Resource Utilization in Cloud Computing as an Optimization Problem

Ala'a Al-Shaikh, Hebatallah Khattab, Ahmad Sharieh, Azzam Sleit
Department of Computer Science
King Abdulla II School for Information Technology
University of Jordan
Amman, Jordan

***Abstract*—In this paper, an algorithm for resource utilization problem in cloud computing based on greedy method is presented. A privately-owned cloud that provides services to a huge number of users is assumed. For a given resource, hundreds or thousands of requests accumulate over time to use that resource by different users worldwide via the Internet. A prior knowledge of the requests to use that resource is also assumed. The main concern is to find the best utilization schedule for a given resource in terms of profit obtained by utilizing that resource, and the number of time slices during which the resource will be utilized. The problem is proved to be an NP-Complete problem. A greedy algorithm is proposed and analyzed in terms of its runtime complexity. The proposed solution is based on a combination of the 0/1 Knapsack problem and the activity-selection problem. The algorithm is implemented using Java. Results show good performance with a runtime complexity *O((F-S)nLogn)*.**

***Keywords*—Activity Selection; NP-Complete; Optimization Problem; Resource Utilization; 0/1 Knapsack**

I. Introduction

The term cloud computing has become a buzzword in the recent years due to the publicity and widespread of the term in all aspects of life. Cloud computing in its basic form is a model of on-demand provisioning of computing resources to users [1]. Resources such as computers, network servers, storage, applications, services, etc. are shared and reusable among users, this is referred to as Multi-tenancy [2]. Clouding has a great influence on the cost of operation of information technology (IT) infrastructure. Companies no longer need to spend on building on-premises IT departments to support their operations. Adopting the pay-as-you-go strategy, i.e. pay only for resource usage, will cut the costs of IT operations which include maintenance, employment, training, etc. In its simplest form, provisioning of resources via clouds is similar to the way of obtaining electricity from power stations without the need for everyone to establish his privately-owned station [3].

Resources lie at the heart of cloud computing. Resource utilization (pooling) is an important topic in the field of computer science, yet it is a hot research area. The need for resource utilization never stops as long resources are limited compared to the increasing demand on computers and computing. Resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand [1].

Internet plays an important role in signifying the importance of resources. The demand on the Internet and the resources are ever increasing. The advent of cloud computing encouraged companies and computer professionals to use more and more resources even if they are not available on their premises. However, this might incur fees to be paid by those users, on the other hand, service providers have to find how to best utilize their resources so as they can serve more users during specific operation time. The main idea of cloud computing is about providing (leasing) services to users. The service providers can think in leasing their services in ways that maximize their overall profit.

In this paper, a privately-owned cloud that provides services to a huge number of users is assumed. For a given resource, hundreds or thousands of requests accumulate over time to use that resource by different users worldwide via the Internet. The main concern is to find the best utilization schedule for a given resource in terms of profit obtained by utilizing that resource, and the number of time slices during which the resource will be utilized. A prior knowledge of the requests to use that resource is assumed.

The proposed algorithm, based on a greedy method, is a combination between the solutions of two different problems, the Knapsack Problem and the Activity-Selection Problem. Based on these two techniques, this utilization problem is an NP-Complete problem.

After formalizing the problem and defining it, a greedy algorithm to solve that problem is proposed. The proposed algorithm is then analyzed in terms of runtime complexity. Finally, experimental results are recorded and discussed.

The paper is organized as follows: in Section II, a sample of related work is presented. In Section III, a mathematical formulation to the problem, the proposed algorithm and a detailed discussion of algorithm design, complexity, and NP-Completeness of the problem are introduced. In Section IV, the experimental results are discussed. Finally, conclusion and future work are presented in sections V and VI.

II. Related Work

Maya Hristakeva et al [4], presented a number of methods to solve the 0/1 Knapsack problem. One of the methods presented is the greedy method. At the beginning, the 0/1 Knapsack problem is identified and formalized, then a greedy

algorithm is discussed, analyzed, and compared to other algorithms for different methods used in the research.

