A Linear Programming Approach for Optimizing Workload Distribution in a Cloud

Vadym Borovskiy, Johannes Wust, Christian Schwarz, Alexander Zeier
Hasso-Plattner-Institut, Potsdam, Germany
{*vadym.borovskiy, johannes.wust, christian.schwarz, alexander.zeier*}*@hpi.uni-potsdam.de*

Wolfgang Koch
SAP AG, Walldorf, Germany
wolfgang.koch@sap.com

***Abstract*—Cloud computing's usage-based pricing model creates an incentive for subscribers to optimize the utilization of the rented resources. The goal of the current work is to devise a formal approach for distributing workload among a minimum number of servers. The paper models this problem as a set partitioning problem and describes two solution approaches. The first one generates a set of candidate blocks and then composes an optimal partition by solving an integer programming problem. The second approach solves the set partitioning problem with column generation technique. Both methods were implemented and evaluated. The experiment results led to a conclusion that the second approach delivers the best results.**

***Keywords*-Workload distribution; Set partitioning; Column generation**

I. Introduction

Cloud computing continues to gain momentum due to its ability to provide on-demand computing resources in both an economically and computationally efficient manner. The economic benefit of cloud computing from a provider's point of view comes from economies of scale [1]. The more resources a data center has, the lower the cost of individual resource. From a resource consumer's point of view, the benefit derives from converting the fixed cost of owning and maintaining on-premise infrastructure into the variable cost of renting it on demand. On average, on-premise infrastructure is underutilized, because its capacity is driven by the system's peak load. But peak loads only account for a small part of a systems' operating time. Companies make large investments in their infrastructure only to find it idle for a majority of the time. By subscribing to cloud services companies pay only for the resources they actually use, whereas with on-premise hardware, the amount of resources they pay for is driven by peak workload [2].

Cloud computing's usage-based pricing model creates an incentive for subscribers to optimize the utilization of the rented resources. This is especially relevant for multi-module systems, because of many possible deployment options. Selecting a particular number of servers and the distribution of the system's modules among the servers produces visible effects. If few modules are installed on each server, the overall number of servers is bigger than absolutely necessary, which implies extra cost. On the other hand, when too many modules are deployed on each server, bottlenecks appear, which implies lower throughput and lower quality of service. Thus, by choosing a proper deployment configuration subscribers can [1]: (i) avoid resource over-provisioning; (ii) maintain the desired quality of service in the face of increasing workload by provisioning on-the-fly additional resources.

The paper is structured as follows. Section III presents a formal model of the workload distribution problem. Section IV describes a straightforward solution procedure based on the suggested model. Section V describes how a column generation technique can improve the solution procedure. Section VI presents computational results of the suggested algorithms. Section VII concludes the paper.

II. Related Work

Even though load balancing has received much attention in the research community [3], [4], [5], [6], no conventional techniques can be applied to the discussed problem. A fundamental obstacle limiting the applicability of the conventional techniques is the violation of the requirement that any server in a cluster can handle any request coming from any client (i.e., the servers must be interchangeable). In our case servers are not interchangeble, because they perform different tasks (i.e., run different modules). The lack of existing approaches motivated us to apply knowledge from other areas. In particular, our work has been inspired by research in airline crew scheduling [7], [8], where the set partitioning approach has been successfully applied for resource allocation problems. With regards, to column generation as a method of dealing with large linear programs, many researchers have observed that it is a very powerful technique for solving a wide range of industrial problems to an optimum or to a near optimum. Ford and Fulkerson, for example, suggested column generation in the context of a multi-commodity network flow problem [9]. Gilmore and Gomory then demonstrated its effectiveness in a cutting stock problem [10]. More recently, vehicle routing, crew scheduling, and other integer-constrained problems were successfully solved with column generation [11].

III. Mathematical Model of the Problem

The goal of the current work is to devise a formal approach for distributing modules of a system among a number of servers. Given the workload of each module, we want to assign it to one of the available servers with given

 ISBN: 978-1-61208-153-3

capacity. The workload of a module and the capacity of a server must be measured in the same units that represent the amount of a resource consumed or provided. The resource can be, for example, CPU time, memory, storage or network bandwidth. Measuring the exact workload of a module may be impossible, due to its dynamic nature. However, we believe that with the help of profiling tools, a reasonably precise workload estimate is feasible to obtain.

A simpler way of figuring out the workload of a module is to measure it relatively to the capacity of a server. For that, install multiple instances of a module on the same server as long as the service level of each module satisfies requirements. If a server can handle four instances of a module, then the module's workload is 25% of the server's capacity. Assuming server capacity is 100, the workload of an item is 25. If the workload of modules changes significantly over time, then no distribution can be optimal for a long time. In this case the workload distribution procedure must be carried out more frequently.

