Network Flow Heuristic Algorithm for a Distributed Web Service Selection Problem

Maliha Sultana[1], Md. Mostofa Akbar[2,3], Mushfiqur Rouf[4]
[1]*Department of Electrical and Computer Engineering, the University of British Columbia, Vancouver, BC, Canada.* [2]*PANDA Lab, Department of Computer Science, the University of Victoria, Victoria, BC, Canada.* [3]*Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.* [4]*Department of Computer Science, the University of British Columbia, Vancouver, BC, Canada.*
[1]*malihas@ece.ubc.ca* [3]*mostofa@cse.buet.ac.bd* [4]*nasarouf@cs.ubc.ca*

Abstract

In this paper a new model for a distributed web service system is presented. The proposed system is composed of multiple web service components having multiple alternative versions distributed among multiple servers. For a given set of requests an allocation is to be found that maximizes total client satisfaction subject to the resource constraints of the servers. To solve this Multidimensional Multi Knapsack Problem, which is NP hard, we propose a heuristic using a variant of the network flow maximization algorithm. Not only the heuristic is polynomial but also it inherently rules out the number of requests from contributing in time complexity of the algorithm.

1. Introduction

Nowadays all applications are moving towards the zero-installation web services based approach. A web service is an interface that describes a collection of operations that are network accessible to remote users [6]. Web service standards define how software applications should communicate and share their resources. Besides regular web applications like web mail and search engines, there are now web services based applications such as office suites (Google Docs and Spreadsheets), online antiviruses and custom information systems such as Google Earth, Google Maps, Youtube and the mashups with their APIs. To ensure the best possible service with limited resources, the resource allocation must be optimized for maximum gain. This is exactly what our proposed algorithm is designed to solve.

In this research we present a model for a distributed web service system (as shown in Fig 1) where a set of web service components, each having multiple alternative versions, are distributed among multiple servers and are accessible to the clients through a broker that takes the decision of admission or rejection of a client into the system. The requests of the admitted clients are redirected to the assigned servers, which in turn send the results back to the clients, directly or via the broker, either way the web service selection problem discussed here remains the same. The decision process is based on the maximization of total satisfaction of all clients where satisfaction implies the utility of the service provided by the web servers. Satisfaction from the service could be a function of reliability, performance, security and any other performance related attributes. Redundant allocation by using *N-Version Programming* (*NVP*) [3] is used as a means to increase individual client's satisfaction.

Our proposed problem is different from typical web service selection problems which deal with optimally selecting a set of component services for a composite service. Yu and Lin [8] presented a web service selection problem where exactly one service from each group of web services is to be selected so as to maximize a utility function subject to multiple Quality of Service constraints. They proposed two different models for the problem: *the combinatorial model* addresses the problem as a Multi-dimensional Multiple-choice 0-1 Knapsack Problem (MMKP), and *the graph model* addresses it as a multi-constraint optimal path problem (MCOP). Branch and Bound, Dynamic Programming and a combination of these two approaches can be used to solve these variants of Knapsack Problem exactly [7]. But these methods are not suitable when the search space is combinatorially explosive and thus can not be applied to real time applications and web service selection problems. Many metaheuristic algorithms have also been proposed to address the web service selection problems. Zo et. al. [9] presented a Genetic Algorithm formulation and Chang et. al. [2] presented an Evolutionary Algorithm to solve similar problems. Also, for modular software systems, a Tabu Search based metaheuristic approach is used to solve a component selection problem [1].

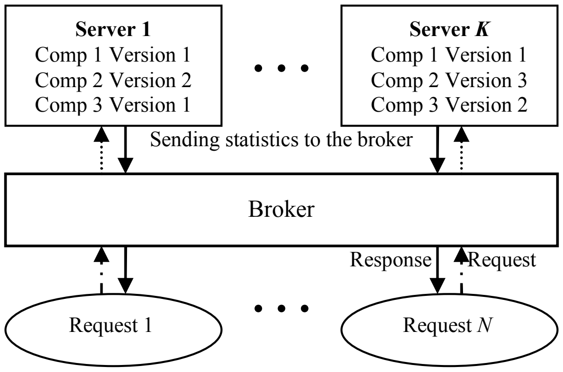

Figure 1. Typical architecture of the distributed web service system.