In [5], authors described an algorithm which generates an optimal solution for the 0/1 integer Knapsack problem on the NCUBE hypercube computer. Experimental data which supports the theoretical claims were provided for large instances of the one- and two-dimensional Knapsack problems.

In Knapsack problem, a number of items have to be chosen to fill the knapsack without exceeding its capacity so as the knapsack profit is maximized [6]. The 0-1 Knapsack Problem is formulated as follows:

- The knapsack (K) has a capacity C.
- The item (T) is a tuple $T < w, p >$, such that w is the weight of the item and p is the profit.
- The objective is as in (1):

$$Maximize \sum_{i=1}^{n} p_i x_i \;\; Subject\ to \sum_{i=1}^{n} w_i x_i \leq c$$
$$such\ that\ x_i \in \{0,1\}, i = 1, \dots, n \qquad (1)$$

In [7], authors considered a setting in which they organized one or several group activities for a group of agents. Their goal was to assign agents to activities in a desirable way. They gave a general model, then studied some existence and optimization problems related to their solutions. Their results were positive as they found desirable assignments that proved to be tractable for several restrictions of the problem.

The Weighted Activity-Selection problem is an optimization problem [8], and it is a variant of the Activity-Selection Problem. Components of the problem are as follows:

- An activity (A) is a tuple $A < s, f, p >$, such that s is the activity's start time, f is its finish time, and p is the profit of that activity.
- For an activity $A_i, s_i < f_i \; and \; p_i \geq 0$.
- Two activities A_i and A_j are said to be compatible if and only if $s_j \geq f_i$ or $s_i \geq f_j$.
- A feasible schedule (S) is a set $S \subseteq \{1, 2, \cdots, n\}$, such that every two distinct numbers in S are compatible.
- The profit (P) of a schedule (S) is $P(S) = \sum_{i \in S} p_i$.
- The objective is to find a schedule that maximizes the profit.

III. ALGORITHM

Assume a resource *R*, with a start time *S*, finish time *F*, maximum capacity *C*, and Profit per Unit of Weight *PU*. The resource R is expressed as a tuple $R < S, F, C, PU >$. The resource is connected to a network, mainly a public network like the Internet, and receives a huge number of requests. Each request Q is identified by its Id, and has a start time *S*, finish time *T*, and weight *W*. The request Q is expressed as a tuple $Q < S, F, W >$. Two requests q_i and q_j are said to be *compatible* if and only if they do not overlap, i.e. the start time of the latter must be greater than or equal to the finish time of the former.

The goal is to allocate the resource in a way that achieves *best utilization* within the following constraints:

- Maximize the profit of utilization.
- The weight of each request must not exceed the maximum capacity of the resource.
- Start and finish time of selected requests must not go beyond the boundaries of start and finish time of the resource.
- Requests must be compatible (must not overlap).

Formally, Let:

- $R < S,\ F,\ C,\ PU >$.
- Q is a set of Requests $Q = \{q_i |\ i = 1,\ 2, \dots,\ n\}$, such that $q_i < s_i, f_i, w_i >$, whereas $s_i, f_i, and\ w_i$ are the start time, finish time, and weight of request i respectively.
- $x_1, x_2, \dots, x_n$ are binary variables that indicate item selection ($x_i = 1$) or exclusion ($x_i = 0$).
- P is the total profit of utilization, W is the total weight of solution.

$$Maximize\ P = \sum_{i=1}^{n} x_i p_i\ ,\ such\ that\ W = \sum_{i=1}^{n} x_i w_i \leq C$$
$$where\ p_i = w_i(f_i - s_i) \times PU, s_i \geq S, f_i \leq F,$$
$$and\ x_i = \begin{cases} 1 & s_i \cap f_j = \emptyset\ and\ w_i \leq C \\ 0 & otherwise \end{cases},$$
$$such\ that\ S, F, C, PU, s_i, f_i, w_i, P, W, x_i, p_i \in \mathbb{N}. \qquad (2)$$