In set theory terms the workload distribution problem is stated as follows: divide a set into one or more disjoint subsets called blocks. This problem is called set partitioning and is well-known in computer science [12]. Through partitioning a set of workload items and assigning each block of a partition to one processing unit the workload distribution is carried out. The following example demonstrates the idea. Suppose there are four workload items, denoted as $w_i, i = \overline{1..4}$. In order to distribute them among processing units, a partition of the set $W = \{w_1, w_2, w_3, w_4\}$ must be generated. The set W can be partitioned in 15 different ways.

$$\begin{array}{lll} w_1w_2w_3w_4, & w_1w_2w_3|w_4, & w_1w_2w_4|w_3, \\ w_1w_2|w_3w_4, & w_1w_2|w_3|w_4, & w_1w_3w_4|w_2, \\ w_1w_3|w_2w_4, & w_1w_3|w_2|w_4, & w_1w_4|w_2w_3, \\ w_1|w_2w_3w_4, & w_1|w_2w_3|w_4, & w_1w_4|w_2|w_3, \\ w_1|w_2w_4|w_3, & w_1|w_2|w_3w_4, & w_1|w_2|w_3|w_4 \end{array} \quad (1)$$

Each of the partitions represents a possible workload distribution. The seventh partition, for instance, consists of two blocks: w_1w_3 and w_2w_4. Therefore, the corresponding distribution will require two processing units (one per block). Multiple ways of partitioning a set create the possibility of choice and the task of finding the best partition. This leads to an optimization formulation: *Given a set of workload items find its feasible partition that has minimum number of blocks.* A partition is called feasible if the workload created by any of its blocks is less than or equal to the capacity of a processing unit. For simplicity reasons we assume all processing units have the same capacity.

The next step is the formalization of the above statement. As with any optimization problem, the set partitioning problem must have three parts: (i) Decision variables: the representation of possible partitions; (ii) Objective function: a criterion of evaluating the "quality" of a partition; (iii) Constraints: feasibility restrictions on possible partitions.

In the current work, we use the classic integer programming formulation of the set partitioning problem. The formulation assumes a two-step solution procedure:

1) Generation of a set $B = \{b_j : j = \overline{1..N}\}$ of feasible candidate blocks $b_j = \{w_l : l \in \overline{1..n}\}$, where n is the number of workload items and N is the number of candidate blocks.
2) Construction of an optimal partition out of the previously generated blocks with the help of the following integer program:

$$\sum_{j=1}^{N} x_j \rightarrow min \quad (2)$$

$$\sum_{j=1}^{N} a_{ij} \cdot x_j = 1 \quad i = \overline{1..n} \quad (3)$$

$$x_j \in \{0,1\} \quad j = \overline{1..N} \quad (4)$$

where a decision variable x_j equals 1 if the j^{th} block is included in the optimal partition and 0 otherwise; a_{ij} is an element of matrix A of size $n \times N$ and calculated as:

$$a_{ij} = \begin{cases} 1 \text{ if } w_i \in b_j \\ 0 \text{ if otherwise} \end{cases} \quad (5)$$

The objective function (2) favors partitions with the smallest number of blocks. The constraints (3) force each workload item w_i to appear only in one block of the optimal partition. For convenience these constraints can be expressed in matrix form:

$$A \cdot x = \mathbb{1} \quad (6)$$

where A is defined by the expression (5), x is the vector with N decision variables and $\mathbb{1}$ is a vector of size n with all elements equal 1.

The reason for choosing the two-step procedure and the formulation (2) – (5) of the set partitioning problem is their wide and successful application in other areas, in particular airline crew pairing and stock cutting. Research results from these areas form a solid foundation for our own effort. Subsequent sections of the paper discuss different aspects of the solution procedure that influence its computational characteristics and the quality of the solution it produces.

To illustrate the usage of this set partition problem formulation, we apply it to the example mentioned earlier. Suppose the feasible candidate blocks are

$$\begin{array}{c} w_1,\ w_2,\ w_3,\ w_4,\ w_1w_2,\ w_3w_4, \\ w_1w_3,\ w_2w_4,\ w_1w_4,\ w_2w_3 \end{array} \quad (7)$$

while the rest of the blocks present in (1) are deemed infeasible. Hence, $n = 4$ (the number of workload items) $N = 10$ (the number of blocks to be considered). Given

the feasible blocks, the following integer program can be constructed:

$$\sum_{j=1}^{10} x_j \to min$$
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_{10} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ . \\ . \\ 1 \end{bmatrix}$$
$$x_j \in \{0,1\} \quad j = \overline{1..10} \tag{8}$$

The solution of the problem corresponds to the optimal partition (that has the smallest number of blocks). Given the small size of the problem, it is not difficult to see the three optimal solutions:

$$x^1_{opt} = \{0,0,0,0,1,1,0,0,0,0\} \Rightarrow \{w_1w_2\},\{w_3w_4\}$$
$$x^2_{opt} = \{0,0,0,0,0,0,1,1,0,0\} \Rightarrow \{w_1w_3\},\{w_2w_4\}$$
$$x^3_{opt} = \{0,0,0,0,0,0,0,0,1,1\} \Rightarrow \{w_1w_4\},\{w_2w_3\}$$

Note that there may be both multiple optimal solutions or no solution at all, in the case that there is at least one item with workload higher than the capacity of a processing unit.