In this paper we consider the problem of allocating a set of components to the requesting clients. This is the scenario of a distributed web service system where the clients request for components whenever necessary. At each interval, the broker will consider the batch of requests received during that interval and will allocate appropriate versions to the clients. Due to the large search space and real time nature of our problem, neither exact algorithms nor metaheuristics is a good solution. Here we propose *Network Flow Heuristic* (NFH) using a modified form of the Push-Relabel Network Flow Maximization algorithm due to Goldberg [4][5].

The rest of the paper is organized as follows. Our model is described, along with its mathematical formulation, in Section 2. Section 3 describes NFH and its worst case complexity analysis. A comparative result analysis of NFH against a Brute Force and a Greedy algorithm is presented in Section 4. Section 5 concludes the paper.

2. Mathematical Formulation

Consider N web service *components* where component i has m_i alternative *versions*. All versions of a component perform the same task, provide different levels of satisfaction to the client and have different multidimensional resource requirements. The variation in satisfaction of these alternative versions arises from the fact that each version is developed separately and has a different set of values for measures such as reliability, performance and security. While a single version of a component suffices, providing more than one version of the same web service component to a client increases the satisfaction of the service by means of redundant allocation. However, allocating the same version of a component multiple times from multiple servers does not improve total satisfaction of a client, since these versions will fail identically. For a given set of requests an optimal set of versions is to be found that maximizes total client satisfaction subject to the resource constraints of the servers. For the formulation of the model, the following parameters are defined:

L = Max number of versions per component

C_j^i = Version j of Component i

w_{jr}^i = Amount of Resource r required *by* C_j^i

a_{kr} = Amount of Resource r available at Server k.

$$A_{qjk}^i = \begin{cases} 1, \text{if } C_i^j \text{ is assigned to Request } q \text{ at Server } k \\ 0, \text{otherwise} \end{cases}$$

$$x_{qj}^i = \sum_k A_{qjk}^i = \begin{cases} 1, \text{if } C_i^j \text{ is allocated for Request } q \\ 0, \text{otherwise} \end{cases}$$

$$y_q^i = \begin{cases} 1, \text{if Component } i \text{ is required by Request } q \\ 0, \text{otherwise} \end{cases}$$

$u_{j_1,j_2,\dots,j_L}^i$ indicates satisfaction of a combination of versions for Component i. Each of the subscripts indicates the inclusion of a version. $j_l = \{0,1\}$, implies the absence or presence of the l-th version in the combination of the versions of Component i. $u_{j_1,j_2,\dots,j_L}^i$ is a predefined parameter of the system.

Since Component i has m_i versions, $\sum_{t=m_i+1}^{L} j_t > 0$ implies $u_{j_1,j_2,\dots,j_L}^i = 0$. Actually a version j_t ($t > m_i$) does not exist. For $u_{j_1,j_2,\dots,j_L}^i > 0$, $\sum_{t=m_i+1}^{L} j_t = 0$.

Here,

$i=$	$1, 2,\dots, N$	; N	=	Number of Components
$j=$	$1, 2,\dots, m_i$	; m_i	=	Number of versions of Component i
$k=$	$1, 2,\dots, S$	; S	=	Number of Servers
$q=$	$1, 2,\dots, n$	; n	=	Number of Requests
$r=$	$1, 2,\dots, R$	; R	=	Dimension of Resources

The objective is to maximize total satisfaction of all clients served, which is computed as the sum of the satisfaction values of the assigned components. Thus, the objective function is:

$$\text{Maximize: } \sum_{q=1}^{n}\sum_{i=1}^{N} y_q^i u_{x_{q1}^i, x_{q2}^i, \dots, x_{qL}^i} .$$

Subject to the following constraints:

Constraint 1: $\forall r \forall k \left[\sum_q \sum_j w^i_{jr} A^i_{qjk} \le a_{kr} \right]$, indicating the limitations of resources in the servers.