A. Explanation

Figure 1 shows the proposed algorithm. It comprises four phases, they are: (1) filtering, (2) maximum-request selection, (3) fill-right-to-max, and (4) swipe phase. Lines 7 – 11 represent the *filtering* phase. In this phase, all requests that do not meet the constraints of the resource are filtered (removed from the request array). In other words, any request with a weight exceeds the capacity of the resources, or any request that exceeds any of the boundaries of start and finish time of the resource, is filtered.

Function:	*MaxProfitSchedule()*	
Input:	ReqArr: Requests Array, C, S, F: resource capacity, start time, and finish time	
Output:	Schedule: Maximum profit schedule array	
1	left = S	C_1
2	right = F	C_2
3	available = C	C_3
4		
5	while (!ReqArr.isEmpty())	$C_4(K+1)$
6	{	
7	for each (Req in ReqArr){	$C_5K(n+1)$
8	if (Req.weight> available or Req.S< left or Req.F> right){	C_6Kn
9	remove Req from ReqArr	C_7Kn
10	}	
11	}	
12		
13	sort (non-increasing order) ReqArr by profit	$KnLogn$
14	MaxReq = ReqArr[0]	C_8K
15	add MaxReq to Schedule	C_9K
16	remove ReqArr[0] from ReqArr	$C_{10}K$
17	available = available - MaxReq.weight	$C_{11}K$
18	left = MaxReq.F	$C_{12}K$
19		
20	sort (non-decreasing order) ReqArr by start time	$KnLogn$
21	for each (Req in ReqArr){	$C_{13}K(n+1)$
22	if (Req.S>=left and Req.weight<= available){	$C_{14}Kn$
23	add Req to Schedule	$C_{15}Kn$
24	remove Req from ReqArr	$C_{16}Kn$
25	available = available - Req.weight	$C_{17}Kn$
26	left = Req.F	$C_{18}Kn$
27	}	
28	}	
29		
30	for each (Req in ReqArr){	$C_{19}K(n+1)$
31	if (Req.S>= MaxReq.F){	$C_{20}Kn$
32	remove Req from ReqArr	$C_{21}Kn$
33	}	
34	}	
35	right = MaxReq.S	$C_{22}K$
36	left = S	$C_{23}K$
37	}	
38	sort (in non-decreasing) Schedule by start time	$nLogn$
39	return Schedule	C_{24}

Fig. 1. The proposed algorithm with time complexity for each step

This step is necessary, as it minimized the size of the request array through different iterations of the selection process. Back to line 5, a while statement is used to keep on iterating until the request array is empty.

Lines 13 – 18 represent the *maximum-request selection* phase. It starts by sorting the request array in a non-increasing order by profit. This makes the maximum compatible request at the first location of the request array. As a result of the filtering phase, the first request is guaranteed to be compatible as long its weight is less than the capacity of the resource, or the remaining capacity in later iterations, and it does not exceed the boundaries of the start and finish time of the resource. In line 15, the request is added to the schedule, removed from the request array in line 16, its weight is deducted from the resource capacity in line 17, and set the new start time to the end time of that maximum request.

The third phase is the *fill-right-to-max* phase. Here, all the time slots to the right of the maximum request selected in the previous phase are filled. This phase starts in line 20 by sorting the request array in a non-decreasing order by start time of requests. Lines 21-23 iterate through all requests, pick up any request with weight less than the remaining capacity of the resource, and with a start time greater than or equal to the new left boundary. Until now, it is the finish time of the maximum request already selected in line 14. Finally, add this request in line 23 to the schedule. Similar to the previous phase, any request selected to be in the schedule: (1) is removed from the request array (line 24), (2) its weight is deducted again from the available resource capacity (line 25), and (3) its finish time is set temporarily to be the new resource start time (line 26).

