an “item” to be packed in a “bin” (a PM that meets constraints). The “weight” of a VM copy will be its size, ensuring we do not overpack a PM with more VMs than it can hold.

Next we instantiate the solver and define the variables as follows: there are only two possible values concerning VM assignment: $x[i, j] = 1$ means VM copy i is packed in PM j. $x[i, j] = 0$ means VM copy i is not packed in PM j.

Similarly, y[j] = 1 means bin j is used. Y[j] = 0 means bin j is not used. “Used” means the PM holds at least 1 VM.

With the variables defined, we move on to defining constraints. Our first constraint is ensuring that each VM copy is in exactly one PM. In other words, for each item i, its sum in all bins must equal 1.

Our next constraint ensures we do not violate the space capacity of any PM. In other words, the sum of VMs in each PM (bin), must be less than or equal to that PM’s (bin’s) capacity. Recall that y[j] is binary: 0 if a bin is not used, or 1 if a bin is used. If y[j] were 0, the sum of sizes of all VMs packed in that bin can only be 0 as well. If y[j] were 1, then the sum of sizes of all VMs packed in that PM should be less than or equal to that PM’s capacity. Later, we will define our objective to have the solver minimize the number of PMs where y[j] is 1.

Next we have a more complex constraint: VM copies cannot share a PM with any other copies of that same VM (including the original). For example, this means a copy of VM #1 can never be held in the same PM which contains any other copy of VM #1. To help us define this constraint, we look at a simple example where we define that “item 1 cannot go in the same bin as item 2”:

```
for j in data['bins']:

    solver.Add((x[1,j] + x[2,j]) <= 1)
```

The inequality ensures that either item 1(1 + 0 <= 1), or item 2 (0 + 1 <= 1), or neither (0 + 0 <= 1), can go in any given bin, but any given bin will never contain both at once (the

left side of the inequality would equal 2, violating the constraint). Using this, we write a helper function to mutually exclude any 2 given items, which we call the mutex function.

Using the mutex function, we can then exclude each VM copy from every other copy of that same VM. For example, if we have 4 original VMs, with 10 total copies each, then we have 40 items. Copies of VM #0 will be assigned as items 0-9, copies of VM #1 will be 10-19, copies of VM #2 will be 20-29, and copies of VM #3 will be 30-39. This allows us to do integer division to figure out the originating VM id. 0 through 9 // 10 will all equal 0. 10 through 19 // 10 will all equal 1...

The final constraint involves limiting each VM copy to its minimum cost flow equivalent choices. Here are some examples of how to constrain an item to a specific set of bins:

```
#item 5 must go in bin 9

solver.Add(x[5,9] == 1)
```

Here is another example of how to constrain an item to 2 specific bins:

```
#item 5 must go in bins 9 or 10

solver.Add((x[5,9] + x[5,10]) == 1)
```

An example of constraining an item to 3 specific bins:

```
#item 5 must go in bins 8 or 9 or 10

solver.Add((x[5,8] + x[5,9] + x[5,10]) == 1)
```

We use the above examples to construct a string for each item, composed of its

minimum cost flow equivalent choice PMs. Once the string is constructed we can then use eval() to add it as an expression to the solver. Construction involves referencing the equivalent cost choices table from earlier.

With all our necessary constraints defined, we invoke the solver with the objective of minimizing the number of PMs (bins) used. With some manipulation of the solver's output, we can achieve a more human readable output indicating which PMs contain which VMs:

```
Optimal bin packing solution:
        constraints:
        PM capacity, replication, MCF equivalent cost choice

        PM # 0      stores VMs: [0, 1, 2, 3]
        PM # 1      stores VMs: [0, 1, 2, 3]
        PM # 4      stores VMs: [0, 1, 2, 3]
        PM # 5      stores VMs: [0, 1, 2, 3]
        PM # 6      stores VMs: [0, 1, 2, 3]
        PM # 7      stores VMs: [0, 1, 2, 3]
        PM # 12     stores VMs: [0, 1, 2, 3]
        PM # 13     stores VMs: [0, 1, 2, 3]
        PM # 14     stores VMs: [0, 1, 2, 3]
        PM # 15     stores VMs: [0, 1, 2, 3]

       Optimal Bin Packing calculation time =  364  milliseconds
```

Figure 15 Optimal bin packing output

With more output manipulation, we can construct a diagram similar to the one we used to view the output of the minimum cost flow solution, which used an arbitrary number of PMs:

```
PM-  | PM-space | PM-max-   | Assigned-
ID   | used     | capacity  | VM(s)
-------------------------------------------------
 0       4          30          0    1    2    3
 1       4          30          0    1    2    3
 2       0          30
 3       0          30
 4       4          30          0    1    2    3
 5       4          30          0    1    2    3
 6       4          30          0    1    2    3
 7       4          30          0    1    2    3
 8       0          30
 9       0          30
 10      0          30
 11      0          30
 12      4          30          0    1    2    3
 13      4          30          0    1    2    3
 14      4          30          0    1    2    3
 15      4          30          0    1    2    3

14 PMs used by MCF
14 replacement firstfit consolidation method
12 firstFit consolidation method
11 original consolidation method
10 linear solver method
```

Figure 16 Visualized bin packing output

In the above example, the optimal bin packing method produced by the linear solver has used 10 PMs in its solution, as opposed to 11 by Khani et al.'s best heuristic, and 14 as arbitrarily done by the minimum cost flow solution. Substantial consolidation improvement over the best heuristic is somewhat rare, but the linear solution is guaranteed to be optimal.

Chapter 5: RESULTS

FirstFit (Algorithm 1) is just a benchmark, what we are interested in the flow stage is comparing greedy (Algorithm 2) to optimal (Algorithm 3 MCF), to show that optimal always matches or outperforms greedy. We can see as the number of VMs increases, there is a greater difference in flow costs between the greedy flow heuristic (page 33 for details) and the optimal minimum cost flow. This verifies Khani et al.'s work. The FirstFit flow heuristic (see page 32 for details) will always have an extremely high flow cost because it disregards flow cost in its solution.

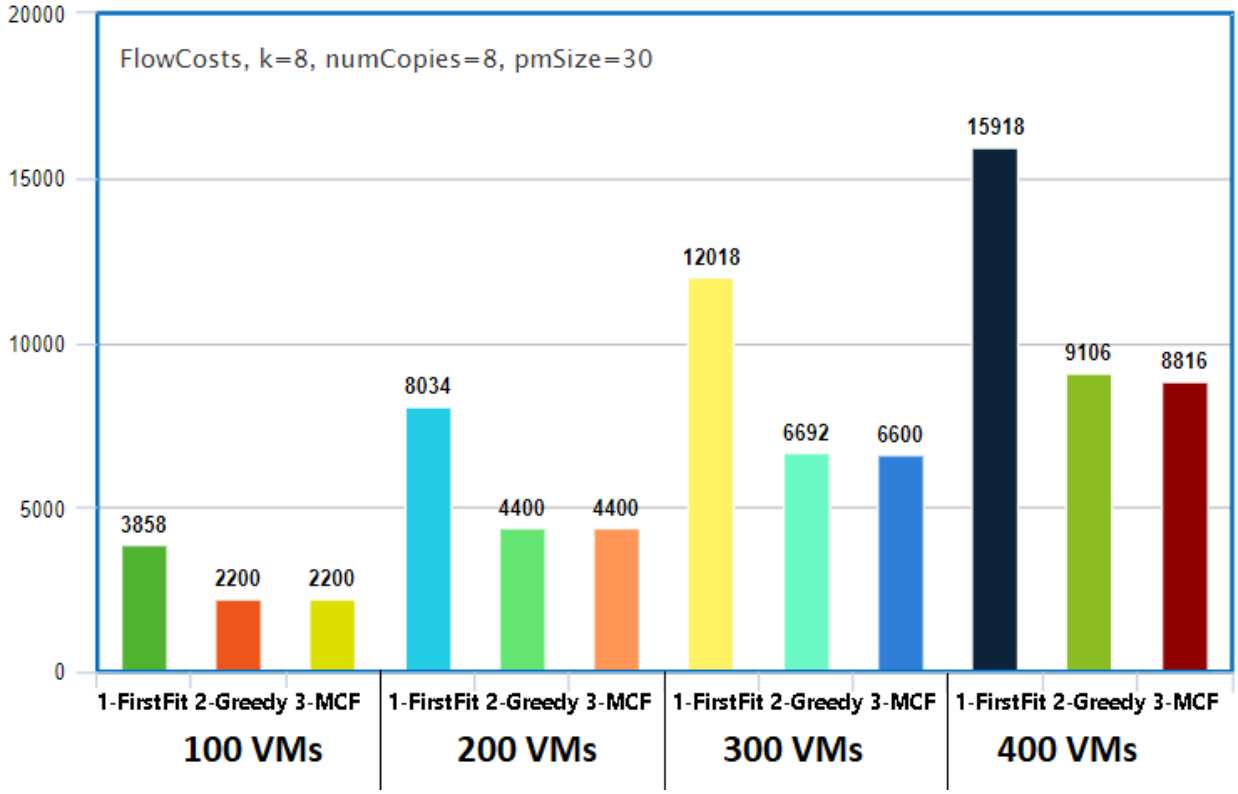

Figure 17 Flow cost results, variable VMs

We can also see that as the number of replica copies increases, there is again greater differentiation between the best flow heuristic and the optimal solution:

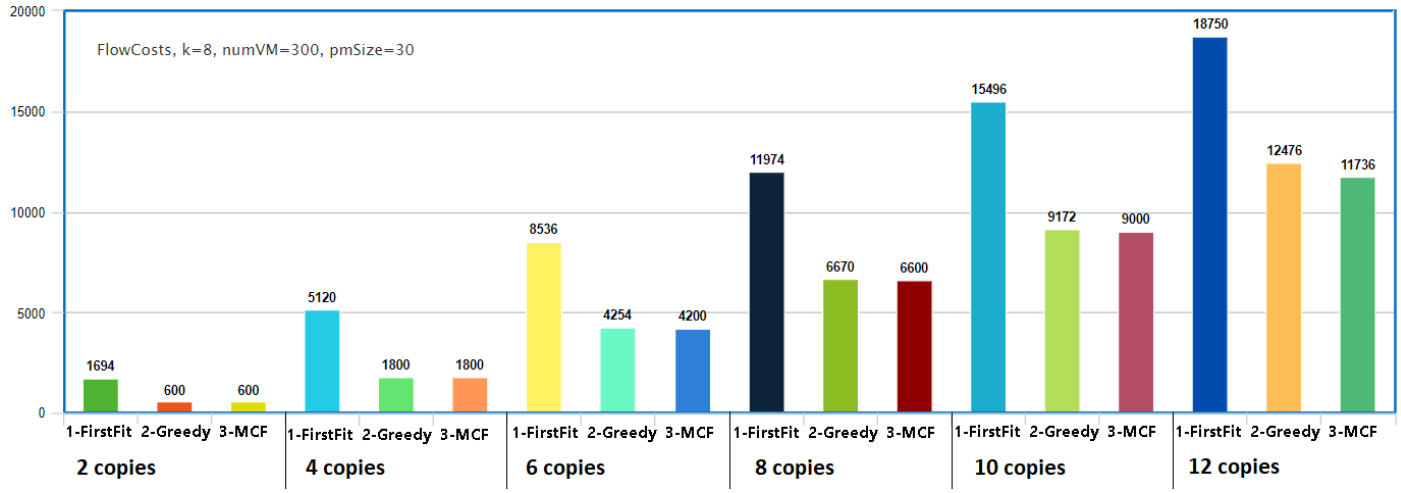

Figure 18 Flow cost results, variable copies

Rather than cherry-pick examples where optimal consolidation outperforms the heuristics, we use the same settings we tested the flow algorithms with:

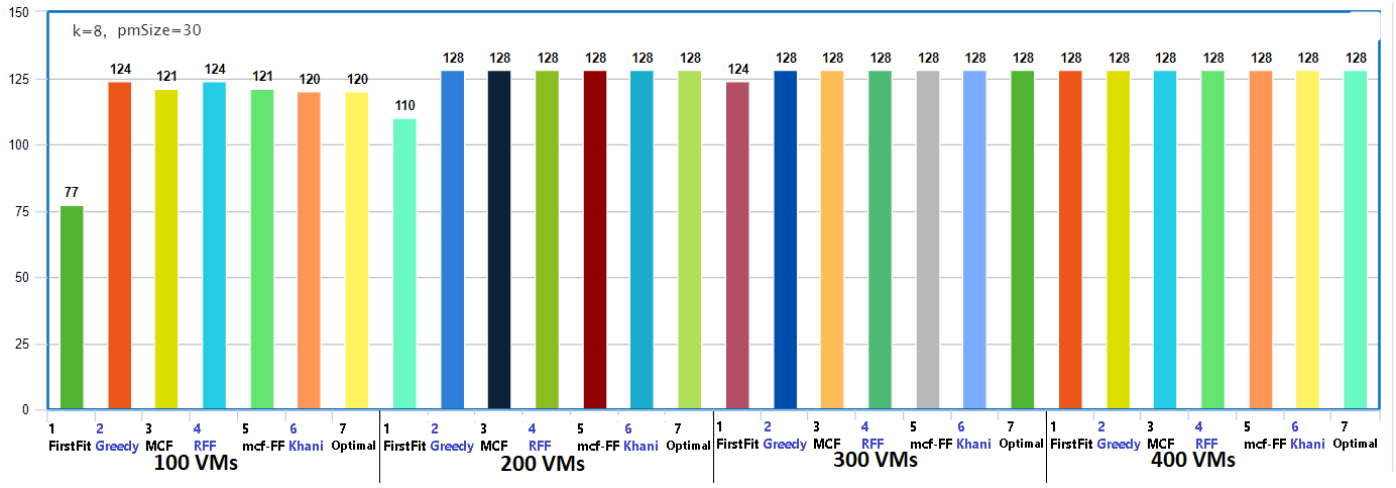

Figure 19 Consolidation results, variable VMs

The number of active PMs do not normally show much differentiation between the consolidation algorithms except under unusually specific conditions, discussed later. The only exception being that FirstFit ignores flow cost, thus it deviates significantly from the number of active PMs used by other algorithms, which are restricted by

considering flow cost. To clarify, the FirstFit benchmark is not outperforming the optimal, it is simply not subject to the constraint of considering flow-cost.