Constraint 2: $\sum_{i=1}^{N} y^i_q = 1$, indicating that a client can request at most one component at one time.

3. Network Flow Model

First we show how the problem can be mapped on to a network flow model and then we describe the modified Push-Relabel algorithm.

3.1. Mapping of the Proposed Model

Here, resources are mapped on to flows. The versions of the web service components (C^i_j) and the servers (Srv_k) form the vertex set V. Note that the requests of the clients are not included in V and therefore complexity of NFH is independent of the number of requests.

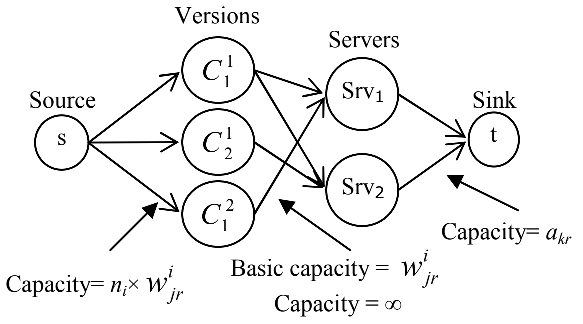

Figure 2. Network flow mapping example.

An edge from a version to server means that the version is hosted by that server. The capacity of edge (s, C^i_j) is $n_i \times w^i_{jr}$ so that Version C^i_j is allocated no more than n_i times, where n_i = number of requests for Component i. A flow from Version C^i_j to the sink via Server Srv_k means one or more instances of C^i_j at Srv_k have been allocated. Since a version can be allocated to multiple users, the flow through an edge (C^i_j, Srv_k) must be a multiple of edge's *basic capacity* w^i_{jr}. Capacity constraints are imposed on the edges (Srv_k, t) to satisfy resource constraints of the servers. An example of flow graph is presented in Figure 2.

3.2. Modified Push Relabel Algorithm

Maximization of resource consumption from each server does not always guarantee maximization of total satisfaction. So we have modified Goldberg's original network flow algorithm in order to solve our model.

In the new algorithm, we allow resources to flow through the servers in as much quantity as possible. But when the server capacity is not sufficient for a flow to be pushed to the sink, the resources are freed up by pushing back the remaining flows from the servers. This enables us to try different combinations of allocations.

Throughout this paper finding an *allocation* means computing how many versions can be executed in each of their hosting servers, given the amount of available resources at the servers. Whenever an allocation is modified in which the total satisfaction may have increased, a snapshot of the current allocation is taken. The flow algorithm ensures the maximization of resource utilization whereas the snapshots help store the best allocation among the ones considered, as shown in Figure 3.

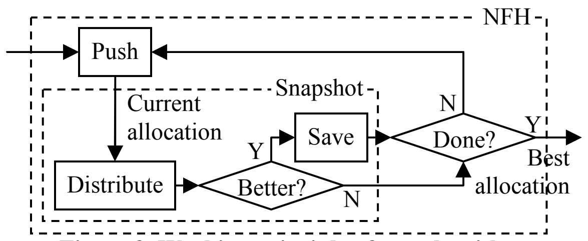

Figure 3. Working principle of our algorithm.

Initial Allocation: NFH starts from an initial allocation of components and moves towards a better allocation by performing push and relabel operations whenever possible. The initialization is done by a greedy method in an iterative fashion. Initially, the best possible version is allocated to each request. Then, if resources are still available the requests are served with a redundant version that maximizes total satisfaction. This process is continued until no more versions can be allocated to any request.