Lines 30-35 signal the start of the *swipe* phase. The phase comprises iteration through the request array and removing all

requests that have start time greater than or equal to the finish time of the maximum request. These requests are still existent in the request array because they are incompatible with either the maximum request or the request to the right of it. They are removed to resize the array and start a new iteration with a fewer number of requests. In line 35, the start time of the maximum request is set to be the new resource finish time, and in line 36, the left boundary of the resource is set back again to the original start time S.

Iterations continue until there are no remaining requests in the request array, i.e. size of the request array equals zero. The iteration will stop at that point. The schedule array will be sorted in a non-decreasing order by start time in line 39, and the schedule is returned to the calling routine.

B. Analysis

All the terms that precede line 5 are constants. Line 5 introduces the term K which is the number of iterations of the outer while loop. The loop is expected to run until the request array is empty. In the worst case, the number of iterations is equal to the number of intervals of the resource ($K = F - S$). Assuming that all requests have weights less than or equal to the capacity of the resource, each with start and finish time within the boundaries of the start and finish time of the resource, and assuming a worst-case scenario in which the maximum request, i.e. the one with the highest profit, is at the end of the request array.

According to the algorithm and the assumptions aforementioned, the filtering phase will not be applicable to the initial setting, so no items will be removed from the request array. In the maximum-request selection phase, the maximum request will be added to the schedule and removed from the request array. The third phase, fill-right-to-max, is not applicable too, as long there are no requests to the right of the maximum request that has been just selected. Similarly, the swipe phase will not be applicable, because there remains no further requests right to the maximum request that are not added to the schedule. Repeating the same steps for K times, an empty array is obtained.

The sorting of an array takes $nLogn$ time, in case of using one of the sorting algorithms of logarithmic runtimes such as the merge sort. When implementing the algorithm using Java, the `Collections.sort()` method is used which has $O(nLogn)$ runtime complexity according to Java documentation [9]. Complexity of the algorithm is evaluated as follows:

$$T(n) = C_1 + C_2 + C_3 + C_4(K+1) + C_5K(n+1) + C_6Kn + C_7Kn + KnLogn + C_8K + C_9K + C_{10}K + C_{11}K + C_{12}K + KnLogn + C_{13}K(n+1) + C_{14}Kn + C_{15}Kn + C_{16}Kn + C_{17}Kn + C_{18}Kn + C_{19}K(n+1) + C_{20}Kn + C_{21}Kn + C_{22}K + C_{23}K + nLogn + C_{24}$$

$$= C_{25} + C_{26}K + C_{27}Kn + nLogn + 2KnLogn \quad (3)$$

The largest term of equation (3) is $KnLogn$, so the effort of the algorithm is $O(KnLogn)$, As mentioned earlier $K = F - S$, thus, the effort of the algorithm is expressed as $O((F - S)nLogn)$.

The value of $F - S$ in the complexity of the algorithm is arguable in the sense whether to remove it from the equation or not. In the case of cloud computing and resource utilization, time slots can be measured in seconds or in factions of seconds. If a time slot of 1 second is assumed, for a 24-hour duty for a resource is equal to 86,400 seconds (time slots), which approximates to 84K of slots. This implies that the value $F - S$ might be influential in the calculation of the complexity of the algorithm, so the complexity is expressed as $O\big((F - S)nLogn\big)$.

C. Example

Consider a privately-owned cloud with a number of resources available to users each with a capacity C and is due to service hours starting from $S = 08{:}00$ and ending in $F = 18{:}00$. The service provider charges an amount of PU as a profit per unit of weight. Assume 15 requests with capacities less than the resource capacity (C) and random profits as shown in Fig. 2 (a).

