The Longest Link Node Deployment Problem in Cloud Computing: a Heuristic Approach[*]

Matheus S. Ataíde, Cid C. de Souza, Pedro J. de Rezende
Institute of Computing – University of Campinas (UNICAMP)
{*matheus.ataide@students.,cid@,rezende@*}*ic.unicamp.br*

Marcos A. Vaz Salles
Department of Computer Science – University of Copenhagen (DIKU)
vmarcos@di.ku.dk

Abstract

In this paper, we propose a GRASP metaheuristic for the *longest link node deployment problem* (LLNDP). The LLNDP is $\mathbb{NP}$-hard and models a machine assignment problem related to the optimal usage of resources in a cloud computing environment. In a previous work [10], greedy and randomized heuristics were proposed for the LLNDP. Computational tests carried out here on 450 instances of various sizes and from three different classes revealed that our GRASP outperformed those heuristics with an average gain of 14.2% in the solution cost. An in-depth discussion of the results is presented.

Keywords: cloud computing; machine assignment optimization; GRASP metaheuristic.

1 Introduction

The cloud has emerged as a new paradigm for deployment of computing applications [2]. With cloud computing, application developers and users can rent computational resources flexibly, paying per hour of allocated resources or per usage of services offered by the cloud provider. A popular model of cloud computing is the one of infrastructure-as-a-service, in which users commonly allocate virtual machines – denoted here VMs – as resources for their computing needs.

When allocating a number of VMs in the cloud, users are given the illusion of a virtual cluster, in which any of the VMs can communicate with any other through standard networking protocols. Unlike with a physical cluster, however, it has been observed that virtual clusters in the cloud suffer from a much higher degree of network latency heterogeneity [9]. In other words, mean latencies for message passing between any pair of VMs in a physical cluster are expected to be very similar; however, these mean latencies in virtual clusters allocated in popular cloud computing platforms, such as Amazon Web Services (AWS), Rackspace, or Google Compute Engine, are different in many cases by a factor of two or more [10]. In addition, the observed differences in mean latencies persist over long periods of time, ranging from days to weeks.

[*]This research was supported by grants from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) #311140/2014-9, #477692/2012-5; #304727/2014-8; Fundação de Amparo à Pesquisa do Estado de São Paulo (Fapesp) #2015/08438-0, #2015/23270-9; FAEPEX/UNICAMP; and an AWS in Education research grant award.

High-performance computing applications are particularly vulnerable to latency heterogeneity in the cloud [4, 11]. The communication pattern in a number of these applications can be abstracted by the bulk-synchronous processes model [8], characterized by supersteps of computation by independent processes, referred to in this paper as *application nodes*, interspersed with barrier synchronization events where data exchange between processes may occur. As the degree of parallelism increases, the efficiency of applications following such a communication pattern relies on low cost communication during barriers. However, since a barrier implies synchronization across all application nodes, either directly or transitively across supersteps, the longest delay in communication between any two application nodes is a prime determinant of performance.

Suppose we model the communication resources between any two VMs in the cloud as a *link* with an associated cost. Now, the problem we are faced with is to deploy the application nodes to corresponding VMs in the cloud in a way that the cost of communication between application nodes arranged in a given topology is minimized. Since this communication cost is determined by the cost of the longest link used, this problem is called the *Longest Link Node Deployment Problem* (LLNDP) [9]. Previous work has shown that LLNDP is $\mathbb{NP}$-hard, and that solutions based on enumeration of random deployments can achieve quality competitive with both mixed-integer and constraint programming algorithms [10]. Besides, the quality achieved by randomization surpasses that of greedy heuristics, though the latter spend much less wall-clock time to produce solutions.

Our contribution In this paper, we observe that the search of a randomized approach can be informed by the greedy heuristics proposed for LLNDP via an instantiation of the *Greedy Randomized Adaptive Search Procedure* (GRASP) metaheuristic [6, 7]. We make the following contributions:

1. We propose a solution method based on GRASP that randomizes the best previous greedy heuristic for the problem, complemented by novel local search schemes.
2. In a set of experiments, we show that our GRASP-based solution method achieves deployments with gains in cost of 14.2% on average compared with the randomized approach over a set of benchmark instances generated based on latency measurements provided by the authors of the previous work proposing the LLNDP [9, 10]. The gains are particularly expressive for application nodes organized in mesh topologies, a pattern very common in high-performance computing applications.