It is clear that an optimal algorithm will always match or outperform a heuristic, but the question is how exactly does optimal outperform a greedy heuristic in this problem? Consider the following example:

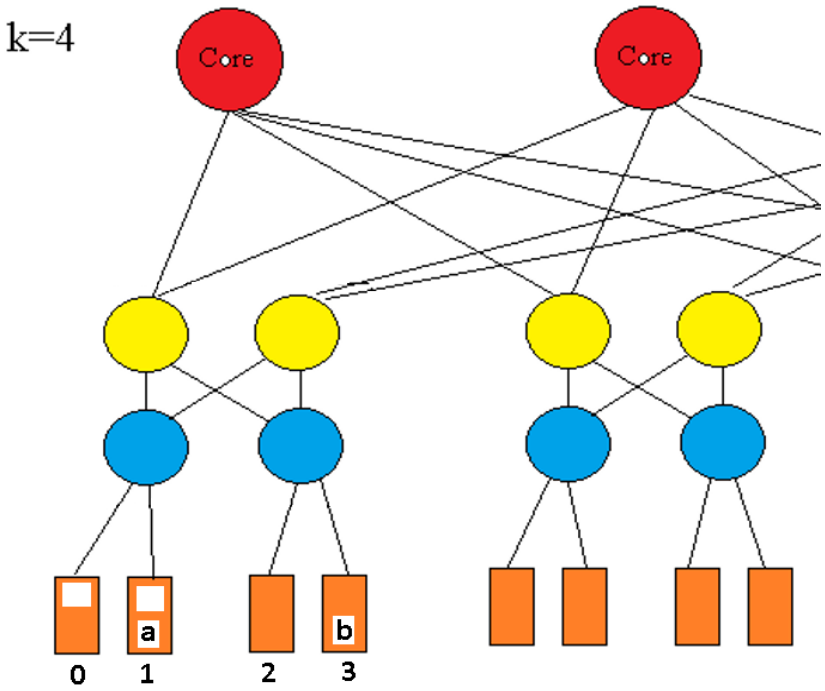

Figure 20 Example, optimal outperforms greedy

VM “b” can place a copy on either PM 0 or PM 1 for a flow-cost of 4 ‘hops’. However, VM “a” has a more important choice: place a copy of VM “a” on PM 0 for a cost of 2, or be forced to pay more to replicate elsewhere. If VM “b” is allowed to choose first, and arbitrarily chooses PM 0 instead of PM 1, this blocks VM “a” from making its optimal choice; this is an example of how the greedy heuristics often fail, they allow arbitrary order of choice and have no mechanism for backtracking. This type of heuristic mistake

happens more often under “stressful conditions”.

What is meant by “stressful” conditions? As stated before, there is a sweet spot in parameter stress where optimal notably outperforms greedy, rather than simply matching it. With low numbers of copies, or lots of extra PM space compared to the total size of VMs, it is easy to find a solution which satisfies the constraints, there will not be significant differentiation between optimal and greedy. There will not be many places for the greedy algorithm to make a mistake, like in the previous example with Figure 20. Likewise, under very stressful conditions, such as where the PMs are filled nearly to capacity for any possible solution, there will not be much differentiation between greedy and optimal, as the difficulty in satisfying the constraints alone leaves almost no room for ‘intelligent’ decision making, because there are a very limited number of possible solutions which satisfy the constraints.

What about the sweet spot of moderately stressful conditions? The following settings where k=4, 4 original VMs, 10 total copies of each, PM size of 10, are considered moderately stressful, a higher number of VM copies approaches the limit as defined by the replication constraint; with k=4, there are only 16 PMs, at least 10 different PMs are required to hold all 10 the copies of any given VM, even before considering flow cost and space capacity.

We now look at an example where optimal Algorithm 7 outperformed the Algorithm 6 consolidation heuristic. The minimum cost flow solution produces a solution using 14 servers:

```
Minimum Cost Flow Solution: Pre-Consolidation
Minimum cost: 184

PM-  | PM-space | PM-max-   | Assigned-
ID   | used     | capacity | VM(s)
-----------------------------------------------
 0       1         30          2
 1       1         30          2
 2       0         30
 3       0         30
 4       2         30          2    3
 5       4         30          2    0    1    3
 6       4         30          2    0    1    3
 7       4         30          2    0    1    3
 8       3         30          2    0    1
 9       3         30          2    0    1
 10      3         30          2    0    3
 11      3         30          2    1    3
 12      3         30          0    1    3
 13      3         30          1    0    3
 14      3         30          0    1    3
 15      3         30          3    0    1
```

Figure 21 Sample MCF output, 14 servers used

Algorithm 6 reduces the number of servers from 14 to 11:

```
pm 0 consolidated
pm 1 consolidated
pm 8 consolidated

PM-  | PM-space | PM-max-   | Assigned-
ID   | used     | capacity | VM(s)
-----------------------------------------------
 0       0         30
 1       0         30
 2       0         30
 3       0         30
 4       4         30          2    3    0    1
 5       4         30          2    0    1    3
 6       4         30          2    0    1    3
 7       4         30          2    0    1    3
 8       0         30
 9       3         30          2    0    1
 10      3         30          2    0    3
 11      3         30          2    1    3
 12      4         30          0    1    3    2
 13      4         30          1    0    3    2
 14      4         30          0    1    3    2
 15      3         30          3    0    1

numvms before,after 40 40
active pms before,after 14 11
```

Figure 22 Sample Khani output, 11 servers used

The optimal solution saves 1 additional server over the best heuristic:

```
        Optimal Bin Packing calculation time =  390  milliseconds

PM-  | PM-space | PM-max-  | Assigned-
ID   | used     | capacity | VM(s)
-------------------------------------------------
 0       4         30          0   1   2   3
 1       4         30          0   1   2   3
 2       0         30
 3       0         30
 4       4         30          0   1   2   3
 5       4         30          0   1   2   3
 6       4         30          0   1   2   3
 7       4         30          0   1   2   3
 8       0         30
 9       0         30
 10      0         30
 11      0         30
 12      4         30          0   1   2   3
 13      4         30          0   1   2   3
 14      4         30          0   1   2   3
 15      4         30          0   1   2   3

14 PMs used by MCF
14 replacement firstfit consolidation method
12 firstFit consolidation method
11 original consolidation method
10 linear solver method
```

Figure 23 Sample optimal output, 10 servers used

Even under “moderately stressful” conditions, improvement is somewhat uncommon:

```
                    number of runs: 1000
    optimal matches best heuristic: 966
optimal outperforms best heuristic: 34 times
   avg servers used,best heuristic: 12.23
       avg servers used in optimal: 12.19
  average magnitude of improvement: 1.09 additional servers
```

Figure 24 Summary results from 1000 runs

Out of 1000 runs under those same settings, k=4, 4 VMs, 10 copies each, 10 PM size, optimal outperformed the best heuristic 34 times.

On average, the best heuristic used 12.23 servers, whereas optimal used 12.19 servers in a flow-cost-equivalent solution. When there was improvement to be found, the magnitude of improvement averaged an additional 1.09 servers consolidated.

Chapter 6: FUTURE DIRECTIONS

Rather than first optimizing flow and then optimizing consolidation in a secondary calculation, it is possible to avoid a consolidation stage altogether, simply by adding in a server activation cost to the initial model before flow cost is optimized.

Recall the initial graph transformation, as specified by Khani et al.:

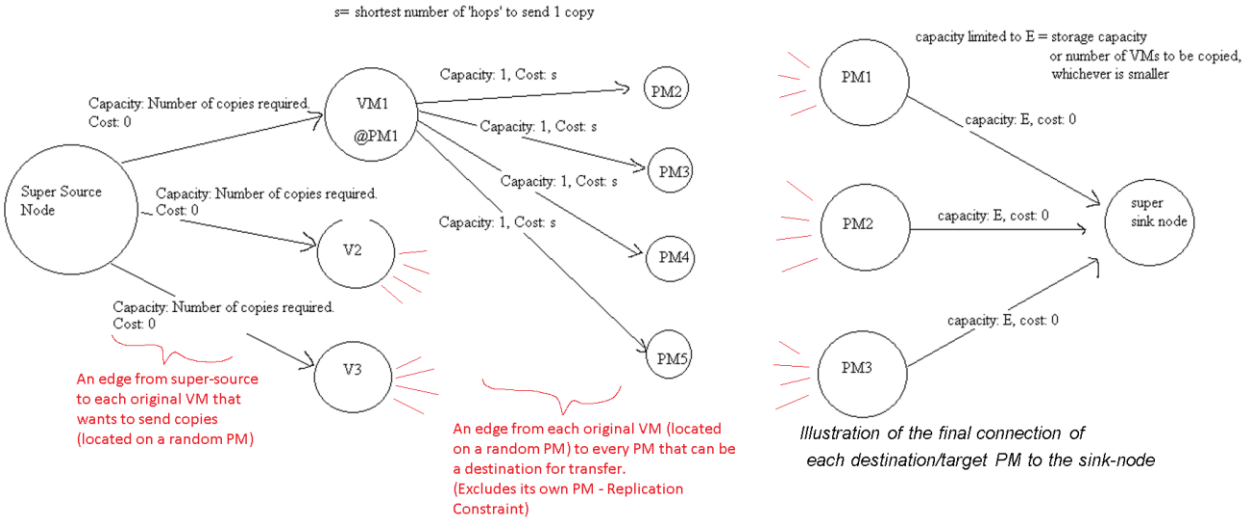

Figure 25 Review of Khani et. al original graph transformation

Note that the final set of edges connecting each potential destination PM to the super sink have a cost of 0. This is the reason why the minimum cost flow solution chooses an arbitrary number of servers. If this cost was changed to represent a one-time cost, or constant cost, of activating a server, there would be no need for a separate consolidation phase, as the model could be solved for both flow and consolidation at the same time:

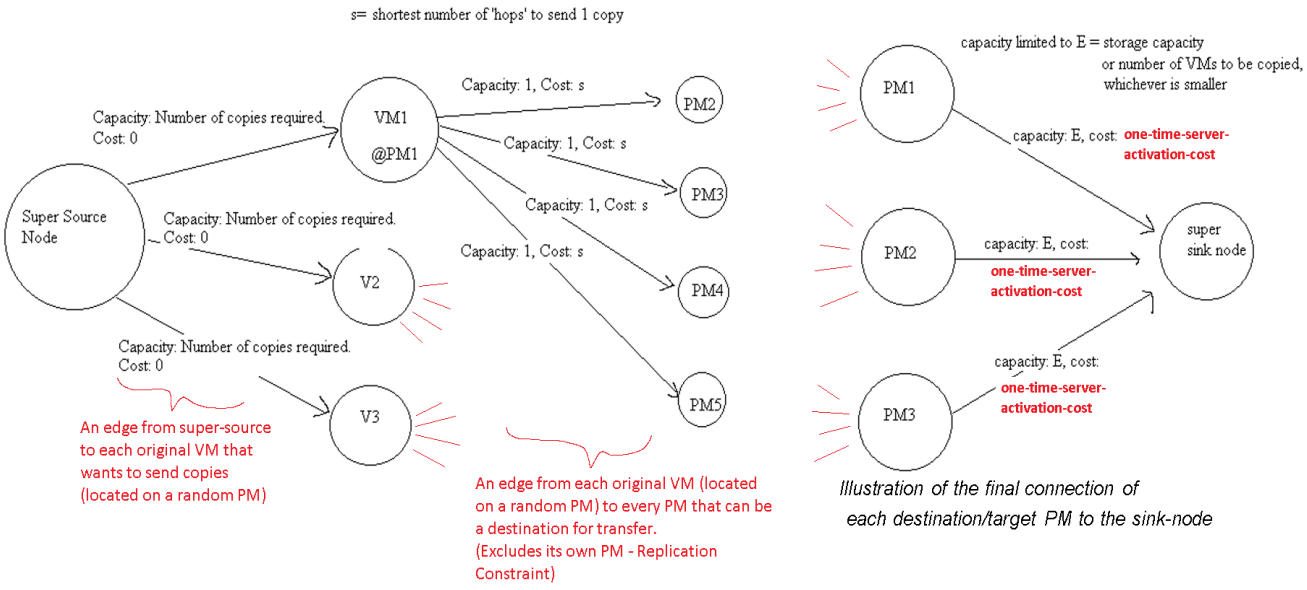

Figure 26 Modified graph transformation, single stage optimization

However, this cannot be done without understanding the difference between representing a typical flow-cost vs. a one-time cost in the flow balance equations. In order to plug this into a linear solver, the equations would have to be adjusted to represent an additional one-time, or constant, cost on the final set of edges, not simply a flow-cost.