The initial schedule for the resource utilization is shown in Fig. 2 (d). When running the algorithm that is shown in Fig. 1, the following steps will be executed:

1) Step 1: Sort the requests according to their profits in a non-increasing order. The result is shown in Fig. 2 (b).

2) Step 2: Comprises the following steps:

- Add request R_{11} which has the maximum profit to the schedule and remove it from the requests array. The schedule will look like as in Fig. 2 (d) row MRS.
- Sort the remaining requests in a non-decreasing order according to their starting times as shown in Fig. 2 (c).

3) Step 3: Select a request that can be fit after R_{11} into the schedule, i.e. its start time is equal to or greater than the finish time of R_{11}. R_2 is the selected request. The result of adding R_2 into the schedule is shown in Fig. 2 (d) row FRM. Now, R_2 must be removed from the requests array. Then, any further requests' selections must be after the finish time of R_2.

4) Step 4: Repeat step-3 for each request that follows R_2, each time changing the new start time to the selected request's finish time until no further requests can be added. Each time, the selected request is removed out of the request array. After this step R_3 and R_7 will be selected to the schedule as in Fig. 2 (c), second row labelled MRS.

5) Step 4: Repeat step-1 to step-4 until no requests can be scheduled. The final schedule will be as shown in Fig. 2 (d) row Final, with a total profit of 1630.

ID	Interval	Profit
R_1	08-09	250
R_2	13-14	200
R_3	14-16	300
R_4	12-14	500
R_5	09-12	250
R_6	11-13	180
R_7	17-18	200
R_8	15-18	450
R_9	09-13	220
R_{10}	13-16	250
R_{11}	10-14	680
R_{12}	11-14	550
R_{13}	13-15	150
R_{14}	08-12	400
R_{15}	17-18	100

(a)

ID	Interval	Profit
R_{11}	10-14	680
R_{12}	11-14	550
R_4	12-14	500
R_8	15-18	450
R_{14}	08-12	400
R_3	14-16	300
R_5	09-12	250
R_{10}	13-16	250
R_1	08-09	250
R_9	09-13	220
R_7	17-18	200
R_2	13-14	200
R_6	11-13	180
R_{13}	13-15	150
R_{15}	17-18	100

(b)

ID	Interval	Profit
R_1	08-09	250
R_{14}	08-12	400
R_5	09-12	250
R_9	09-13	220
R_6	11-13	180
R_{12}	11-14	550
R_4	12-14	500
R_2	13-14	200
R_{13}	13-15	150
R_{10}	13-16	250
R_3	14-16	300
R_8	15-18	450
R_7	17-18	200
R_{15}	17-18	100

(c)

	Intervals										
Phase	8	9	10	11	12	13	14	15	16	17	18
Initial											
MRS			R_{11}								
FRM			R_{11}				R_2				
FRM			R_{11}				R_2	R_3		R_7	
Final	R_1		R_{11}				R_2	R_3		R_7	

(d)

Fig. 2. An example: (a) Array of 15 requests to use the resources. (b) Requests array after sorting it in a non-increasing order by requests weights. (c) The requests array after removing the request with the maximum profit and sorting it again in a non-decreasing order by the starting times of requests. (d) Resources Utilization Schedule; each row represents a phase: Initial: initial setting; MRS: Maximum-Request Selection Phase; FRM: Fill-Right-to-Max phase; Final: Final Resource Utilization Schedule

D. NP-Completeness

Proving the NP-Completeness of a certain problem represented in a language L is a two-step process. It involves [10]: (1) Prove that $L \in NP$, and (2) Prove that *L is NP-Hard*: if there exists a language L', such that $L' \in NPC$, and L is polynomially reducible to L' ($L' \leq_p L$).

To check that $L \in NP$, for the language L that doesn't have a polynomial-time solution, there must be an algorithm (A) that checks (verifies) a proposed solution in polynomial time. This algorithm is referred to as the certificate [11].