Organization of the text Section 2 formalizes the LLNDP and introduces relevant notation for the paper. In Section 3, we present our GRASP-based solution method, moving on to evaluate it experimentally in Section 4. Finally, Section 5 concludes by outlining avenues for future work.

2 Problem Formulation and Modeling

To formalize the description of the *Longest Link Node Deployment Problem* (LLNDP), we regard the application nodes as vertices of a graph – the *Application Graph* – whose directed edges are the expected communication links of the desired topology. On the other hand, allocated VMs in the cloud may similarly be viewed as the vertices of a complete graph – the *Cloud Graph* – whose directed edges are arcs connecting each pair of these vertices, and whose weights are the predetermined latencies for message passing between the particular (ordered) pair of VMs that those two vertices represent.

In this framework, we can regard the LLNDP as a special case of the minimum weight subgraph isomorphism problem (MWSIP) [5] that may be posed as follows. Given a graph H with k vertices and a graph $G = (V_G, E_G, w)$ with weight function $w : E_G \to \mathbb{R}$, an *H-subgraph of G* is a subgraph of G isomorphic to H. The MWSIP seeks to determine a minimally weighted H-subgraph of G (or to establish that one does not exist). To model the LLNDP as an MWSIP, the weight of an H-subgraph H' of G is defined as the weight of the heaviest edge of H'. Besides, notice that a solution always exists as long as $|V_G| \geq k$, since, in our context, G is always a complete graph. Also, it is often the case that an instance of the LLNDP involves a Cloud Graph with excess vertices when compared to the Application Graph. As discussed in Section 4, for the experiments described therein, we set this overload at 10% to provide room for a broader search for a good deployment of the Application Graph within the Cloud Graph.

Problem Formulation The LLNDP may be stated as: given a (weighted) Cloud Graph $G = (V_G, E_G, w)$ and an Application Graph H, we wish to determine an H-subgraph $H' = (V_{H'}, E_{H'})$ of G whose heaviest edge $e \in E'_H$ has the least weight amongst all H-subgraphs of G.

3 A GRASP for LLNDP

Greedy Randomized Adaptive Search Procedure (GRASP) [7] is a metaheuristic extensively employed to solve combinatorial optimization problems as it generates high quality solutions very efficiently [6]. The nature of LLNDP makes it a prime candidate for applying this strategy when seeking high grade solutions in a short computing time.

Due to space constraints, we refer the reader unfamiliar with the workings of a basic GRASP (see Algorithm 1) to [6, 7]. In a nutshell, in the construction phase (see Algorithm 2), a restricted candidate list (RCL) of quasi-greedy choices is built from which a suboptimal solution is generated. One such viable solution needs not be optimal even within simple local neighborhoods. This is where the local search phase (see Algorithm 3) comes in: by successively superseding the current solution with a better one from within a small neighborhood, a local minimum is obtained.

We now describe the specifics of the use of GRASP in the context of the LLNDP, but we limit ourselves to outlining only the construction and the local search phases.

Construction Phase Our choice for a greedy strategy in which the construction phase is based was the one that generated, in practice, the best solutions for the LLNDP among those compared in [10] (see Algorithm 4). For a detailed description of this algorithm, the reader is referred to Zou et al.'s algorithm G2 in [10]. Now, the essence of the *Greedy Randomized Construction* lies on how the Restricted Candidate List (RCL) is built and what is the role of its α parameter. Consider that the greedy choice encapsulated in lines 7 to 15 of Algorithm 4 leads to a single option for extending the mapping $\mathcal{D}$ one additional vertex at a time. By broadening this choice to within a more diverse set of vertices selected in a nearly greedy fashion, controlled by the α parameter, we expect to lower the ultimate cost of the mapping being composed. While this outcome is not guaranteed, the iterated nature of the GRASP algorithm makes this prediction all the more likely.