Sensitivity analysis is needed. Adding in a one-time or constant cost to represent the energy required to turn on a server also presents some additional complexity: what if the energy of an active but empty server is much greater in magnitude than the flow cost? Then an optimal solution would seek to prioritize server consolidation over flow-cost. How much energy would we save in each case? What if we approach the problem asynchronously, more realistically, rather than assuming all VMs are being copied and transferred at once? These types of questions can only be answered by a sensitivity analysis using real world data centers, to determine what the most important factors in

data center energy savings really are.

In a more realistic scenario, there are certainly factors outside of flow and consolidation to consider, including nonlinear considerations as well. To this end, Google has given its artificial intelligence “DeepMind” administrative level access to its data centers in order to optimize energy costs, claiming an energy savings of 40%, although the analysis as to how and the details as to what factors it prioritizes are still unknown [1]. The machine learning approach seems able to adjust for factors that may be nonlinear in nature.

REFERENCES

[1] “DeepMind reduces cooling bill by 40%” https://deepmind.com/blog/article/deepmind-ai-reduces-google-data-centre-cooling-bill-40

[2] H. Goudarzi and M. Pedram, "Energy-Efficient Virtual Machine Replication and Placement in a Cloud Computing System," 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, 2012, pp. 750-757.

[3] https://developers.google.com/optimization/install

[4] https://developers.google.com/optimization/lp/lp

[5] https://developers.google.com/optimization/introduction/python

[6] https://news.stanford.edu/news/2005/may25/dantzigobit-052505.html

[7] 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.

[8] Schaus, Regin, Derval, “Improved Filtering for the Bin-Packing with Cardinality Constraint”. Available at https://www.info.ucl.ac.be/~pschaus/assets/publi/constraints-cp17-binpacking.pdf, checked 7/15/2020.

[9] X. Dai, J. M. Wang and B. Bensaou, "Energy-Efficient Virtual Machines Scheduling in Multi-Tenant Data Centers," in IEEE Transactions on Cloud Computing, vol. 4, no. 2, pp. 210-221, 1 April-June 2016.

[10] Zeghlache, Hadji. “Minimum Cost Maximum Flow Algorithm for Dynamic Resource Allocation in Clouds”. Available at http://makhlouf.hadji.free.fr/Publications/Dynamic_resources-Hadji.pdf, checked 7/15/2020.

APPENDIX

Code requires python 3.6+, 64 bit system, Google OR-tools to run. The fattree.py version of the program prints out visualizations of the PM states to the screen, use this for k=4 simulations when you want to see what is happening. The fattree_writeonly.py version has no screen printing, it writes a summary of all results to fattreeresults.txt. Both versions read in the simulation settings from fattreesettings.txt, which looks like:

```
k 4
numVM 4
minVMsize 1
maxVMsize 1
numCopies 5
minPMsize 30
maxPMsize 30
numRuns 20
```

You can change the integer values to whatever you wish (k should be a positive even integer). For small simulations (k=4, 4 VMs, 10 copies) it takes less than 1 second per run. It gets slow (10 minutes) at high settings, such as 300+ VMs, 10+ copies each on k=8 fat tree due to a lack of optimization on the data structures storing PM and VM data.

Link to full code folder:

https://drive.google.com/drive/folders/1D4qHIxmuf4c30e9dy9mTLX1mWwzRrtxc?usp=sharing