Figure 3 shows a list of known NP-Complete problems organized in a hierarchical way so as a problem in a lower level of the tree can be polynomially reduced to a problem in a higher level of the hierarchy.

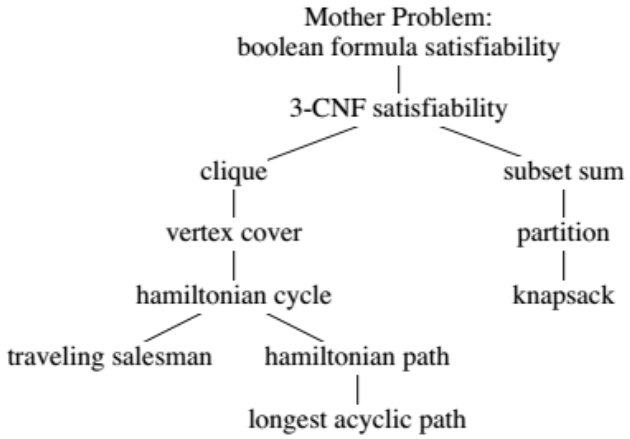

Fig. 3. A family tree of reductions [11]

Theorem 1. The resource utilization problem is an NP-Complete problem.

Proof. According to the two steps discussed earlier:

- The result of running the algorithm shown in Fig. 1 can be taken to verify that it is a solution to the resource utilization problem. An iteration through the requests in the final schedule checking that all of them are within the start and finish time of the resource working hours takes only $O(n)$ for the verification process. This means that a solution is verifiable in a polynomial time, which means that *Resource-Utilization-Problem* $\in NP$.
- To prove that the resource-allocation problem is NP-Hard, there must be a language (L') to which L can be polynomially reduced, that is the knapsack problem. To show that Knapsack-Problem $\leq_p$ Resource-Utilization-Problem, resource utilization must be casted to an instance of a knapsack problem to prove its NP-Hardness. Let the resource R be the knapsack and the capacity of the resource be the knapsack capacity. The objective is to fill the knapsack, or utilize the resources, with requests so as they do not exceed the capacity of the knapsack and the profit is maximized. It is clear that the resource-allocation problem is polynomially reducible to the knapsack problem, Knapsack-Problem $\leq_p$ Resource-Utilization-Problem, which means that the resource utilization problem is NP-Hard.

From the previous two steps, it is proven that the resources utilization problem is an NP-Complete problem, *Resource-Utilization-Problem* $\in NPC$. ■ ■

IV. Results

Tests are conducted on different datasets of sizes: 32K, 64K, 128K, 256K, 512K, 1M, 2M and 3MB. Datasets with further sizes were unable to be tested on the test PC due to memory limitations. Tests are performed on an Intel Core(TM) i5-3230M CPU with 2.60 GHz and 3 MB cache with 4 cores and 4 GB of RAM (3.86 GB is only usable). The PC runs windows 7 Enterprise edition 32-bit. The application program was written in Java. Datasets are generated by the application and saved to disk files.

Each dataset is experimented 10 times, runtime in milliseconds is recorded, and an average runtime is calculated. The parameters are set as follows: start time: 1, finish time: 86400 (number of seconds in a 24-hour period), resource capacity: 1048576, PU: 0.001. Results are shown in TABLE I.

Figure 4 shows the experimental runtimes depicted directly from TABLE I.

Figure 5 shows the chart for the asymptotic notation $O((F-S)nLogn)$, such that $S = 1$ and $F = 86{,}400$.

It is clear from both Fig. 4 and Fig. 5 that experimental and theoretical results converge. Many terms are removed from the asymptotic notation of the runtime complexity when calculated theoretically, and that explains the slight difference in shape between the two graphs.