Of course, the most encompassing candidate set to be considered when forming the RCL is the set of all vertices yet unmapped. If we limit the choices among these to the $\alpha\%$ that add the least cost to the mapping, we are able to introduce flexibility within reasonable bounds. However, too broad a spectrum ($\alpha = 100\%$) leads to a completely random behavior, while in the opposite direction we end up with an entirely greedy method. To assess the impact of various values for α,

Algorithm 1: Generic GRASP

```
while termination conditions not met do
    $\mathcal{D} \leftarrow$ GreedyRandomizedConstruction;
    LocalSearch($\mathcal{D}$);
    MemorizeBestSolutionSoFar($\mathcal{D}$);
return best solution found.
```

Algorithm 2: GreedyRandomizedConstruction

```
Output: Viable Solution
$\mathcal{D} = \emptyset$;
Initialize the candidate set $C$ and $w(e)\ \forall e \in C$;
while $C \neq \emptyset$ do
    Build restricted candidate list: $\mathrm{RCL}_\alpha$;
    $\mathcal{D} = \mathcal{D} \cup$ PickRandomCandidate($\mathrm{RCL}_\alpha$);
    Update set $C$ and $w(e)\ \forall e \in C$;
return $\mathcal{D}$.
```

Algorithm 3: LocalSearch

```
Input: Viable Solution $\mathcal{D}$
Output: Improved Viable Solution
while $\mathcal{D}$ is not locally optimal do
    Find $\mathcal{D}' \in N(\mathcal{D})$ with lower cost than $\mathcal{D}$;
    $\mathcal{D} = \mathcal{D}'$;
return $\mathcal{D}$.
```

Algorithm 4: Greedy for LLNDP

```
Input: Cloud Graph $G = (V_G, E_G, w)$,
       Application Graph $H = (V_H, E_H)$
Output: Mapping $\mathcal{D} : V_H \to V_G$
Find $(u_0, v_0) \in E_G$ of least $w((u_0, v_0))$;
Choose an arbitrary edge $(x, y) \in E_H$;
$\mathcal{D}(x) = u_0$, $\mathcal{D}(y) = v_0$;
for $i = 1$ to $|V_H| - 2$ do
    $c_{\min} = \infty$;
    for $(u, v) \in E_G$ do
        for $t$ s.t. $(\mathcal{D}^{-1}(u), t) \in E_H$ do
            $c_{uv} = w((u, v))$;
            for $(t, x) \in E_H$ do
                if $\mathcal{D}(x)$ is defined and
                  $w((v, \mathcal{D}(x))) > c_{uv}$ then
                    $c_{uv} = w((v, \mathcal{D}(x)))$;
            if $c_{uv} < c_{\min}$ then
                $c_{\min} = c_{uv}$;
                $v_{\min} = v, t_{\min} = t$;
    $\mathcal{D}(t_{\min}) = v_{\min}$;
    return $\mathcal{D}$.
```

Figure 1: GRASP, GRASP's Construction Phase, GRASP's Local Search, LLNDP Greedy Algorithms

we developed five versions of the GRASP algorithm for $\alpha \in \{1, 2, 5, 10, 20\}$.

Local Search At the core of the local search phase lies the concept of neighborhood of a viable solution. Firstly, notice that interchanging the images of any two mapped vertices of a viable solution leads to a new mapping that is equally feasible. Moreover, since the Cloud Graph is a complete graph, our choices for remapping are as large as the allocation overload. These are, in essence, the two schemes we contrived to establish neighborhoods within which the local search navigates while seeking a local minimum.

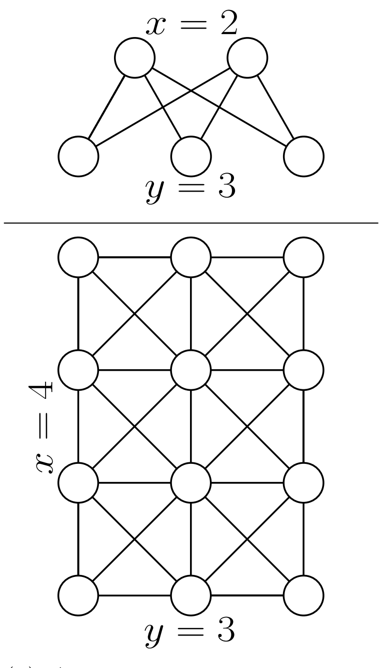
While a thorough search can lead to a *best-improvement* type scrutiny and a lazier *first-improvement* analysis strategy is certainly much more efficient, our experiments showed that the quality gain of the former is immaterial compared to that of the latter. Therefore, we saw it best to lean towards higher efficiency in all further experiments, as reported in the next section.

4 Computational Experiments

In this section, we describe the computational experiments that were carried out, reporting and analyzing the results obtained for a collection of 450 instances of various sizes belonging to the three different classes discussed in [10].