IV. The "Full Set" Approach

As presented in the previous section the solution process starts with the generation of feasible candidate blocks. In our work we use two different methods to generate these blocks. The first one is a brute-force method that generates all possible combinations of workload items (i.e., all possible blocks). After that, the blocks must be validated against the feasibility constraints, that is, the workload of a block must not exceed the capacity of a processing unit. The total number of blocks to be considered will always be less than the number of all possible combinations of workload items:

$$N \leq \sum_{i=1}^{n} C_n^i = \sum_{i=1}^{n} \frac{n!}{i!(n-i)!} \tag{9}$$

Having generated the candidate blocks, the integer program is composed and solved. We call this method *the basic* version of the "full set" approach, because we explicitly consider all possible solutions. Section VI shows that this version is applicable only for a very small number of workload items. The reason is obvious: N, defined by the expression (9) grows very fast and the set of candidate blocks (B) quickly becomes unmanageable (either exceeds the amount of RAM or takes too much time to be processed). Nevertheless, considering all feasible combinations guarantees the best possible result. The example from Section III is solved by the "full set" approach.

In order to achieve better performance of the candidate generation, more appropriate limits for i in the expression (9) can be found. Experimenting with the "full set" approach we found that considering the blocks with too few or too many workload items is useless. Such blocks never appear in optimal partitions. Intuitively, big-size blocks are most probably infeasible, while small-size blocks increase the number of required processing units, which is not favored by criterion (2) and is therefore rejected. By considering only medium-size blocks the performance of the "full set" approach can be significantly improved. Restricting the set of candidate partitions is a very common approach used to improve the solution of the set partitioning problem. Experiments showed that in comparison with brute-forcing, the following limits produce better computational characteristics (e.g., lower memory consumption and faster processing time) without deteriorating the quality of the solution.

$$N = \sum_{i=lower}^{upper} C_n^i = \sum_{i=lower}^{upper} \frac{n!}{i!(n-i)!}$$
$$lower = \lceil \frac{1}{2} n_{avg} \rceil, \; upper = \lfloor \frac{3}{2} n_{avg} \rfloor \tag{10}$$
$$n_{avg} = \frac{Capacity}{w_{avg}} = \frac{Capacity \cdot n}{\sum_{i=1}^{n} w_i}$$

Here n_{avg} is the average number of items in a block, w_{avg} is the average workload of an item and $Capacity$ is the capacity of processing units. We call this modification "size-restricted" modification of the "full set" approach.

As one can see the expression (10) takes into account only the number of items in a block. This, however, is not the only factor that can reduce the size of the set B. Another tendency was revealed by observing the results of the conducted experiments: the workload is distributed in such a way that every processing unit is utilized to the highest possible extent. In other words, every processing unit is packed with workload items as much as possible. Based on this observation, we suggest a second way of generating set B. By sequentially iterating through the set of workload items, we select items as long as the total selected workload is less than or equal to the capacity of the processing units. We call this version of the "full set" approach *load-restricted*. The listing below presents the details of the algorithm.

```
generate (items, eps, capacity) {
  iter = items.begin();
  while (items.size() > 0) {
    s1 = s2 = count = 0;
    //select items to a new block
    while (count < items.size() && items.size() > 0) {
      s2 = s1 + iter->value;
      if (s2 < capacity - eps) {
        //add the workload item to the block and continue
        s1 = s2; count = 0;
        block.add(iter.value);
        items.remove(iter); iter++;
      }
      else if (s2 <= capacity) {
        //add the item and stop selecting more items
        block.add(iter.value);
        items.remove(iter); iter++;
        if (iter == items.end())
```

```
          iter = items.begin();
        break;
      }
      else {
        //skip the item and continue
        count++;
        item++;
      }
      if (iter == items.end())
        iter = items.begin();
    }
    B.add(block);
    block.clear();
  }
  return B;
}
```

At the end of the procedure, the set B contains a number of blocks constituting a feasible partition of a given set. By running the $generate$ procedure multiple times and shuffling the set of items before each run a required number of candidate blocks (i.e., the set B) can be generated.

The run-time complexity of the both versions of the "full set" algorithm is determined by the algorithm used for solving the integer problem (2)-(4). We used the branch-and-bound algorithm. Its complexity is exponential [13]. Hence, the complexity of the "full set" algorithms is also exponential, $O(2^N)$, where N is the number of candidate blocks.