Copy of code for fattree.py (screen printing version):

```
import random #used to randomize initial pm and vm placement
from copy import deepcopy #used for saving pm states before each algorithm
from ortools.graph import pywrapgraph #used for mcf optimal solver
from ortools.linear_solver import pywraplp #used for bin packing optimal solver
#import time #used to time algorithms
import sys

def calcPMtoPMcost(pm1,pm2):
    pm1=int(pm1)
    pm2=int(pm2)
    if pm1 == pm2: #pm to itself costs 0
        return 0
    pmPerEdge=int(numPM / totalEdge)
    pm1Edge=int(pm1 / pmPerEdge)
    pm2Edge=int(pm2 / pmPerEdge)
    if pm1Edge == pm2Edge: #pms on same edge switch costs 2
        return 2
```

```
    pm1Pod=int(pm1Edge / numEdgePerPod)
    pm2Pod=int(pm2Edge / numEdgePerPod)
    if pm1Pod == pm2Pod: #pms on same pod costs 4
        return 4
    return 6 #routing through core switch costs 6

with open('fattreesettings.txt','r') as f:
    settings=[]
    for line in f:
        line=line.strip().split()
        settings.append(line)

#read in settings saved from txt file
k=          int(settings[0][1])
numVM=      int(settings[1][1])
minVMsize=  int(settings[2][1])
maxVMsize=  int(settings[3][1])
numCopies=  int(settings[4][1]) #replica + original
minPMsize=  int(settings[5][1])
maxPMsize=  int(settings[6][1])
numRuns=    int(settings[7][1])
#how many runs to simulate,
#only used in write-only version for data collection



#calculate number of switches and edges
numPod=int(k)
numCore=int((k/2)**2)
numAggPerPod=int(k/2)
numEdgePerPod=int(k/2)
totalAgg=numAggPerPod * numPod
totalEdge=numEdgePerPod * numPod
numPM=int((k**3) / 4)
totalSwitches=int(numCore + totalAgg + totalEdge)
totalNode=totalSwitches + numPM
totalEdges=numPM * 3


def printl(label,variable):
    print('%30s:' %(str(label)),variable)

printl('Core switches',numCore)
printl('Number of pods',numPod)
printl('Aggregation switches per pod',numAggPerPod)
printl('Edge switches per pod',numEdgePerPod)
printl('Number of Physical Machines',numPM)
printl('Total aggregate switches',totalAgg)
printl('Total edge switches',totalEdge)
```

```
printl('Total Switches',totalSwitches)
printl('Total Switches + PMs',totalNode)
printl('Total Edges',totalEdges)


#assign ID to each switch and PM
v= [x for x in range(totalNode)]
vc=[x for x in range(totalNode - numCore,totalNode)]
va=[x for x in range(numPM + totalEdge, totalNode - numCore)]
ve=[x for x in range(numPM, totalNode - numCore - totalEdge)]
pmDict={}
#pmDict has [ [pmID, space-used, max-space] ,
#[containedVM1-id, vm-size, originatingPM-id], [containedVM2-id...] ]

totalPMCapacity = 0
for x in range(numPM):
    randomPMsize = random.randint(minPMsize,maxPMsize)
    totalPMCapacity+=randomPMsize
    pmDict[x] = [x,0,randomPMsize]

def printv(label,vlist):
    print('%30s:' %(str(label)),vlist[0],'...',vlist[-1])

print()
print('%37s'%('PM IDs: 0 ...'),numPM-1)
printv('PMs and all switches IDs', v)
printv('Core switch IDs', vc)
printv('Aggregation switch IDs',va)
printv('Edge switch IDs',ve)
print()

#creating cost dictionary for later reference
cost_dict={}
for x in range(numPM):
    for y in range(x):
        costkey=(x,y)
        costvalue=calcPMtoPMcost(x,y)
        cost_dict[costkey]=costvalue
        reversecostkey = (y,x) #reverse pair has same cost
        cost_dict[reversecostkey]=costvalue
for x in range(numPM): #pm to itself is 0
    costkey = (x,x)
    cost_dict[costkey] = 0

vmList = []
totalOrigVMsize = 0
totalOrigPlusReplicaSize = 0
vmDict = {}
#key: vmID
```

```
#value: [vm-id, vm-size, originating-PM-id]

for x in range(numVM):
    randomVMsize = random.randint(minVMsize,maxVMsize)
    totalOrigVMsize += randomVMsize
    totalOrigPlusReplicaSize += (randomVMsize * numCopies)

    #find a PM to place this VM on
    placed = False
    allPMids = [y for y in range(numPM)]
    #random.shuffle(allPMids) #not necessary to shuffle since we pop randomly
    for p in range(len(allPMids)):
        randomPM = allPMids.pop(random.randint(0,len(allPMids)-1))
        randomPM = pmDict[randomPM]
        randomPMspaceremaining = randomPM[2] - randomPM[1] #max - used
        if randomPMspaceremaining >= randomVMsize:
            randomPMid = randomPM[0]
            placed = True
            thisvm = [x,randomVMsize,randomPMid]
            vmList.append(thisvm) #add to vm list
            vmDict[x]=thisvm #add to vm dict
            pmDict[randomPMid].append(thisvm)
            pmDict[randomPMid][1] += randomVMsize #update pm used space
            break
    if placed == False: #never placed the VM
        print('warning, VM',x,' could not be placed')
        sys.exit()

replicaSize = totalOrigPlusReplicaSize - totalOrigVMsize

print('VM-ID, VM-Size, PM-location')
for x in vmList:
    print('%5s %6s %5s' %(str(x[0]),str(x[1]),str(x[2])))

pmList= []
for k,v in pmDict.items():
    pmList.append(v)

def printPM(pmList):
    print('\nPM- | PM-space | PM-max-  | Assigned-')
    print('ID  | used     | capacity | VM(s)')
    print('-'*50)
    for pm in pmList:
        print(' %-5d  %-8d %-10d'%(pm[0],pm[1],pm[2]),end=' ')
        for vm in pm[3:]:
            print('%-3d'%(vm[0]),end=' ')
        print()
    print()
```

```
print('\nInitial, random original VM placement')
printPM(pmList)
######################## nonrandom pmlist and vmlist for testing purposes
#pmList = [[0, 0, 30], [1, 0, 30], [2, 0, 30], [3, 0, 30], [4, 0, 30], [5, 0, 30], [6, 0, 30], [7, 1, 30, [2, 1, 7]], [8, 0, 30], [9, 0, 30], [10, 0, 30], [11, 0, 30], [12, 1, 30, [0, 1, 12]], [13, 1, 30, [1, 1, 13]], [14, 0, 30], [15, 1, 30, [3, 1, 15]]]
#vmList = [[0, 1, 12], [1, 1, 13], [2, 1, 7], [3, 1, 15]]
########################



#first fit flow heuristic. in arbitrary order, each VM replica copy chooses the lowest ID available PM. flow cost ignored.
firstFitDict = deepcopy(pmDict)
firstFit_pmList=deepcopy(pmList) #deep copy pmList for different solutions
firstFitFlowCost = 0

def pmXcontainsvmY(pmx,vmy):
    for x in firstFitDict[pmx[0]][3:]:
        if x[0] == vmy[0]:
            return True
    return False

replicaCopies=numCopies-1
if replicaCopies > 0:
    for vm in vmList:
        needtoplace = replicaCopies
        placed = 0
        transferCost = 0
        for pm in firstFit_pmList:
            if placed==replicaCopies: #already placed enough replica copies
                break
            if pmXcontainsvmY(pm,vm):
                continue
            pmvalues = firstFitDict[pm[0]]
            if pmvalues[2] - pmvalues[1] >= vm[1]: #pm-max - pm-used >= vm-size
                needtoplace-=1
                placed+=1
                pmvalues[1] += vm[1] #update space used
                pmvalues.append(vm)
                firstFitDict[pm[0]] = deepcopy(pmvalues)
                costkey = (pm[0],vm[2])
                transferCost += (cost_dict[costkey] * vm[1])
        if needtoplace != 0:
            print('warning: didnt place all copies of VM',vm[0])
        firstFitFlowCost+=transferCost

firstfitPMlist = []
firstfit_pms_used = 0
```

```
for k,v in firstFitDict.items():
    firstfitPMlist.append(v)
    if len(v) > 3:
        firstfit_pms_used+=1
print('\nFirstFit, flowcost:',firstFitFlowCost,' num-active-PMs:',firstfit_pms_used)
printPM(firstfitPMlist)

#greedy: each vm seeks its lowest flow-cost available PM
greedy_pmList=deepcopy(pmList)
greedy_pmDict=deepcopy(pmDict)
greedyFlowCost=0

def greedycheckPMcontainsVM(pmx,vmy):
    for x in greedy_pmDict[pmx[0]][3:]:
        if x[0] == vmy[0]:
            return True
    return False

greedyFlowCost=0
for v in vmList:
    for y in range (replicaCopies):
        choices=[]
        for pm in greedy_pmList:#calc flow cost from origPM to all other PMs
            #pmID=pm[0]
            #origPM=v[2]
            costkey = (v[2],pm[0])
            #cost = cost_dict[costkey]
            choice_tuple=(pm[0],cost_dict[costkey])
            choices.append(choice_tuple)

        choices=sorted(choices, key = lambda x: x[1]) #sort by cost of choices, lowest cost first
        transferCost=0
        for choice in choices:
            #pmID=choice[0], vmID=v[0]
            if greedycheckPMcontainsVM(choice,v):
                continue #if pm contains this vm already
            pmvalues = greedy_pmDict[choice[0]]
            #vmSize=v[1]
            if pmvalues[2] - pmvalues[1] >= v[1]: #pm-max - pm-used >= vm-size
                pmvalues[1] += v[1] #update space used
                pmvalues.append(v)
                greedy_pmDict[choice[0]] = deepcopy(pmvalues)
                costkey = (choice[0],v[2])
                transferCost += (cost_dict[costkey] * v[1])
                break #stop trying to place in more choices PMs
        greedyFlowCost+=transferCost

greedy_pmList = []
greedy_pms_used = 0
```

```
for k,v in greedy_pmDict.items():
    greedy_pmList.append(v)
    if len(v) > 3:
        greedy_pms_used+=1
print('\nGreedy, flowcost:',greedyFlowCost,' num-active-PMs:',greedy_pms_used)
printPM(greedy_pmList)

#Minimum Cost Flow Solution
mcf_pmDict=deepcopy(pmDict)
mcf_vmDict=deepcopy(vmDict)

#Instantiate an OR-Tools SimpleMinCostFlow solver
min_cost_flow = pywrapgraph.SimpleMinCostFlow()
print('Minimum Cost Flow Solution: Pre-Consolidation')

superSourceID = numVM
superSinkID = numVM+1
offset = numVM+2 #PM ids will be offset by this number, to avoid conflict

superSourceSupply=replicaSize #set source supply to sum of replica sizes
min_cost_flow.SetNodeSupply(superSourceID,superSourceSupply)

superSinkDemand=superSourceSupply * -1 #set sink demand
min_cost_flow.SetNodeSupply(superSinkID,superSinkDemand)

#generate arcs from super source to each orig vm
#min_cost_flow.AddArcWithCapacityAndUnitCost parameters are
#                                (start-node, end-node, capacity, unit-cost)
finalsetdict={}
for k in range(numVM):
    thisvmsize = mcf_vmDict[k][1] #get size of this vm
    #add arcs from super-source to each original VM
    min_cost_flow.AddArcWithCapacityAndUnitCost(superSourceID,k,replicaCopies*thisvmsize,0)
    thisvmorigPM = mcf_vmDict[k][2]
    for pmID in mcf_pmDict.keys():
        if pmID == thisvmorigPM: #dont create arc to originating-PM
            continue
        costkey = (thisvmorigPM,pmID)
        cost = cost_dict[costkey]
        #add arcs from each orig-VM to potential-destination-PMs
        newPMid = pmID+offset
        min_cost_flow.AddArcWithCapacityAndUnitCost(k,newPMid,thisvmsize,cost)
        if newPMid not in finalsetdict:
            finalsetdict[newPMid] = 0 #value doesnt matter here, storing the key does
            pm_remaining_space = mcf_pmDict[pmID][2] - mcf_pmDict[pmID][1]#maxSpace - usedSpace
            #generate arcs from each potential-destination-PM to the super sink
            min_cost_flow.AddArcWithCapacityAndUnitCost(newPMid,superSinkID,pm_remaining_space,0)

#vm_table holds information about MCF equivalent cost choices for each VM
```

```
if min_cost_flow.Solve() == min_cost_flow.OPTIMAL:
    vm_table = []
    print('Minimum cost:', min_cost_flow.OptimalCost(),end='  ')
    copyid=0
    for i in range(min_cost_flow.NumArcs()):
        cost = min_cost_flow.Flow(i) * min_cost_flow.UnitCost(i)
        if min_cost_flow.Flow(i) > 0 and cost > 0:

#a,b,c,d,e=min_cost_flow.Tail(i),min_cost_flow.Head(i),min_cost_flow.Flow(i),min_cost_flow.Capacity(i),c
ost
            origVMid,destinationPMid=min_cost_flow.Tail(i),min_cost_flow.Head(i)
            destinationPMid -= offset
            #vm_table will have structure of: [copyID, origVMid, orig PM, destination PM, sunk cost]
            vm_table.append([copyid,origVMid,mcf_vmDict[origVMid][2],destinationPMid,cost])
            copyid+=1
            #update the pm dict to show a vm was replicated to the destination PM
            pmvalues = mcf_pmDict[destinationPMid]
            pmvalues.append(mcf_vmDict[origVMid])
            pmvalues[1] += mcf_vmDict[origVMid][1] #update pm-space-used
            mcf_pmDict[destinationPMid] = deepcopy(pmvalues)

else:
    print('Error: There was an issue with the min cost flow input.')


mcf_pmList=[]
numMCFActivePM = 0
for v in mcf_pmDict.values():
    if len(v) > 3:
        numMCFActivePM+=1#tally up active PMs
    mcf_pmList.append(v)
print('num-active-PMs:',numMCFActivePM)
printPM(mcf_pmList)

#EQUIVALENT-COST-DESTINATION-PM-CHOICE TABLE HERE
print('replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs')
for vm in vm_table:
    #filter the cost_dict for equivalent cost choices, using origPM and cost
    #append the list of equivalent-cost-choice-PMs to each replicaVM
    #origpm,cost = vm[2],vm[4]
    filtered_list = [k[1] for k,v in cost_dict.items() if (k[0] == vm[2]) and v == vm[4]]
    vm.append(filtered_list)

    #print out the replica-vm-table
    for y in vm:
        print(y,'\t',end='  ')
    print()
```

```
rffPMdict = deepcopy(pmDict)

#Replacement first fit, consolidation heuristic
print('\nReplacement First Fit - MCF cost considered, MCF placement disregarded')
#acts as though all vms are unplaced but mcf cost is known, then first fit places them according to
equivalent mcf cost per copy

def RFFcheckPMcontainsVM(pmID,vmID):
    for vm in rffPMdict[pmID][3:]:
        if vm[0] == vmID:
            return True
    return False

for vm in vm_table:
    for choice in vm[5]:
        #check if pm contains this vm already. replication constraint.
        vmID = vm[1]
        #pmID = choice
        if RFFcheckPMcontainsVM(choice,vmID):
            continue
        #pm-space-max  = rffPMdict[pmID][2]
        #pm-space-used = rffPMdict[pmID][1]
        #pm-space-remaining = max - used
        if (rffPMdict[choice][2] - rffPMdict[choice][1]) >= vmDict[vmID][1]:#pm has enough space, we can
place
            pmvalues = rffPMdict[choice]
            vmSize = vmDict[vmID][1]#vmID = vm[1]
            addVM = [vmID,vmSize,vm[2]]#vmOrigPM = vm[2] or vmDict[vmID][2]
            pmvalues[1] += vmSize
            pmvalues.append(addVM)
            rffPMdict[choice] = deepcopy(pmvalues) #[x for x in pmvalues]
            break

#find active PMs and visualize rff solution
rff_pmList=[]
numRFFactivePM = 0
for v in rffPMdict.values():
    if len(v) > 3:
        numRFFactivePM+=1#tally up active PMs
    rff_pmList.append(v)
print('num-active-PMs:',numRFFactivePM)
printPM(rff_pmList)

#First fit consolidation heuristic
#Keeps mcf placement, tries to firstfit consolidate from there, using mcf-equivalent-cost for each copy
#will only transfer if destination-PM is not empty

print('\nFirst Fit Consolidation - from MCF solution, \nEach VM jumps to a same-cost, lowest-id PM')
```

```
#before transfer dictionary
ffnotemptyDict = {k:v for k,v in mcf_pmDict.items()}

def FFcheckPMcontainsVM(pmID,vmID):
    for vm in ffnotemptyDict[pmID][3:]:
        if vm[0] == vmID:
            return True
    return False

#print('replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs')
for vm in vm_table:
    for choice in vm[5]:
        vmID = vm[1]
        #pmID = choice
        if len(ffnotemptyDict[choice]) == 3: #pm is empty, dont transfer
            continue
        if choice == vm[3]:
            continue #dest-pm is same as mcf solution, dont transfer
        if FFcheckPMcontainsVM(choice,vmID):
            continue #dest-pm contains a copy of this vm already, dont transfer
        #pm-space-max  = ffnotemptyDict[pmID][2]
        #pm-space-used = ffnotemptyDict[pmID][1]
        #pm-space-remaining = max - used
        if (ffnotemptyDict[choice][2] - ffnotemptyDict[choice][1]) >= vmDict[vmID][1]:
            #pm has enough space, we can transfer
            pmvalues = [x for x in ffnotemptyDict[choice]]
            vmSize = vmDict[vmID][1]#vmID = vm[1]
            #add vm to chosen dest PM
            addVM = [vmID,vmSize,vm[2]]#vmOrigPM = vm[2] or vmDict[vmID][2]
            pmvalues[1] += vmSize