TABLE I. Runtimes (in Milliseconds) of 10 Experiments Conducted on Different Sizes of Datasets

DS	Experiment										Average
	1	2	3	4	5	6	7	8	9	10	
64K	95.02	90.04	68.17	76.03	78.27	72.25	76.67	76.38	73.63	70.54	**77.70**
128K	256.39	205.83	204.05	197.31	200.54	199.60	208.70	204.15	199.46	201.77	**207.78**
256K	463.88	350.29	352.48	347.63	347.63	345.24	352.96	371.47	373.05	373.04	**367.77**
512K	957.36	936.62	936.57	943.66	1031.68	967.55	923.58	923.01	924.04	927.52	**947.16**
1M	2103.97	2093.14	2066.18	2126.77	2062.16	2066.81	2123.03	2079.78	2069.10	2119.20	**2091.01**
2M	4652.05	4737.74	4510.62	4931.98	4743.48	4478.73	4787.84	4678.89	4501.99	4911.64	**4693.49**
3M	9960.13	8582.96	8396.89	8766.30	8773.53	8558.79	9396.24	8603.12	9484.08	8704.05	**8922.61**

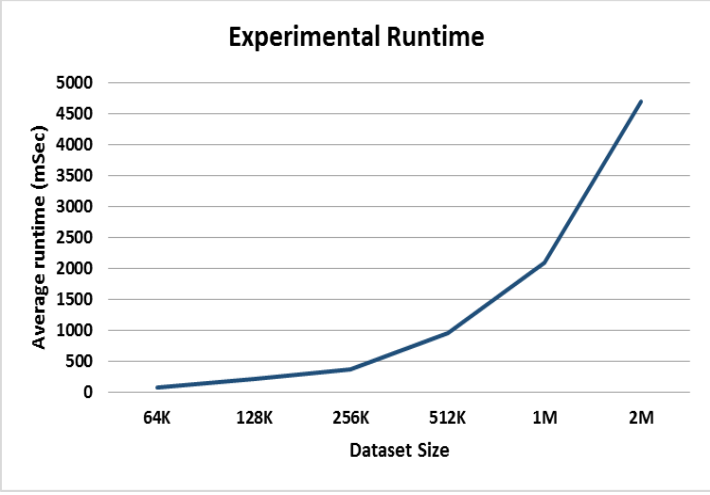

Fig. 4. Runtime chart for experimental results

Figure 6 shows the asymptotic $O(nLogn)$ complexity. To depict this graph, the same dataset sizes in the experiments need to be used, then the shapes of the graphs are compared together. This step is very important in the way it is used to prove the asymptotic notation. The controversial part in the asymptotic notation was the use of $F - S$ in the expression. Some can argue that this term is not influential in the notation. Mathematically, based on the values used above for both *S* and *F*, the difference is very high which may lead the results of comparing both notations to differ significantly.

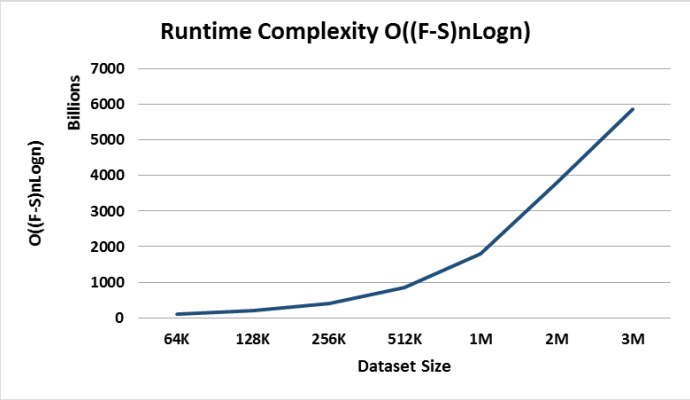

Fig. 5. Theoretical runtime graph when complexity is expressed as $O((F-s)nLogn)$