Push and Relabel Operations: As in Goldberg's algorithm, each vertex is associated with a height and a reservoir to store the excess flow. Two types of operations are defined: *push* and *relabel*. During a push, as much excess flow is pushed from a higher vertex to a lower vertex as is permitted by the corresponding edge capacity. When there is excess flow in a vertex u and all of its adjacent vertices are in a higher level, a relabel operation takes place that increases the height of u as $h(u) = 1 + \min(h(v))$,where $v \in$ adjacent(u), so that a push can now be performed from u.

At each iteration, after initialization, the lowest height vertex among all overflowing vertices is selected. Excess flow from a version node is pushed to its adjacent lowest height server. Excess flow from a server node is pushed to the sink. A push from the server to the sink implies the allocation of one or more versions that are available at that server and it is possible only when the server has sufficient resources to execute those versions.

Due to the limited capacity of the server to sink edges, all the excess flows of a server may not be pushed. Thus there is a Knapsack problem to solve at each server. We propose a greedy solution by sorting all the versions hosted by a specific server in decreasing order of their satisfaction divided by aggregate resource requirement, computed as $\sqrt{\sum_{r=1}^{R} {w_{jr}^{i}}^{2}}$. Thus when an overflowing server is pushed (that is when an allocation is found), the versions with greater satisfaction values and smaller resource requirements are considered first.

In effect, each version tries to push its excess flow (which includes the requirement of resources according to the number of requests for that version) to multiple servers, possibly in several iterations, to find a path to the sink. After each push to sink, the number of allocations of each version is updated. When the residual capacity of that server to sink edge is less than the excess flow, a backward push from the sink to a version via the server takes place in which one or more previously allocated versions are de-allocated.

Snapshot: After each push to the sink, a snapshot of the current allocation is taken. In this step, the allocated versions of each component are distributed among the requesting clients. Finding the optimal distribution of allocated versions that maximizes satisfaction will not be applicable for a real time broker. Thus a simple greedy strategy is used in our algorithm. There might be at best 2^{m_i} different combinations of m_i versions of Component *i*. In the practical scenario we can assume $n_i >> 2^{m_i}$. Therefore, instead of allocating versions to individual clients at the snapshot phase, we can simply count how many clients receive each different combination of versions. For each allocated version of Component *i*, the existing distributions are modified by including that version to maximize the total satisfaction of the clients. Our technique of maximizing total satisfaction requires calculating the additional satisfaction of the modified distribution once the new version is distributed. The complexity of this calculation is $O(2^{m_i})$. Thus the complexity of a snapshot is at most $O\left(\sum_{i=1}^{N} m_i 2^{m_i}\right)$ for all *N* components. An example of *allocation* of the versions C_1^1, C_2^1 and C_3^1 from servers, Srv_1 and Srv_2, and the corresponding *assignment* of those versions to the requesting clients are shown in Figure 4 below.

In this example, $U_{011}^{1} > U_{110}^{1} > U_{101}^{1} > U_{010}^{1} > U_{100}^{1} > U_{001}^{1}$.

Allocation			Assignment	
C_j^i	Allocation Count	From Server	Client	Served with
C_1^1	2	Srv_1	q_1	$\{C_2^1, C_3^1\}$
C_2^1	2	Srv_1	q_2	$\{C_2^1, C_3^1\}$
	1	Srv_2	q_3	$\{C_1^1, C_2^1\}$
C_3^1	2	Srv_2	q_4	$\{C_1^1\}$

Figure 4. An example allocation and corresponding assignment.

Stopping Criterion: As in Goldberg's algorithm, when a version's height becomes greater than the source's height, it is assumed that the remaining excess of that version can not be passed to sink and is returned to the source, thus completing the push operations from that version.

The height of source must be chosen in such a way that no excess should come back to the source prematurely. On the other hand, the execution time of NFH increases with the increase in source's height. We have found source height of $2|V|$ to be an optimal tradeoff. A comparative analysis of total satisfaction and execution times from experimental results for different source heights is presented in Section 4.