            pmvalues.append(addVM)
            ffnotemptyDict[choice] = [x for x in pmvalues] #[x for x in pmvalues]

            #remove vm from previous dest PM
            ffnotemptyDict[vm[3]] = [x for x in ffnotemptyDict[vm[3]] if x != addVM]
            ffnotemptyDict[vm[3]][1] -= vmSize
            break

#find num-active-PMs and visualize ff solution
ff_pmList=[]
numFFactivePM = 0
howmanyvms = 0
for v in ffnotemptyDict.values():
    if len(v) > 3:
        numFFactivePM+=1#tally up active PMs
        for vm in v[3:]:
            howmanyvms+=1
    ff_pmList.append(v)
print('num-active-PMs:',numFFactivePM)
```

```
printPM(ff_pmList)

#implement original post-MCF consolidation as described in Dr. Tang's paper
#look at PMs with 1 VM in arbitrary order. for each PM, check if all its containing VMs can be moved
elsewhere to a set of TPM. if yes, move them all. if no, dont move any.
#repeat with PMs with 2 VMs, then PMs with 3 VMs, until you reach max possible VM #.

#start_time = time.time()
def printDict(anydict, label = '\nprinting some dict'):
    print(label)
    for k,v in anydict.items():
        print(k,v)
    print()
#copy mcf solution dict but sort by number of vms
#bestconsolidationheuristicDict = {k:v for k,v in sorted(mcf_pmDict.items(), key = lambda x: len(x[1]))}
#printDict(bestconsolidationheuristicDict)

print('Best consolidation heuristic, tries to move all PMs containing')
print('1 VM, then 2 VM, then 3 VM, ... only if all VMs can be moved')
#start_time = time.time()

#if PM contains original vm, cant move
#if destinationPM is VM's origPM, cant move
#cost-equivalent
#destinationPM cannot be empty
#check replica constraint
#check PMhasspace
#update space
#add vm to new pm
#remove vm from old pm
bestheurDict = {k:v for k,v in sorted(pmDict.items(), key = lambda x: len(x[1]))}
#exclude PMs which contain the original VMs
excludelist = [v[2] for v in vmList]
bestheurResult = {k:v for k,v in mcf_pmDict.items()} #final positions will be saved here

vm_tableDict = {} #this is the dict equivalent of vm_table

#build pm-dict using replica ids, not just orig vm ids, for lookup
for vm in vm_table:
    #add vm to dest pm
    #origvmSize = vmDict[vm[1]][1]
    #destpm = vm[3]
    pmvalues = bestheurDict[vm[3]]
    pmvalues.append(vm[0]) #add the replicaID to the destPM

    #since we're looping through the list, build a dict for later
    vm_tableDict[vm[0]] = [x for x in vm]
```

```
sortbynumcontainedVMs={}

for k,v in bestheurDict.items():
    if k in excludelist:
        #dont bother trying to consolidate PMs which contain-
        #an original vm
        #print('skipping',k)
        continue
    sortbynumcontainedVMs[k] = [x for x in v[3:]]

#print('replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs')
sortbynumcontainedVMs = sorted(sortbynumcontainedVMs, key = lambda
x:len(sortbynumcontainedVMs[x])) #sort PM-ids by number of contained vms
#print(sortbynumcontainedVMs)
#we try to consolidate starting from PMs containing the fewest replica VMs
#skipping PMs which contain no VMs
#skipping PMs which contain an original VM
#skipping PMs in which not all VMs can be moved
#valid alternate PM must be: not empty, same cost, contain no other copies of that vm, has enough space
for pmID in sortbynumcontainedVMs:
    pmvalues = [x for x in bestheurDict[pmID]] #trying to consolidate this PM
    #print('\npmvalues',pmvalues)
    #print('lenpmvalues3:',len(pmvalues[3:])) #num of replica copies on this pm
    #print('PMID',pmID)
    canmoveall = True
    movesmade = []
    #beforemove = {k:v for k,v in bestheurResult.items()} #save state before attempting moves
    for replica in pmvalues[3:]: #check if each replica can be moved. PMs that have no replicas are skipped
here.
        choicedata = [x for x in vm_tableDict[replica]]
        #choicedata is replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs
        #print('choicedata',choicedata)
        #we are looking at each replica copy's choices now
        canmove = False
        for choice in choicedata[5]: #these are the equiv-cost-PMs
            #print('choice',choice)
            #check that new dest is not same as old dest
            if choice == choicedata[3]: #cant move, new pm is same as old pm
                #print('dest pm is same as old dest', choice, choicedata[3])
                continue

            tpmvalues = [x for x in bestheurResult[choice]] #makes a copy instead of a reference to targetPM
            #print('tpmvalues',tpmvalues)
            #check that dest pm is not empty. we wont transfer to an empty pm
            if len(tpmvalues) == 3:
                #print(tpmvalues,'is empty, cant move')
                continue

            #check if alternate pm choice contains any copy of this vm
```

```
            containsflag = False
            for containedvm in tpmvalues[3:]:
                if containedvm[0] == choicedata[1]: #found match for this vm, cant move
                    #print('vm match, cant move', containedvm[0],choicedata[1])
                    containsflag = True
                    break
            if containsflag == True:
                continue #skip to next possible choice, dont place here.

            #check if alternate choice has space
            #pmMax - pmUsed >= originatingVMsize
            #pmMax  = tpmvalues[2]
            #pmUsed = tpmvalues[1]
            #originating-vm-size = vmDict[choicedata[1]][1]
            if (tpmvalues[2] - tpmvalues[1]) >= vmDict[choicedata[1]][1]: #has space to accept this replica
copy
                #print('size data',tpmvalues[2],tpmvalues[1],vmDict[choicedata[1]][1])
                #print(tpmvalues,'has space for',vm_tableDict[replica])
                #print(choice,'has space for',replica)
                #store the data
                #set success flag
                #if you try an implementation that stores all the valid moves before making any, it wont save
any calculations
                #due to some replicas picking the same target pm. updating the space on those target pms is
                #the same amount of calculations as to just make the move, and revert if not all replicas were
able to move.
                #print('moving vm x from pm y to pm z', vmDict[choicedata[1]],
bestheurResult[pmID],bestheurResult[choice])
                canmove = True
                #remove vm from old dest pm
                #update old pm space-used
                #print('old pm bef',bestheurResult[pmID])
                bestheurResult[pmID] = [x for x in bestheurResult[pmID] if x != vmDict[choicedata[1]]]
                bestheurResult[pmID][1] -= vmDict[choicedata[1]][1] #subtract vm-size from space-used, on old
dest pm
                #print('old pm aft',bestheurResult[pmID])

                #print('new pm bef',bestheurResult[choice])
                #add vm to new dest pm
                #update new pm space-used
                bestheurResult[choice].append(vmDict[choicedata[1]])
                bestheurResult[choice][1] += vmDict[choicedata[1]][1] #add vm-size to space-used, on new
dest pm
                #print('new pm aft',bestheurResult[choice])
                #print()
                movesmade.append([ vmDict[choicedata[1]], bestheurResult[pmID], bestheurResult[choice] ]
)#save orig-vm-data, old-dest-pm, new-dest-pm
                break
        if canmove == False:
```

```
            canmoveall = False
    if canmoveall == False:
        #revert
        #bestheurResult = {k:v for k,v in beforemove.items()}
        for move in movesmade:
            #vmsize = move[0][1]
            #print('reverting',move[0])
            #vm, oldpm, newpm
            #print('before revert',move)
            move[2].remove(move[0])
            move[2][1] -= move[0][1] #update pm-used, subtract vm-size
            move[1].append(move[0])
            move[1][1] += move[0][1] #update pm-used, add vm-size
            #print('after  revert',move)
            #print()

#print('calculation time =',time.time() - start_time)
bestheurList = []
numactivePM_bestheurList = 0
for v in bestheurResult.values():
    bestheurList.append(v)
    if len(v) > 3:
        numactivePM_bestheurList +=1
print('best heuristic, numactivePM:',numactivePM_bestheurList)
printPM(bestheurList)


def updatePMspaceused(pmlist):
    pmlistcopy = deepcopy(pmlist)
    for pm in pmlistcopy:
        if len(pm) > 3: #contains VMs
            vms = pm[3:]
            spaceused = 0
            for v in vms:
                spaceused += v[1]
            pm[1] = spaceused
        else:
            pm[1] = 0
    return pmlistcopy

#use google's OR-Tools to optimally solve consolidation
#as a bin packing problem w constraints
#constraint 1: choices are limited to mcf solution equivalent cost choices
#constraint 2: replication constraint (copies of a particular vm cannot share a pm)
#constraint 3: capacity constraint of each pm

optimalbinpack = deepcopy(pmList)
optimalbinpack = [pm[:3] for pm in optimalbinpack] #remove orig vms
optimalbinpack = [[a[0],0,a[2]] for a in optimalbinpack] #set used space=0
```

```
print('\n Optimal bin packing solution:')
print('\t constraints:')
print('\t PM capacity, replication, MCF equivalent cost choice\n')
vmdict = {}
for v in vmList:
    vmdict[v[0]] = (v[1],v[2]) #key: vmID, value: (vmSize,pmLocation)

data = {}
#add the weights, which is the size of each vm copy
weights=[]
items=[]
itemnumber = 0
for vm in vmdict:
    for r in range(numCopies):
        weights.append(vmdict[vm][0]) #append vm size
        items.append(itemnumber) #originating vmID = itemnumber // numCopies
        itemnumber += 1

data['weights'] = weights
data['items'] = items
data['bins'] = list(range(numPM))
bin_capacity = [pm[2] for pm in optimalbinpack]
#dont need pm[2] - pm[1], no space used, orig vms are temporarily removed
data['bin_capacity'] = bin_capacity

def mutex(vm1,vm2): #vm1 cannot go in same pm as vm2
    for j in data['bins']:
        solver.Add(x[vm1,j] + x[vm2,j] <= 1)
    return

#Instantiate MIP solver with CBC backend
solver = pywraplp.Solver('simple_mip_program',
                         pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING)

# Variables
# x[i, j] = 1 if vm i is packed in pm j.
x = {}
for i in data['items']:
    for j in data['bins']:
        x[(i, j)] = solver.IntVar(0, 1, 'x_%i_%i' % (i, j))
        #x[(i, j)] = solver.IntVar(0, 1, '') #testing blank name
# y[j] = 1 if pm j is used.
y = {}
for j in data['bins']:
    y[j] = solver.IntVar(0, 1, 'y[%i]' % j)
    #y[j] = solver.IntVar(0, 1, '')
# Constraints
# Each vm must be in exactly one pm.
```

```
for i in data['items']:
    solver.Add(sum(x[i, j] for j in data['bins']) == 1)

#constrain each PM to its remaining space capacity
for j in data['bins']:
    solver.Add(
    sum(x[(i, j)] * data['weights'][i] for i in data['items']) <= y[j] * data['bin_capacity'][j])
'''
multiplying by y[j] forces y[j] to equal 1 if any vm is packed in pm j.
if y[j] were 0, the right side of the inequality would be 0, while the pm-space-used on the left side would be
greater than 0,
violating the constraint.
the solver minimizes the number of pms where y[j] is 1.
'''
'''
#constrain each vm replica copy to avoid originating pm
for i in data['items']:
    vmID = i//replicaCopies #vmcopy i is a replica of vmID
    avoidPM = vmdict[vmID][1] #gets originating pm id
    solver.Add(x[i,avoidPM] == 0) #vmcopy i avoids originating pm
    if avoidPM == 0:
        print(i,'avoids',avoidPM)
'''
#vm table format: vmid (non uniq), originating pm id, destination pm, cost, equivalent cost choices
choices_table = [x[1:] for x in vm_table]
#vmList format: vmid, size, orig pm
for v in vmList:
    pretendchoice = [v[0],v[2],v[2],0,[v[2]]]
    choices_table.append(pretendchoice)
choices_table=sorted(choices_table, key = lambda x: (x[0],len(x[4]))) #sort by vm id first, number of
choices second
#constrain replica copies of the same vm to avoid other copies of that vm
exdict={}
for i in data['items']:
    vmID = i//numCopies
    startvalue = vmID * numCopies
    for r in range(startvalue,startvalue+numCopies):
        if i == r: #do not mutually exclude itself
            continue
        if (r,i) in exdict.keys(): #already did this combination
            continue
        mutex(i,r) #but DO mutex other copies of the vm
        exdict[(i,r)] = 1
        #print(r,'avoids',i,r//numCopies,i//numCopies)

def vmXgoesinpmY(vmid,pmid):
    strexp = 'x[' + str(vmid) + ',' + str(pmid) + ']'
    return strexp
```

```
#constrain choices to mcf equivalent cost choices
for v in range(len(choices_table)):
    choicelist = choices_table[v][4]
    totalexp = ''
    for c in choicelist: #build constraint expression as string, then eval
        totalexp = totalexp + vmXgoesinpmY(v,c) + ' + '
    totalexp = totalexp[:-3] #truncate last 3 chars, the extra ' + '
    totalexp += ' == 1'
    totalexp = eval(totalexp) #eval lets us eval the string as an expression
    solver.Add(totalexp)

pmdict = {}
for pm in optimalbinpack: #translate list into dict for faster lookups
    pmdict[pm[0]] = pm[1:] #key is pmid, value is rest of pm attributes

# Objective: minimize the number of PMs used.
solver.Minimize(solver.Sum([y[j] for j in data['bins']]))
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    num_bins = 0.
    for j in data['bins']:
        if y[j].solution_value() == 1:
            bin_items = []
            bin_weight = 0
            for i in data['items']:
                if x[i, j].solution_value() > 0:
                    vmid = i//numCopies
                    bin_items.append(vmid)
                    #add vm to pm in pmdict [vmid, size, origpmID]
                    addvm = [vmid, vmdict[vmid][0],vmdict[vmid][1]]
                    pmdict[j].append(addvm)
                    bin_weight += data['weights'][i]
            if bin_weight > 0:
                num_bins += 1
                #print(' %-5d  %-8d %-10d'%(pm[0],pm[1],pm[2]),end=' ')
                print('\t PM # %-5d' %(j), end = ' ')
                print(' stores VMs:', bin_items)
                #print('           using:', bin_weight, 'space')
    print()
    #print('PMs used:', int(num_bins))
    print('\tOptimal Bin Packing calculation time = ', solver.WallTime(), ' milliseconds')
else:
    print('No optimal solution is possible.')

optimalbinpack = [] #rebuild pm list from dict
numactive_optimalbinpack=0
for key,values in pmdict.items():
    pm = []
```

```
    pm.append(key)
    if len(values)>2:
        numactive_optimalbinpack+=1
    for v in values:
        pm.append(v)
    optimalbinpack.append(pm)

optimalbinpack = updatePMspaceused(optimalbinpack)

printPM(optimalbinpack)
print(numMCFActivePM,'PMs used by MCF')
print(numRFFactivePM, 'replacement firstfit consolidation method')
print(numFFactivePM, 'firstFit consolidation heuristic')
print(numactivePM_bestheurList,'best consolidation heuristic')
print(numactive_optimalbinpack,'linear solver method')
'''
if checkNumActivePM(optimalbinpack) < checkNumActivePM(originalconsolidationpmlist):
    print('\n')
    print(pmList)
    print(vmList)
'''
'''
def mutex (i1, i2): #i1 and i2 become mutually exclusive in any given bin
    for j in data['bins']:
      solver.Add((x[i1,j] + x[i2,j]) <= 1)
    return

Useful examples of adding constraints:
#item 1 can't go in bin 6
solver.Add((x[1, 6] == 0))

#item 1 can't go in the same bin as item 2
#for j in data['bins']:
#    solver.Add((x[1,j] + x[2,j]) <= 1)

#item 0 can't go in the same bin as any other item
for jk in range(all bin numbers):
    if jk != 0: #avoid excluding itself from itself
        mutex(0,jk)

#item 5 must go in bins 9 or 10
solver.Add((x[5,9] + x[5,10]) == 1)


'''
#sort dictionary example
#x = {1: 2, 3: 4, 4: 3, 2: 1, 0: 0}
#print(x)
#x= {k: v for k, v in sorted(x.items(), key=lambda item: item[1])}
```