Environment All experiments were run on a desktop PC equipped with an Intel® Core™ `i7-2600` processor at 3.40 GHz, 8 GB of RAM and running `Ubuntu 12.04.5 LTS (GNU/Linux 3.2.0-31-generic x86_64)`. The coding was done in `C++` and the programs compiled using `g++`. Furthermore, all tests were performed without other processes running concurrently on the machine. Each algorithm was allowed to run for 4 minutes on each instance.

Instances Recall that an instance of the LLNDP is comprised of two graphs: the Cloud Graph and the Application Graph (cf. Section 2). The former is a weighted complete graph in which edge weights are the latencies between pairs of virtual machines in the cloud. Actual latency values were measured between 110 virtual machines on the EC2 service of AWS [1] by Dr. Tao Zou, one of the developers of ClouDiA, and kindly made available to us in the form of a 110×110 matrix L. To generate the edge weights of a cloud graph on $n < 110$ vertices, we selected, at random, n row and n column indices and assembled the corresponding $n\times n$ square submatrix of L. On the other hand, the Application Graphs were generated within the three graph classes reported in [10] as being amongst the most common in practice, namely, complete bipartite graphs, trees and meshes (see Figure 2a).



(a) A complete bipartite graph for $n = 5$ (top) and a mesh graph for $n = 12$.

```
Procedure GenerateTreeInstance($n$)
1  $E \leftarrow \emptyset$, $V \leftarrow \{1,\dots,n\}$
2  For $j$ from 2 to $n$
3      $i \leftarrow$ Random_Integer$[1, j-1]$
4      $E \leftarrow E \cup \{(i,j)\}$
5  Return $(V, E)$
```

Parameters for Generating Complete Bipartite Graphs ($n = x + y, x = (n/2) - j$)

n	group index (j)	# instances per group
$= 20$	$j = 0,\dots,5$	5
> 20	$j = 0,\dots,9$	3

Parameters for Generating Mesh Graphs

n	# instances per group	$(x \times y)$ groups
20	30	(4×5)
40	$15, 15$	(4×10), (5×8)
60	$10, 10, 10$	(4×15), (5×12), (6×10)
80	$10, 10, 10$	(4×20), (5×16), (8×10)
96	$10, 10, 10$	(4×24), (6×16), (8×12)

(b) Algorithms and parameters to generate random instances.

Figure 2: Instance generation.

In each of theses classes, 30 instances with 20, 40, 60, 80 and 96 vertices were created. The *overload* we used was fixed at 10%, meaning that the size of the Cloud Graphs for these instances were 22, 44, 66, 88 and 106, respectively. The procedure to create random trees as well as the parameters used in the generation of meshes and complete bipartite random graphs are shown in Figure 2b. For mesh graphs, the parameters x and y in the table denote the number of rows and columns of vertices, respectively, whereas for complete bipartite graphs, these parameters correspond to the cardinality of the two vertex subsets.

Analysis The experiments reported here aim to assess the following aspects: *(i)* the impact of the α parameter in the construction phase of GRASP; *(ii)* the improvement on GRASP's performance resulting from the inclusion of a local search procedure and, *(iii)* the increase, if any, in solution quality compared to the procedure R2 from [10], reported there as yielding the best solutions among all other heuristic alternatives considered.

Five implementations of GRASP were tested with different values for the α parameter in the construction phase: 1, 2, 5, 10 and 20%. When $\alpha = 1\%$ the algorithm behaves essentially like a purely greedy procedure and for values of α larger than 20, we observed a clear deterioration in the quality of the solutions. The different GRASP implementations and the random solution procedure R2 were executed for 4 minutes on all 450 instances in our benchmark. For each of them, the *average rank* of all six algorithms was computed. Recall that, when a tie occurs among k algorithms for the i-th rank, the average rank of all of them is set to $\sum_{j=0}^{k-1}(i+j)/k$ (e.g., when a tie happens between two algorithms in the first rank, the average rank for both is 1.5). The means of the average ranks of GRASP$_\alpha$, for $\alpha \in \{1, 2, 5, 10, 20\}$ (in percentage), and R2 are given below:

Algorithm	GRASP$_1$	GRASP$_2$	GRASP$_5$	GRASP$_{10}$	GRASP$_{20}$	R2
Average Rank	3.371	3.112	2.862	2.677	3.206	5.772