3.3. Worst Case Complexity Analysis

Since only source to version edge capacities vary with the number of requests, total $O(M)$ time is required to update the network flow graph for each set of requests, where $M = \sum_{i=1}^{N} m_i$. The greedy initialization procedure follows the same strategy used in snapshot and the complexity is $O\left(\sum_{i=1}^{N} m_i S 2^{m_i}\right)$.

The complexity of NFH depends on the number and complexity of the push and relabel operations which in turn depends on the height of the source, $2|V|$ in our algorithm, and the size of the network flow graph, $|V|$, where $|V|$ = M+S+2. Each push operation takes constant time. Total complexity of different types of push operations is as follows:

Source→Version Push: $O(M)$

Version→Source Push: $O(M)$

Version→Server Push: $O(M|V|S)$

Server→Sink Push (including snapshot complexity): $O\left(S|V|M\sum_{i=1}^{N} m_i 2^{m_i}\right)$

Server → Version Push: $O(S|V|M)$.

Only the version nodes or the server nodes are relabeled and the total relabel complexity is $O(SM|V|)$. Thus total worst case complexity of NFH is:

$$O\left(S(|V|)M\sum_{i=1}^{N} m_i 2^{m_i}\right) = O\left((SM^2 + S^2M)\sum_{i=1}^{N} m_i 2^{m_i}\right)$$

The worst case complexity of the algorithm as computed above is independent of *n*, the total number of requests. The exponential term 2^{m_i} cannot significantly affect the performance as more than 3 versions are rare in the practical scenario and since $n_i >> 2^{m_i}$.

4. Result Analysis

NFH is compared with a Brute Force (BF) algorithm and a Greedy algorithm. All these algorithms were implemented and tested against randomly generated datasets. In the Brute Force algorithm the optimal solution is achieved through exhaustive search. The inherent idea of the Greedy algorithm is to maximize satisfaction by serving more requests and allowing redundancy when the total number of accepted requests can no more be increased. Because of the large search space, Brute Force algorithm takes minutes to compute the results for even small unrealistic datasets and is unable to produce results for even moderate size of data sets. Figure 5 presents the experimental results for achieved total satisfaction using these algorithms. The networks are randomly generated, with at most 15 requests, 15 versions in total and 5 servers.

For larger data sets NFH is compared with the Greedy algorithm only. Figure 6 shows that in most of the cases satisfaction obtained from NFH are almost 20% better than that of the Greedy algorithm. Each point in the figure is the average of 50 randomly generated datasets.

The most attractive part of our algorithm is that its worst case complexity does not depend on the number of requests. Figure 7 shows the execution times of NFH for three different network sizes. The algorithm has been run on a Personal Computer having Intel Pentium 4 processor and 512 Mb of RAM. With increasing number of requests, execution time of the algorithm increases until the total number of push operations reaches its maximum and after that point the execution time does not increase with additional requests.

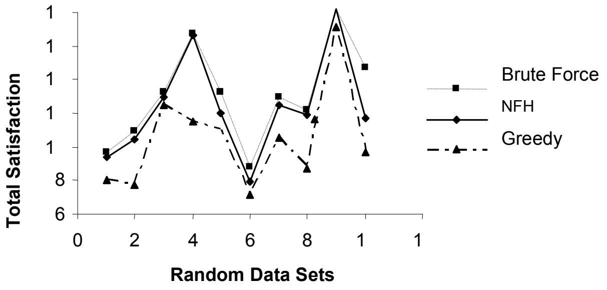

Figure 5. Comparison of Total Satisfaction achieved by different algorithms.

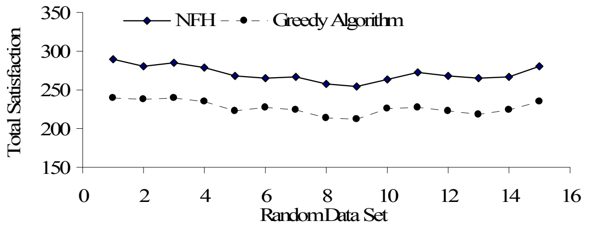

Figure 6. Comparison of Satisfaction for larger data sets.