Copy of fattree_writeonly.py (no screen printing, writes summary of results to fattreeresults.txt):

```
import random #used to randomize initial pm and vm placement
from copy import deepcopy #used for saving pm states before each algorithm
from ortools.graph import pywrapgraph #used for mcf optimal solver
from ortools.linear_solver import pywraplp #used for bin packing optimal solver
#import time #used to time algorithms
import sys
import os

def calcPMtoPMcost(pm1,pm2):
    pm1=int(pm1)
    pm2=int(pm2)
    if pm1 == pm2: #pm to itself costs 0
        return 0
    pmPerEdge=int(numPM / totalEdge)
    pm1Edge=int(pm1 / pmPerEdge)
    pm2Edge=int(pm2 / pmPerEdge)
    if pm1Edge == pm2Edge: #pms on same edge switch costs 2
        return 2
    pm1Pod=int(pm1Edge / numEdgePerPod)
    pm2Pod=int(pm2Edge / numEdgePerPod)
    if pm1Pod == pm2Pod: #pms on same pod costs 4
        return 4
    return 6 #routing through core switch costs 6

with open('fattreesettings.txt','r') as f:
    settings=[]
    for line in f:
        line=line.strip().split()
        settings.append(line)

#read in settings saved from txt file
k=          int(settings[0][1])
numVM=      int(settings[1][1])
minVMsize=  int(settings[2][1])
maxVMsize=  int(settings[3][1])
numCopies=  int(settings[4][1]) #replica + original
minPMsize=  int(settings[5][1])
maxPMsize=  int(settings[6][1])
numRuns=    int(settings[7][1])

fattreek = k
#how many runs to simulate,
#only used in write-only version for data collection
'''
fattreesettings.txt should be in this format:
k 4
numVM 4
minVMsize 1
```

```
maxVMsize 1
numCopies 5
minPMsize 30
maxPMsize 30
'''

#calculate number of switches and edges
#this doesn't change with randomized placements. can leave outside of loop
numPod=int(k)
numCore=int((k/2)**2)
numAggPerPod=int(k/2)
numEdgePerPod=int(k/2)
totalAgg=numAggPerPod * numPod
totalEdge=numEdgePerPod * numPod
numPM=int((k**3) / 4)
totalSwitches=int(numCore + totalAgg + totalEdge)
totalNode=totalSwitches + numPM
totalEdges=numPM * 3

#set of lists for tracking the stats of each algorithm per run ####################################
firstFitFlowCost_list = []
firstfit_pms_used_list = []
greedyFlowCost_list = []
greedy_pms_used_list = []
mcf_flowcost_list = []

numMCFActivePM_list = []
numRFFactivePM_list = []
numFFactivePM_list = []
numactivePM_bestheurList_list = []
numactive_optimalbinpack_list = []

improvementAmountList = []

improvementcount=0
errorcount=0
samecount=0
##################LOOP STARTS HERE#########################################

for number in range(numRuns):
    pmDict={}
    #pmDict has [ [pmID, space-used, max-space] ,
    #[containedVM1-id, vm-size, originatingPM-id], [containedVM2-id...] ]

    totalPMCapacity = 0
    for x in range(numPM):
        randomPMsize = random.randint(minPMsize,maxPMsize)
        totalPMCapacity+=randomPMsize
        pmDict[x] = [x,0,randomPMsize]
```

```
    #creating cost dictionary for later reference
    cost_dict={}
    for x in range(numPM):
        for y in range(x):
            costkey=(x,y)
            costvalue=calcPMtoPMcost(x,y)
            cost_dict[costkey]=costvalue
            reversecostkey = (y,x) #reverse pair has same cost
            cost_dict[reversecostkey]=costvalue
    for x in range(numPM): #pm to itself is 0
        costkey = (x,x)
        cost_dict[costkey] = 0

    vmList = []
    totalOrigVMsize = 0
    totalOrigPlusReplicaSize = 0
    vmDict = {}
    #key: vmID
    #value: [vm-id, vm-size, originating-PM-id]

    for x in range(numVM):
        randomVMsize = random.randint(minVMsize,maxVMsize)
        totalOrigVMsize += randomVMsize
        totalOrigPlusReplicaSize += (randomVMsize * numCopies)

        #find a PM to place this VM on
        placed = False
        allPMids = [y for y in range(numPM)]
        #random.shuffle(allPMids) #not necessary to shuffle since we pop randomly
        for p in range(len(allPMids)):
            randomPM = allPMids.pop(random.randint(0,len(allPMids)-1))
            randomPM = pmDict[randomPM]
            randomPMspaceremaining = randomPM[2] - randomPM[1] #max - used
            if randomPMspaceremaining >= randomVMsize:
                randomPMid = randomPM[0]
                placed = True
                thisvm = [x,randomVMsize,randomPMid]
                vmList.append(thisvm) #add to vm list
                vmDict[x]=thisvm #add to vm dict
                pmDict[randomPMid].append(thisvm)
                pmDict[randomPMid][1] += randomVMsize #update pm used space
                break
        if placed == False: #never placed the VM
            print('warning, VM',x,' could not be placed')
            errorcount+=1
            sys.exit() #being unable to place an orig vm is a serious error. it implies there is absolutely no
space for replication.
```

```
    replicaSize = totalOrigPlusReplicaSize - totalOrigVMsize

    #print('VM-ID, VM-Size, PM-location')
    #for x in vmList:
    #    print('%5s %6s %5s' %(str(x[0]),str(x[1]),str(x[2])))

    pmList= []
    for k,v in pmDict.items():
        pmList.append(v)
    '''
    def printPM(pmList):
        print('\nPM- | PM-space | PM-max-  | Assigned-')
        print('ID  | used     | capacity | VM(s)')
        print('-'*50)
        for pm in pmList:
            print(' %-5d  %-8d %-10d'%(pm[0],pm[1],pm[2]),end=' ')
            for vm in pm[3:]:
                print('%-3d'%(vm[0]),end=' ')
            print()
        print()

    print('\nInitial, random original VM placement')
    printPM(pmList)
    '''
    ######################### nonrandom pmlist and vmlist for testing purposes
    #pmList = [[0, 0, 30], [1, 0, 30], [2, 0, 30], [3, 0, 30], [4, 0, 30], [5, 0, 30], [6, 0, 30], [7, 1, 30, [2, 1, 7]], [8,
0, 30], [9, 0, 30], [10, 0, 30], [11, 0, 30], [12, 1, 30, [0, 1, 12]], [13, 1, 30, [1, 1, 13]], [14, 0, 30], [15, 1, 30,
[3, 1, 15]]]
    #vmList = [[0, 1, 12], [1, 1, 13], [2, 1, 7], [3, 1, 15]]
    ########################

    #first fit flow heuristic. in arbitrary order, each VM replica copy chooses the lowest ID available PM. flow
cost ignored.
    firstFitDict = deepcopy(pmDict)
    firstFit_pmList=deepcopy(pmList) #deep copy pmList for different solutions
    firstFitFlowCost = 0

    def pmXcontainsvmY(pmx,vmy):
        for x in firstFitDict[pmx[0]][3:]:
            if x[0] == vmy[0]:
                return True
        return False

    replicaCopies=numCopies-1
    if replicaCopies > 0:
        for vm in vmList:
            needtoplace = replicaCopies
            placed = 0
            transferCost = 0
```

```
        for pm in firstFit_pmList:
            if placed==replicaCopies: #already placed enough replica copies
                break
            if pmXcontainsvmY(pm,vm):
                continue
            pmvalues = firstFitDict[pm[0]]
            if pmvalues[2] - pmvalues[1] >= vm[1]: #pm-max - pm-used >= vm-size
                needtoplace-=1
                placed+=1
                pmvalues[1] += vm[1] #update space used
                pmvalues.append(vm)
                firstFitDict[pm[0]] = deepcopy(pmvalues)
                costkey = (pm[0],vm[2])
                transferCost += (cost_dict[costkey] * vm[1])
        if needtoplace != 0:
            print('warning: didnt place all copies of VM',vm[0])
            sys.exit()#this should never happen,
            #but just exit instead of continuing all calculations
            errorcount+=1
        firstFitFlowCost+=transferCost

firstfitPMlist = []
firstfit_pms_used = 0
for k,v in firstFitDict.items():
    firstfitPMlist.append(v)
    if len(v) > 3:
        firstfit_pms_used+=1
#print('\nFirstFit, flowcost:',firstFitFlowCost,' num-active-PMs:',firstfit_pms_used)
#printPM(firstfitPMlist)

#greedy: each vm seeks its lowest flow-cost available PM
greedy_pmList=deepcopy(pmList)
greedy_pmDict=deepcopy(pmDict)
greedyFlowCost=0

def greedycheckPMcontainsVM(pmx,vmy):
    for x in greedy_pmDict[pmx[0]][3:]:
        if x[0] == vmy[0]:
            return True
    return False

greedyFlowCost=0
for v in vmList:
    for y in range (replicaCopies):
        choices=[]
        for pm in greedy_pmList:#calc flow cost from origPM to all other PMs
            #pmID=pm[0]
            #origPM=v[2]
            costkey = (v[2],pm[0])
```

```
            #cost = cost_dict[costkey]
            choice_tuple=(pm[0],cost_dict[costkey])
            choices.append(choice_tuple)

        choices=sorted(choices, key = lambda x: x[1]) #sort by cost of choices, lowest cost first
        transferCost=0
        for choice in choices:
            #pmID=choice[0], vmID=v[0]
            if greedycheckPMcontainsVM(choice,v):
                continue #if pm contains this vm already
            pmvalues = greedy_pmDict[choice[0]]
            #vmSize=v[1]
            if pmvalues[2] - pmvalues[1] >= v[1]: #pm-max - pm-used >= vm-size
                pmvalues[1] += v[1] #update space used
                pmvalues.append(v)
                greedy_pmDict[choice[0]] = deepcopy(pmvalues)
                costkey = (choice[0],v[2])
                transferCost += (cost_dict[costkey] * v[1])
                break #stop trying to place in more choices PMs
        greedyFlowCost+=transferCost

greedy_pmList = []
greedy_pms_used = 0
for k,v in greedy_pmDict.items():
    greedy_pmList.append(v)
    if len(v) > 3:
        greedy_pms_used+=1
#print('\nGreedy, flowcost:',greedyFlowCost,' num-active-PMs:',greedy_pms_used)
#printPM(greedy_pmList)

#Minimum Cost Flow Solution
mcf_pmDict=deepcopy(pmDict) #copy over dicts so we dont change the originals
mcf_vmDict=deepcopy(vmDict)

#Instantiate an OR-Tools SimpleMinCostFlow solver
min_cost_flow = pywrapgraph.SimpleMinCostFlow()
#print('Minimum Cost Flow Solution: Pre-Consolidation')

superSourceID = numVM
superSinkID = numVM+1
offset = numVM+2 #PM ids will be offset by this number, to avoid conflict

superSourceSupply=replicaSize #set source supply to sum of replica sizes
min_cost_flow.SetNodeSupply(superSourceID,superSourceSupply)

superSinkDemand=superSourceSupply * -1 #set sink demand
min_cost_flow.SetNodeSupply(superSinkID,superSinkDemand)

#generate arcs from super source to each orig vm
```

```
   #min_cost_flow.AddArcWithCapacityAndUnitCost parameters are
   #                                (start-node, end-node, capacity, unit-cost)
   finalsetdict={}
   for k in range(numVM):
      thisvmsize = mcf_vmDict[k][1] #get size of this vm
      #add arcs from super-source to each original VM
      min_cost_flow.AddArcWithCapacityAndUnitCost(superSourceID,k,replicaCopies*thisvmsize,0)
      thisvmorigPM = mcf_vmDict[k][2]
      for pmID in mcf_pmDict.keys():
         if pmID == thisvmorigPM: #dont create arc to originating-PM
            continue
         costkey = (thisvmorigPM,pmID)
         cost = cost_dict[costkey]
         #add arcs from each orig-VM to potential-destination-PMs
         newPMid = pmID+offset
         min_cost_flow.AddArcWithCapacityAndUnitCost(k,newPMid,thisvmsize,cost)
         if newPMid not in finalsetdict:
            finalsetdict[newPMid] = 0 #value doesnt matter here, storing the key does
            pm_remaining_space = mcf_pmDict[pmID][2] - mcf_pmDict[pmID][1]#maxSpace - usedSpace
            #generate arcs from each potential-destination-PM to the super sink

min_cost_flow.AddArcWithCapacityAndUnitCost(newPMid,superSinkID,pm_remaining_space,0)

   #vm_table holds information about MCF equivalent cost choices for each VM

   if min_cost_flow.Solve() == min_cost_flow.OPTIMAL:
      vm_table = []
#       print('Minimum cost:', min_cost_flow.OptimalCost(),end='  ')
      mcf_flowcost = min_cost_flow.OptimalCost()
      copyid=0
      for i in range(min_cost_flow.NumArcs()):
         cost = min_cost_flow.Flow(i) * min_cost_flow.UnitCost(i)
         if min_cost_flow.Flow(i) > 0 and cost > 0:

#a,b,c,d,e=min_cost_flow.Tail(i),min_cost_flow.Head(i),min_cost_flow.Flow(i),min_cost_flow.Capacity(i),c
ost
            origVMid,destinationPMid=min_cost_flow.Tail(i),min_cost_flow.Head(i)
            destinationPMid -= offset