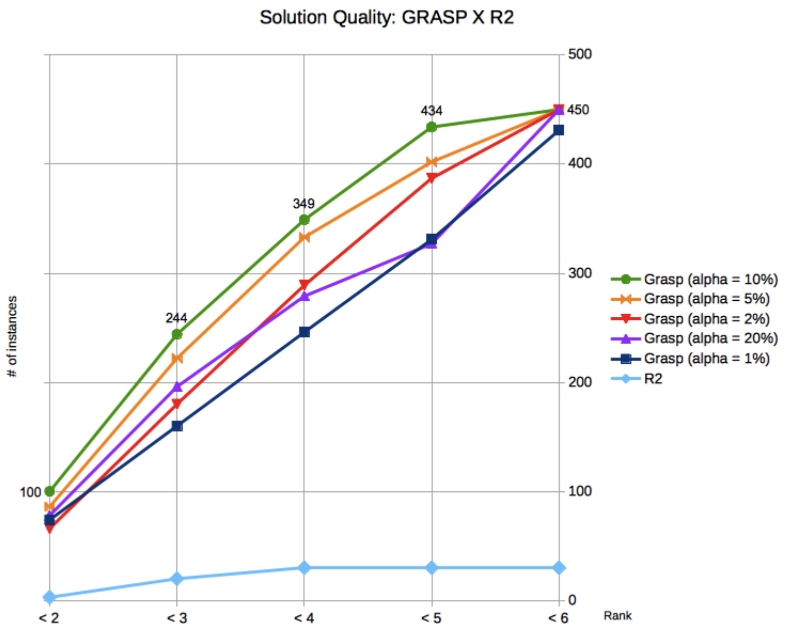


Figure 3: Comparison of GRASP algorithms and R2.

From these means, one sees that GRASP$_{10}$ performed better than all remaining GRASPs and that R2 is clearly lagging behind. This is confirmed by the graph in Figure 3 in which we show the number of instances for which each algorithm was ranked below $k = 2, \ldots, 6$. In fact, this graph reveals that, except for GRASP$_{10}$ and GRASP$_5$, the algorithms had quite distinct performances and could be ordered, from best to worst, in the following order: GRASP$_{10}$, GRASP$_5$, GRASP$_2$, GRASP$_{20}$, GRASP$_1$ and R2. To further investigate this issue, we ran statistical tests to compare the algorithms according to the methodology suggested in [3]. Since we have more than two algorithms, the (non-parametric) Friedman test was used. As one would expect from the results, the null hypothesis (stating that all algorithms perform equally well) was rejected, for a confidence level of 95%. The post-hoc Nemenyi test was then executed and, for the same confidence level, resulted in a *critical difference* of 0.355. If the absolute values of the differences between the mean rank averages of two algorithms is below the critical difference, their performances are considered similar. The Nemenyi test left R2 in an isolated group confirming its poor performance when compared to the GRASP algorithms. Also, according to the test, only GRASP$_5$ could be grouped with GRASP$_{10}$, since their average rank difference was 0.19. However, statistically, GRASP$_2$ and GRASP$_{20}$ were not identified by the test as being significantly different from GRASP$_5$, with the observed differences relative to the latter of 0.25 and 0.34, respectively. The analysis of the Nemenyi test, together with that of Figure 3, led us to run the statistically stronger Wilcoxon Signed-Rank test to evaluate the performances of GRASP$_5$ and GRASP$_{10}$, once again at 95% confidence level. The conclusion of the test was that none of the two algorithms outperforms the other in statistical terms. We then decided to continue our tests using GRASP$_{10}$, since Figure 3 suggests that it produces better solutions in general.

	%Gain R2	%LS Gain	%Iter2Best
≤ 0	29 (29 / 0 / 0)	30 (11 / 0 / 19)	0 (0 / 0 / 0)
Average	14.2 (5.3 / 19.7 / 17.5)	3.1 (1.9 / 5.0 / 2.3)	37.9 (31.8 / 46.5 / 35.5)
Max	27.0 (12.3 / 26.6 / 27.0)	12.3 (5.9 / 12.3 / 6.0)	99.9 (98.3 / 99.9 / 99.3)
Stdev	8.5 (3.0 / 6.4 / 6.8)	2.3 (1.2 / 2.6 / 1.3)	31.0 (31.0 / 28.6 / 31.6)
Min	-0.8 (-0.8 / 2.4 / 2.3)	0.0 (0.0 / 1.1 / 0.0)	0.0 (0.0 / 0.2 / 0.0)

Table 1: GRASP$_{10}$: summary of results.