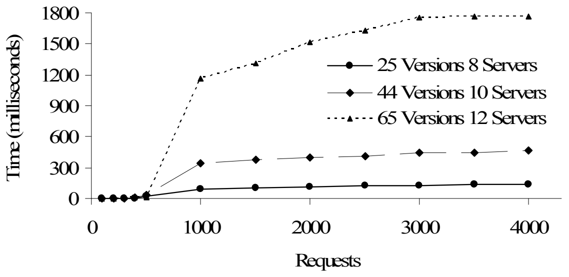

Figure 7. Execution times of NFH for different network sizes and requests.

With the increasing height of the source, NFH explores more options and provides better results but at the cost of increasing execution times. The algorithm is run for four different source heights and Figure 8 shows the percentage increase in Total Satisfaction with respect to source height. Figure 9 shows how the execution time of NFH varies with increasing source heights. It is evident that percentage increase in satisfaction is very low as compared to the increase in execution time for source heights greater than $2|V|$.

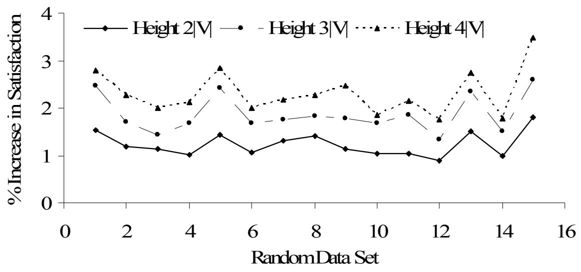

Figure 8. Percentage increase in satisfaction of NFH with increasing source heights.

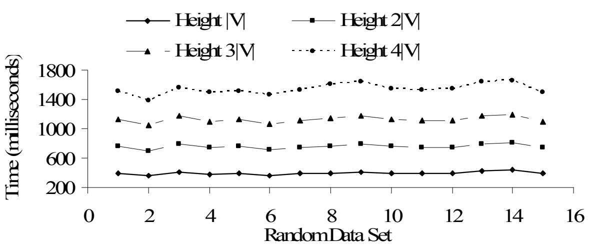

Figure 9. Execution times of NFH for four source heights.

5. Conclusion and Future Works

The network flow approach presented here not only finds the solution in polynomial time, but also is independent of the number of requests, the varying parameter of the system. The proposed algorithm can be implemented in any system that incorporates distributed resources, for example, a server firm. The distributed system considered here is controlled centrally. Our algorithm can be extended for a totally distributed one, where all the servers can communicate with each other thus omitting the role of the broker.

References

[1] M. Caserta and A.M. Uribe. Tabu search-based metaheuristic algorithm for software system reliability problems, Computers and Operations Research, In press.

[2] W.C. Chang, C.S. Wu and C. Chang. Optimizing dynamic web service component composition, Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (2005).

[3] L. Chen and A. Avizienis. N-Version Programming: a fault tolerance approach to reliability of software operation, IEEE Proceedings of FTCS-25 3 (1996) 113-119.

[4] A.V. Goldberg, E. Tardos and R.E. Tarjan. Network flow algorithms, Algorithms and Combinatorics 9 (1990) 100-164.

[5] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem, Proceedings of the 18th ACM STOC (1986) 136-146.

[6] H. Kreger. Fulfilling the web services promise, Communications of the ACM 46 (2003) 29-34.

[7] S. Martello, D. Pisinger and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem, European Journal of Operational Research 123 (2000) 325-332.

[8] T. Yu and K.J. Lin. Service selection algorithms for composing complex services with multiple QoS constraints, Lecture Notes in Computer Science 3826 (2005) 130–143.

[9] H. Zo, D.L. Nazareth and H.K. Jain. Measuring reliability of applications composed of web services, Proceedings of the 40th Annual Hawaii International Conference on System Sciences (2007) 278-287.