            #vm_table will have structure of: [copyID, origVMid, orig PM, destination PM, sunk cost]
            vm_table.append([copyid,origVMid,mcf_vmDict[origVMid][2],destinationPMid,cost])
            copyid+=1
            #update the pm dict to show a vm was replicated to the destination PM
            pmvalues = mcf_pmDict[destinationPMid]
            pmvalues.append(mcf_vmDict[origVMid])
            pmvalues[1] += mcf_vmDict[origVMid][1] #update pm-space-used
            mcf_pmDict[destinationPMid] = deepcopy(pmvalues)

   else:
      errorcount+=1
```

```
        pass #pass, this shouldnt happen. could sys.exit
        #print('Error: There was an issue with the min cost flow input.')


    mcf_pmList=[]
    numMCFActivePM = 0
    for v in mcf_pmDict.values():
        if len(v) > 3:
            numMCFActivePM+=1#tally up active PMs
        mcf_pmList.append(v)
    #print('num-active-PMs:',numMCFActivePM)
    #printPM(mcf_pmList)

    #EQUIVALENT-COST-DESTINATION-PM-CHOICE TABLE HERE
    #print('replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs')
    for vm in vm_table:
        #filter the cost_dict for equivalent cost choices, using origPM and cost
        #append the list of equivalent-cost-choice-PMs to each replicaVM
        #origpm,cost = vm[2],vm[4]
        filtered_list = [k[1] for k,v in cost_dict.items() if (k[0] == vm[2]) and v == vm[4]]
        vm.append(filtered_list)

        #print out the replica-vm-table
    #    for y in vm:
    #        print(y,'\t',end='  ')
    #    print()

    rffPMdict = deepcopy(pmDict)

    #Replacement first fit, consolidation heuristic
    #print('\nReplacement First Fit - MCF cost considered, MCF placement disregarded')
    #acts as though all vms are unplaced but mcf cost is known, then first fit places them according to
equivalent mcf cost per copy

    def RFFcheckPMcontainsVM(pmID,vmID):
        for vm in rffPMdict[pmID][3:]:
            if vm[0] == vmID:
                return True
        return False

    for vm in vm_table:
        for choice in vm[5]:
            #check if pm contains this vm already. replication constraint.
            vmID = vm[1]
            #pmID = choice
            if RFFcheckPMcontainsVM(choice,vmID):
                continue
            #pm-space-max  = rffPMdict[pmID][2]
            #pm-space-used = rffPMdict[pmID][1]
```

```
            #pm-space-remaining = max - used
            if (rffPMdict[choice][2] - rffPMdict[choice][1]) >= vmDict[vmID][1]:#pm has enough space, we can
place
                pmvalues = rffPMdict[choice]
                vmSize = vmDict[vmID][1]#vmID = vm[1]
                addVM = [vmID,vmSize,vm[2]]#vmOrigPM = vm[2] or vmDict[vmID][2]
                pmvalues[1] += vmSize
                pmvalues.append(addVM)
                rffPMdict[choice] = deepcopy(pmvalues) #[x for x in pmvalues]
                break

    #find active PMs and visualize rff solution
    rff_pmList=[]
    numRFFactivePM = 0
    for v in rffPMdict.values():
        if len(v) > 3:
            numRFFactivePM+=1#tally up active PMs
        rff_pmList.append(v)
    #print('num-active-PMs:',numRFFactivePM)
    #printPM(rff_pmList)

    #First fit consolidation heuristic
    #Keeps mcf placement, tries to firstfit consolidate from there, using mcf-equivalent-cost for each copy
    #will only transfer if destination-PM is not empty

    #print('\nFirst Fit Consolidation - from MCF solution, \nEach VM jumps to a same-cost, lowest-id PM')

    #before transfer dictionary
    ffnotemptyDict = {k:v for k,v in mcf_pmDict.items()}

    def FFcheckPMcontainsVM(pmID,vmID):
        for vm in ffnotemptyDict[pmID][3:]:
            if vm[0] == vmID:
                return True
        return False

    #print('replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs')
    for vm in vm_table:
        for choice in vm[5]:
            vmID = vm[1]
            #pmID = choice
            if len(ffnotemptyDict[choice]) == 3: #pm is empty, dont transfer
                continue
            if choice == vm[3]:
                continue #dest-pm is same as mcf solution, dont transfer
            if FFcheckPMcontainsVM(choice,vmID):
                continue #dest-pm contains a copy of this vm already, dont transfer
            #pm-space-max  = ffnotemptyDict[pmID][2]
            #pm-space-used = ffnotemptyDict[pmID][1]
```

```
            #pm-space-remaining = max - used
            if (ffnotemptyDict[choice][2] - ffnotemptyDict[choice][1]) >= vmDict[vmID][1]:
                #pm has enough space, we can transfer
                pmvalues = [x for x in ffnotemptyDict[choice]]
                vmSize = vmDict[vmID][1]#vmID = vm[1]
                #add vm to chosen dest PM
                addVM = [vmID,vmSize,vm[2]]#vmOrigPM = vm[2] or vmDict[vmID][2]
                pmvalues[1] += vmSize
                pmvalues.append(addVM)
                ffnotemptyDict[choice] = [x for x in pmvalues] #[x for x in pmvalues]

                #remove vm from previous dest PM
                ffnotemptyDict[vm[3]] = [x for x in ffnotemptyDict[vm[3]] if x != addVM]
                ffnotemptyDict[vm[3]][1] -= vmSize
                break

    #find num-active-PMs and visualize ff solution
    ff_pmList=[]
    numFFactivePM = 0
    howmanyvms = 0
    for v in ffnotemptyDict.values():
        if len(v) > 3:
            numFFactivePM+=1#tally up active PMs
            for vm in v[3:]:
                howmanyvms+=1
        ff_pmList.append(v)
    #print('num-active-PMs:',numFFactivePM)
    #printPM(ff_pmList)

    #implement original post-MCF consolidation as described in Dr. Tang's paper
    #look at PMs with 1 VM in arbitrary order. for each PM, check if all its containing VMs can be moved
elsewhere to a set of TPM. if yes, move them all. if no, dont move any.
    #repeat with PMs with 2 VMs, then PMs with 3 VMs, until you reach max possible VM #.

    #start_time = time.time()
    '''
    def printDict(anydict, label = '\nprinting some dict'):
        print(label)
        for k,v in anydict.items():
            print(k,v)
        print()
    '''
    #copy mcf solution dict but sort by number of vms
    #bestconsolidationheuristicDict = {k:v for k,v in sorted(mcf_pmDict.items(), key = lambda x: len(x[1]))}
    #printDict(bestconsolidationheuristicDict)

    #print('Best consolidation heuristic, tries to move all PMs containing')
    #print('1 VM, then 2 VM, then 3 VM, ... only if all VMs can be moved')
    #start_time = time.time()
```

```
    #if PM contains original vm, cant move
    #if destinationPM is VM's origPM, cant move
    #cost-equivalent
    #destinationPM cannot be empty
    #check replica constraint
    #check PMhasspace
    #update space
    #add vm to new pm
    #remove vm from old pm
    bestheurDict = {k:v for k,v in sorted(pmDict.items(), key = lambda x: len(x[1]))}
    #exclude PMs which contain the original VMs
    excludelist = [v[2] for v in vmList]
    bestheurResult = {k:v for k,v in mcf_pmDict.items()} #final positions will be saved here

    vm_tableDict = {} #this is the dict equivalent of vm_table

    #build pm-dict using replica ids, not just orig vm ids, for lookup
    for vm in vm_table:
        #add vm to dest pm
        #origvmSize = vmDict[vm[1]][1]
        #destpm = vm[3]
        pmvalues = bestheurDict[vm[3]]
        pmvalues.append(vm[0]) #add the replicaID to the destPM

        #since we're looping through the list, build a dict for later
        vm_tableDict[vm[0]] = [x for x in vm]


    sortbynumcontainedVMs={}

    for k,v in bestheurDict.items():
        if k in excludelist:
            #dont bother trying to consolidate PMs which contain-
            #an original vm
            #print('skipping',k)
            continue
        sortbynumcontainedVMs[k] = [x for x in v[3:]]

    #print('replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs')
    sortbynumcontainedVMs = sorted(sortbynumcontainedVMs, key = lambda
x:len(sortbynumcontainedVMs[x])) #sort PM-ids by number of contained vms
    #print(sortbynumcontainedVMs)
    #we try to consolidate starting from PMs containing the fewest replica VMs
    #skipping PMs which contain no VMs
    #skipping PMs which contain an original VM
    #skipping PMs in which not all VMs can be moved
    #valid alternate PM must be: not empty, same cost, contain no other copies of that vm, has enough
space
```

```
    '''
            #check if alternate choice has space
            #pmMax - pmUsed >= originatingVMsize
            #pmMax  = tpmvalues[2]
            #pmUsed = tpmvalues[1]
            #originating-vm-size = vmDict[choicedata[1]][1]

                #print('size data',tpmvalues[2],tpmvalues[1],vmDict[choicedata[1]][1])
                #print(tpmvalues,'has space for',vm_tableDict[replica])
                #print(choice,'has space for',replica)
                #store the data
                #set success flag
                #if you try an implementation that stores all the valid moves before making any, it wont save
any calculations
                #due to some replicas picking the same target pm. updating the space on those target pms is
                #the same amount of calculations as to just make the move, and revert if not all replicas were
able to move.
                #print('moving vm x from pm y to pm z', vmDict[choicedata[1]],
bestheurResult[pmID],bestheurResult[choice])


    '''
    for pmID in sortbynumcontainedVMs:
        pmvalues = [x for x in bestheurDict[pmID]] #trying to consolidate this PM
        #print('\npmvalues',pmvalues)
        #print('lenpmvalues3:',len(pmvalues[3:])) #num of replica copies on this pm
        #print('PMID',pmID)
        canmoveall = True
        movesmade = []
        #beforemove = {k:v for k,v in bestheurResult.items()} #save state before attempting moves
        for replica in pmvalues[3:]: #check if each replica can be moved. PMs that have no replicas are
skipped here.
            choicedata = [x for x in vm_tableDict[replica]]
            #choicedata is replicaID,origVMid,origPM,destPM,flowcost,equiv-cost-choice-PMs
            #print('choicedata',choicedata)
            #we are looking at each replica copy's choices now
            canmove = False
            for choice in choicedata[5]: #these are the equiv-cost-PMs
                #print('choice',choice)
                #check that new dest is not same as old dest
                if choice == choicedata[3]: #cant move, new pm is same as old pm
                    #print('dest pm is same as old dest', choice, choicedata[3])
                    continue

                tpmvalues = [x for x in bestheurResult[choice]] #makes a copy instead of a reference to
targetPM
                #print('tpmvalues',tpmvalues)
                #check that dest pm is not empty. we wont transfer to an empty pm
                if len(tpmvalues) == 3:
```

```
                #print(tpmvalues,'is empty, cant move')
                continue

            #check if alternate pm choice contains any copy of this vm
            containsflag = False
            for containedvm in tpmvalues[3:]:
                if containedvm[0] == choicedata[1]: #found match for this vm, cant move
                    #print('vm match, cant move', containedvm[0],choicedata[1])
                    containsflag = True
                    break
            if containsflag == True:
                continue #skip to next possible choice, dont place here.

            if (tpmvalues[2] - tpmvalues[1]) >= vmDict[choicedata[1]][1]: #has space to accept this replica
copy

                canmove = True
                #remove vm from old dest pm
                #update old pm space-used
                #print('old pm bef',bestheurResult[pmID])
                bestheurResult[pmID] = [x for x in bestheurResult[pmID] if x != vmDict[choicedata[1]]]
                bestheurResult[pmID][1] -= vmDict[choicedata[1]][1] #subtract vm-size from space-used, on
old dest pm
                #print('old pm aft',bestheurResult[pmID])

                #print('new pm bef',bestheurResult[choice])