Table 1 summarizes the results for GRASP$_{10}$. All values are given as percentages. The cells in column "%Gain R2" refer to improvements yielded by GRASP$_{10}$ relative to R2. Those in column "%LS Gain" represent the portion of the previous improvement that was obtained exclusively by the local search phase. Finally, the third column displays the percentage of the total number of iterations that had been executed when the best solution was found by GRASP$_{10}$. The triplet of values in parentheses denote the same statistics computed individually for each class of Application Graphs, namely, bipartite graphs, mesh graphs and trees, in this order.

Consider the column %Gain R2 in Table 1. In only 29 instances R2 produced a solution no worse than that of GRASP$_{10}$. All of these instances correspond to graphs of 20 vertices (the smallest ones), and belong to the bipartite class. This suggests that the bipartite graphs are harder to optimize. That claim is supported by the fact that the average (max and min) improvements are always smaller for that class than for the others. Let us examine the third column (%LS Gain). There, we see that only 30 instances the best solution found by GRASP$_{10}$ did not benefit from the local search phase. Roughly one fourth (3.1%) of the overall average improvement (14.2%) was obtained by the local search (relative to the construction phase). However, the local search was more important for meshes than for bipartite graphs or trees. Finally, considering the last column of Table 1, we see that, on average, only 37.9% of the total number of iterations executed within the 4 minutes allotted for computation were needed to reach the best solution. As each iteration takes about the same running time, if normal distribution is assumed, about 83% of the instances would have their best solution found after 68.9% $(37.9 + 31.0)$ of the 4 minutes (≈ 2.75 minutes) had passed. The results revealed that this actually happened for 79% (357) of the instances, which is not too far from the above rough estimation. This shows that the time limit we imposed for the tests was appropriate, at least for the classes of graphs and sizes in our benchmark. On the other hand, the average gain relative to R2 obtained for bipartite instances on 96 vertices is 6.8%, less than the average (14.2%). In this case, GRASP$_{10}$ executed no more than 245 iterations. Nonetheless, in many successful GRASP implementations reported in the literature, the total number of iterations is at least three times larger than that.

5 Conclusions

We developed here a GRASP heuristic for the LLNDP. Our tests showed that significant gains can be achieved when compared to the random procedure R2, pointed out in [10] as a very good alternative to generate high quality solutions for the problem. Despite these encouraging results, further developments are expected to improve the performance of GRASP. Among those, the implementation of a reactive GRASP and of a *path relinking* procedure are planned. Moreover, we envision tackling in the near future the *longest path node deployment problem*, also discussed in [10], using GRASP.

References

[1] Amazon web services. Elastic compute cloud (EC2). `https://aws.amazon.com/ec2/`.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Electrical Eng. and Comp. Sciences, University of California at Berkeley, 2009.

[3] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, Dec. 2006.

[4] C. Evangelinos and C. N. Hill. Cloud computing for parallel scientific HPC applications. In *Proc. First Workshop on Cloud Computing and its Applications (CCA)*, Chicago, IL, October 2008.

[5] J. Plehn and B. Voigt. Finding minimally weighted subgraphs. In *Graph-Theoretic Concepts in Computer Science: Proceedings of the 16th International Workshop WG*, pages 18–29. Springer Berlin, 1991.

[6] M. G. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *Int. Series in Operations Research & Management Science*, pages 283–319. Springer US, 2010.

[7] M. G. Resende and C. C. Ribeiro. GRASP: Greedy Randomized Adaptive Search Procedures. In E. K. Burke and G. Kendall, editors, *Search Methodologies*, pages 287–312. Springer US, 2014.

[8] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[9] T. Zou, R. L. Bras, M. A. V. Salles, A. J. Demers, and J. Gehrke. ClouDiA: A deployment advisor for public clouds. *PVLDB*, 6(2):109–120, 2012.

[10] T. Zou, R. Le Bras, M. Salles, A. Demers, and J. Gehrke. ClouDiA: A deployment advisor for public clouds. *The VLDB Journal*, 24(5):633–653, 2015.

[11] T. Zou, G. Wang, M. V. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White. Making time-stepped applications tick in the cloud. In *Proc. SOCC*, pages 20:1–20:14, 2011.