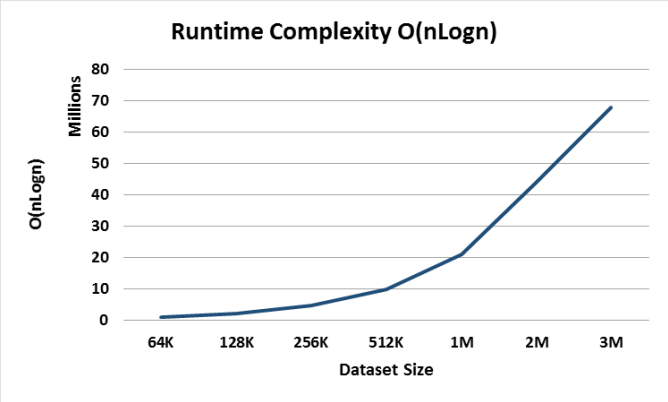

Fig. 6. Theoretical runtime graph for complexity $O(nLogn)$

From Fig. 5 and Fig. 6, it is clear that the value $F-S$ is highly influential on the overall performance of the algorithm, which means it is not possible to be removed from the runtime complexity. Thus the complexity is asymptotically expressed as $O((F-S)nLogn)$. This proves our asymptotic runtime complexity of the proposed algorithm.

V. Conclusion

In this paper, an optimization to the resources utilization problem in cloud computing is suggested. The solution is based on a combination between the 0/1 Knapsack problem and the activity-selection problem. The problem was introduced. The proposed greedy algorithm was analyzed, and then implemented using a Java program. It is proved that the problem is an NP-Complete problem. Asymptotically, the algorithm's runtime is $O((F-S)nLogn)$. Results proved the asymptotic runtime is $O((F-S)nLogn)$. An important part in that proof was whether to omit the term $F-S$ from the asymptotic notation or not by depicting two charts for the notations, one for $O(nLogn)$ and the other for $O((F-S)nLogn)$. The second notation was proved when compared to the experimental runtime results.

VI. Future Work

As a future work, the algorithm could be implemented on a supercomputer. The scheduling can be made online by using preemption to obtain better utilization and higher profits. As an addition to the currently suggested model, different pricing schemes for different periods of the working hours might be added, for example the peak time.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing, Recommendations of the National Institute of Standards and Technology," National Institute of Standards and Technology, U.S. Department of Commerce, 2011.

[2] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau and J. Xu, "Multi-Tenancy in Cloud Computing," in 2014 IEEE 8th International Symposium on Service Oriented System Engineering, 2014.

[3] Azeez, S. Perera, D. Gamage, R. Linton and P. Siriwardana, "Multi-Tenant SOA Middleware for Cloud Computing," in 3rd International Conference on Cloud Computing, Florida, 2010.

[4] M. Hristakeva and D. Shrestha, "Shrestha, Different Approaches to Solve the 0/1 Knapsack Problem," in Midwest Instruction and Computing Symposium, 2005.

[5] L. J., S. E. and S. S., "A HYPERCUBE ALGORITHM FOR THE 0/1 KNAPSACK PROBLEM," Journal of Parallel & Distributed Computing, vol. 5, no. 4, pp. 438-456, 1988.

[6] D. Pisinger, Algorithms for Knapsack Problems, PhD Thesis, Dept. of Computer Science: Univeristy of Copenhagen, 1995.

[7] D. A., E. E., K. S., L. J., S. J. and W. G., "Group Activity Selection Problem," Lecture Notes in Computer Science, vol. 7695, pp. 157-170, 2012.

[8] V. K. Patel and M. H. Pandya, "Learning of Scheduling Algorithm with Maximum Compatible Activity or Minimum Makespan," International Journal of Engineering Development and Research (IJEDR), vol. 1, no. 2, pp. 121-124, 2014.

[9] Java Documentation, "Class Collections, Java Doc," Oracle, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html. [Accessed 28 12 2015].

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 3rd ed., The MIT Press, 2009.

[11] T. H. Cormen, Algorithms Unlocked, The MIT Press, 2013.