                #add vm to new dest pm
                #update new pm space-used
                bestheurResult[choice].append(vmDict[choicedata[1]])
                bestheurResult[choice][1] += vmDict[choicedata[1]][1] #add vm-size to space-used, on new
dest pm
                #print('new pm aft',bestheurResult[choice])
                #print()
                movesmade.append([ vmDict[choicedata[1]], bestheurResult[pmID], bestheurResult[choice] ]
)#save orig-vm-data, old-dest-pm, new-dest-pm
                break
        if canmove == False:
            canmoveall = False
    if canmoveall == False:
        #revert
        #bestheurResult = {k:v for k,v in beforemove.items()}
        for move in movesmade:
            #vmsize = move[0][1]
            #print('reverting',move[0])
            #vm, oldpm, newpm
            #print('before revert',move)
            move[2].remove(move[0])
            move[2][1] -= move[0][1] #update pm-used, subtract vm-size
            move[1].append(move[0])
```

```
            move[1][1] += move[0][1] #update pm-used, add vm-size
            #print('after  revert',move)

#print('calculation time =',time.time() - start_time)
bestheurList = []
numactivePM_bestheurList = 0
for v in bestheurResult.values():
    bestheurList.append(v)
    if len(v) > 3:
        numactivePM_bestheurList +=1
#print('best heuristic, numactivePM:',numactivePM_bestheurList)
#printPM(bestheurList)


def updatePMspaceused(pmlist):
    pmlistcopy = deepcopy(pmlist)
    for pm in pmlistcopy:
        if len(pm) > 3: #contains VMs
            vms = pm[3:]
            spaceused = 0
            for v in vms:
                spaceused += v[1]
            pm[1] = spaceused
        else:
            pm[1] = 0
    return pmlistcopy

#use google's OR-Tools to optimally solve consolidation
#as a bin packing problem w constraints
#constraint 1: choices are limited to mcf solution equivalent cost choices
#constraint 2: replication constraint (copies of a particular vm cannot share a pm)
#constraint 3: capacity constraint of each pm

optimalbinpack = deepcopy(pmList)
optimalbinpack = [pm[:3] for pm in optimalbinpack] #remove orig vms
optimalbinpack = [[a[0],0,a[2]] for a in optimalbinpack] #set used space=0

#print('\n Optimal bin packing solution:')
#print('\t constraints:')
#print('\t PM capacity, replication, MCF equivalent cost choice\n')
vmdict = {}
for v in vmList:
    vmdict[v[0]] = (v[1],v[2]) #key: vmID, value: (vmSize,pmLocation)

data = {}
#add the weights, which is the size of each vm copy
weights=[]
items=[]
itemnumber = 0
```

```
    for vm in vmdict:
        for r in range(numCopies):
            weights.append(vmdict[vm][0]) #append vm size
            items.append(itemnumber) #originating vmID = itemnumber // numCopies
            itemnumber += 1

    data['weights'] = weights
    data['items'] = items
    data['bins'] = list(range(numPM))
    bin_capacity = [pm[2] for pm in optimalbinpack]
    #dont need pm[2] - pm[1], no space used, orig vms are temporarily removed
    data['bin_capacity'] = bin_capacity

    def mutex(vm1,vm2): #vm1 cannot go in same pm as vm2
        for j in data['bins']:
            solver.Add(x[vm1,j] + x[vm2,j] <= 1)
        return

    #Instantiate MIP solver with CBC backend
    solver = pywraplp.Solver('simple_mip_program',
                             pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING)

    # Variables
    # x[i, j] = 1 if vm i is packed in pm j.
    x = {}
    for i in data['items']:
        for j in data['bins']:
            x[(i, j)] = solver.IntVar(0, 1, 'x_%i_%i' % (i, j))
            #x[(i, j)] = solver.IntVar(0, 1, '') #testing blank name
    # y[j] = 1 if pm j is used.
    y = {}
    for j in data['bins']:
        y[j] = solver.IntVar(0, 1, 'y[%i]' % j)
        #y[j] = solver.IntVar(0, 1, '')
    # Constraints
    # Each vm must be in exactly one pm.
    for i in data['items']:
        solver.Add(sum(x[i, j] for j in data['bins']) == 1)

    #constrain each PM to its remaining space capacity
    for j in data['bins']:
        solver.Add(
        sum(x[(i, j)] * data['weights'][i] for i in data['items']) <= y[j] * data['bin_capacity'][j])
    '''
    multiplying by y[j] forces y[j] to equal 1 if any vm is packed in pm j.
    if y[j] were 0, the right side of the inequality would be 0, while the pm-space-used on the left side would
be greater than 0,
    violating the constraint.
    the solver minimizes the number of pms where y[j] is 1.
```

```
    '''
    '''
    #constrain each vm replica copy to avoid originating pm
    for i in data['items']:
        vmID = i//replicaCopies #vmcopy i is a replica of vmID
        avoidPM = vmdict[vmID][1] #gets originating pm id
        solver.Add(x[i,avoidPM] == 0) #vmcopy i avoids originating pm
        if avoidPM == 0:
            print(i,'avoids',avoidPM)
    '''
    #vm table format: vmid (non uniq), originating pm id, destination pm, cost, equivalent cost choices
    choices_table = [x[1:] for x in vm_table]
    #vmList format: vmid, size, orig pm
    for v in vmList:
        pretendchoice = [v[0],v[2],v[2],0,[v[2]]]
        choices_table.append(pretendchoice)
    choices_table=sorted(choices_table, key = lambda x: (x[0],len(x[4]))) #sort by vm id first, number of
choices second
    #constrain replica copies of the same vm to avoid other copies of that vm
    exdict={}
    for i in data['items']:
        vmID = i//numCopies
        startvalue = vmID * numCopies
        for r in range(startvalue,startvalue+numCopies):
            if i == r: #do not mutually exclude itself
                continue
            if (r,i) in exdict.keys(): #already did this combination
                continue
            mutex(i,r) #but DO mutex other copies of the vm
            exdict[(i,r)] = 1
            #print(r,'avoids',i,r//numCopies,i//numCopies)

    def vmXgoesinpmY(vmid,pmid):
        strexp = 'x[' + str(vmid) + ',' + str(pmid) + ']'
        return strexp

    #constrain choices to mcf equivalent cost choices
    for v in range(len(choices_table)):
        choicelist = choices_table[v][4]
        totalexp = ''
        for c in choicelist: #build constraint expression as string, then eval
            totalexp = totalexp + vmXgoesinpmY(v,c) + ' + '
        totalexp = totalexp[:-3] #truncate last 3 chars, the extra ' + '
        totalexp += ' == 1'
        totalexp = eval(totalexp) #eval lets us eval the string as an expression
        solver.Add(totalexp)

    pmdict = {}
    for pm in optimalbinpack: #translate list into dict for faster lookups
```

```
    pmdict[pm[0]] = pm[1:] #key is pmid, value is rest of pm attributes

# Objective: minimize the number of PMs used.
solver.Minimize(solver.Sum([y[j] for j in data['bins']]))
status = solver.Solve()

#format output for the optimal solution, print optimal solution
if status == pywraplp.Solver.OPTIMAL:
    num_bins = 0.
    for j in data['bins']:
        if y[j].solution_value() == 1:
            bin_items = []
            bin_weight = 0
            for i in data['items']:
                if x[i, j].solution_value() > 0:
                    vmid = i//numCopies
                    bin_items.append(vmid)
                    #add vm to pm in pmdict [vmid, size, origpmID]
                    addvm = [vmid, vmdict[vmid][0],vmdict[vmid][1]]
                    pmdict[j].append(addvm)
                    bin_weight += data['weights'][i]
            if bin_weight > 0:
                num_bins += 1
                #print(' %-5d  %-8d %-10d'%(pm[0],pm[1],pm[2]),end=' ')
#               print('\t PM # %-5d' %(j), end = ' ')
#               print(' stores VMs:', bin_items)
                #print('          using:', bin_weight, 'space')
#    print()
    #print('PMs used:', int(num_bins))
    #print('\tOptimal Bin Packing calculation time = ', solver.WallTime(), ' milliseconds')
else:
    errorcount+=1
    pass#this should never happen, could sys.exit()
#    print('No optimal solution is possible.')

optimalbinpack = [] #rebuild pm list from dict
numactive_optimalbinpack=0
for key,values in pmdict.items():
    pm = []
    pm.append(key)
    if len(values)>2:
        numactive_optimalbinpack+=1
    for v in values:
        pm.append(v)
    optimalbinpack.append(pm)

optimalbinpack = updatePMspaceused(optimalbinpack)

firstFitFlowCost_list.append(firstFitFlowCost)
```

```
    firstfit_pms_used_list.append(firstfit_pms_used)
    greedyFlowCost_list.append(greedyFlowCost)
    greedy_pms_used_list.append(greedy_pms_used)
    mcf_flowcost_list.append(mcf_flowcost)

    numMCFActivePM_list.append(numMCFActivePM)
    numRFFactivePM_list.append(numRFFactivePM)
    numFFactivePM_list.append(numFFactivePM)
    numactivePM_bestheurList_list.append(numactivePM_bestheurList)
    numactive_optimalbinpack_list.append(numactive_optimalbinpack)

    if numactive_optimalbinpack < numactivePM_bestheurList:
        improvementAmountList.append(numactivePM_bestheurList-numactive_optimalbinpack)
        improvementcount+=1

    if numactive_optimalbinpack > numactivePM_bestheurList:
        errorcount+=1000000 #this is a joke

    if numactive_optimalbinpack == numactivePM_bestheurList:
        samecount+=1

###################################LOOP ENDS HERE
def writelistoneline(alist):
    for a in alist:
        fout.write('%-4s'%(str(a))+' ')
    fout.write('\n')
with open('fattreeresults.txt','w') as fout:
    fout.write('\t\t         k '+str(fattreek))
    fout.write('\n\t\t     numVM '+str(numVM))
    fout.write('\n\t\t minVMsize '+str(minVMsize))
    fout.write('\n\t\t maxVMsize '+str(maxVMsize))
    fout.write('\n\t\t numCopies '+str(numCopies))
    fout.write('\n\t\t minPMsize '+str(minPMsize))
    fout.write('\n\t\t maxPMsize '+str(maxPMsize))
    fout.write('\n\t\t   numRuns '+str(numRuns)+'\n\n')

    fout.write('          FirstFitFlowCost ')
    writelistoneline(firstFitFlowCost_list )
    fout.write('            GreedyFlowCost ')
    writelistoneline(greedyFlowCost_list )
    fout.write('               MinFlowCost ')
    writelistoneline(mcf_flowcost_list )
    fout.write('\n')

    fout.write('          FirstFitActivePM ')
    writelistoneline(firstfit_pms_used_list )
    fout.write('            GreedyActivePM ')
    writelistoneline(greedy_pms_used_list )
    fout.write('               MCFActivePM ')
```

```
    writelistoneline(numMCFActivePM_list )
    fout.write('\n')

    fout.write('               RFFActivePM ')
    writelistoneline(numRFFactivePM_list )
    fout.write(' mcfPlacedFirstFitActivePM ')
    writelistoneline(numFFactivePM_list )
    fout.write('      BestConsAlgoActivePM ')
    writelistoneline(numactivePM_bestheurList_list )

    fout.write('       optimalbinpack_list ')
    writelistoneline(numactive_optimalbinpack_list )
    fout.write('\n')
    '''
    print('                          error count:',errorcount)
    print('                        number of runs:',numRuns)
    print('         optimal matches best heuristic:',samecount)
    print('     optimal outperforms best heuristic:',improvementcount,'times')
    '''
    heuravg = sum(numactivePM_bestheurList_list) / len(numactivePM_bestheurList_list)
    heuravg = round(heuravg,2)



    #print('        avg servers used,best heuristic:', heuravg)

    optavg = sum(numactive_optimalbinpack_list) / len(numactive_optimalbinpack_list)
    optavg = round(optavg,2)
    #print('            avg servers used in optimal:', optavg)

    fout.write('          bestheur avg PMs '+str(heuravg))
    fout.write('\n           optimal avg PMs '+str(optavg))
    fout.write('\n      optimal matches best '+str(samecount))
    fout.write('\n  optimal outperforms best '+str(improvementcount))
    mag=0
    if improvementAmountList:
        mag = sum(improvementAmountList)/len(improvementAmountList)
        mag = round(mag,2)
    #    print('       average magnitude of improvement:',mag,'additional servers')

    fout.write('\n       avg mag improvement '+str(mag))

#os.startfile('C:\\Users\\alext\\Desktop\\csc\\fattreeresults.txt')
```