V. The "Column Generation" Approach

Two factors make the "full set" approach impractical. The first one is the size of N, which can be enormous, even when n is still reasonable (say, less than 10000). The second factor is the integrality constraint (4). Solving large-scale integer programming problems is not a trivial exercise, and requires more a complex solution procedure in comparison to linear programming problems. These two factors significantly limit the applicability of the approach.

This section shows how these difficulties can be overcome by a method suggested in [14]. The main idea is to enhance the "pricing out" stage of the simplex method. At this stage, Danzig and Wolfe [14] suggest *generating* a useful *column* by solving an auxiliary integer programming problem instead of looking over a vast existing collection of columns to pick out a useful one.

Put simply, column generation means beginning with a manageable part of a linear optimization problem, solving that subproblem, and then discovering the way of improving the solution by extending the subproblem with the parts of the original problem. This process is repeated until a satisfactory solution to the original problem is achieved [15]. In formal terms, column generation is a modification to the simplex method that adds columns corresponding to constrained variables during the pricing phase [13].

Column generation relies on the fact that in the simplex method, the solver does not need access all the variables of the problem simultaneously. In fact, a solver can begin working with only the basis (a particular subset of the constrained variables) and then use reduced cost to decide which other variables are needed [16].

To solve a set partition problem by column generation we start with a subproblem, called the master problem. That is, we choose several feasible blocks and solve the problem (2) – (5) for them. This will surely work in that it produces some answer (a feasible solution) to the problem, but it will not necessarily produce a satisfactory answer. To move closer to a satisfactory solution, we can then generate other columns. Other decision variables (other x_j) will be chosen to add to the model. Those decision variables are chosen on the basis of their favorable reduced cost with the help of a subproblem. This subproblem is defined to identify the coefficients of a new column of the master problem with minimal reduced cost.

Let π be the vector of the dual variables of the current solution to the master problem. The subproblem is then defined as follows:

$$1 - \sum_{i=1}^{n} \pi_i c_i \rightarrow min \tag{11}$$

$$\sum_{i=1}^{n} w_i c_i \leq capacity \tag{12}$$

$$c_i \in \{0,1\} \quad i = \overline{1..n} \tag{13}$$

The solution to the problem (11) – (13), vector c_{opt}, represents the coefficients of a new column of the constraint matrix A of the master problem. Adding a new decision variable (i.e., a new candidate block) to the master problem with the constraints coefficients c_{opt} will result in the best possible improvement of its solution. In this way, instead of explicitly considering a fixed set of columns (candidate blocks), we generate and add new ones to the master problem only if they improve its solution. This avoids the need of explicitly enumerating candidate blocks.

Having discussed all necessary aspects of column generation we present the *basic* five-step algorithm of solving the set partitioning problem with column generation.

- **S1** Compose the master problem (2) – (5) for a limited set of candidate blocks. The simplest way to generate this set is to place each of the n workload items to a separate block. In this case the matrix A is a diagonal matrix: elements of the main diagonal equal 1, while non-diagonal ones 0.
- **S2** Solve the master problem.
- **S3** Given the optimal dual solution of the master problem, compose the auxiliary column generation problem (11) – (13).
- **S4** Generate a new column (i.e., a new candidate block) by solving the auxiliary problem.
- **S5** Add the new column to the master problem and return to the step **S2**. Repeat the procedure until the improvement of the master problem solution becomes negligible.

 ISBN: 978-1-61208-153-3

Having experimented with the algorithm, we observed an even stronger tendency to pack blocks with as many items as possible. In comparison to the blocks generated by the "full set" algorithms, column generation produced blocks with workload closer to the capacity of a processing unit. This tendency is true for every n, large or small. This fact, in turn, created a hypothesis that if the correlation between the block fulfillment and n is weak, we can split a sufficiently large set of workload items into a number of smaller subsets of the size k, $k \leq n$, and run the algorithm on each subset independently without deterioration of the quality of the overall solution. The prime reason for this is decreasing the execution time. In linear and integer programming the solution time increases non-linearly with the size of a problem. This implies that solving two problems with 50 variables takes less time than solving one problem with 100 variables. In order to check this hypothesis, we modified the basic column generation algorithm accordingly. The resulting version of the algorithm is called *parallel*. As one can see from the experiment results, the hypothesis proved to be true and allowed a very efficient algorithm.

The run-time complexity of both versions of the "column generation" algorithm is determined by the complexity of the simplex method, which is exponential [13]. Thus, the complexity of the basic version is $O(2^N)$, where N is the number of considered blocks and corresponds to the number of generated columns. The complexity of the parallel version slightly differs and is $\frac{n \cdot O(2^k)}{k}$, where n is the number workload items to be distributed and k is described above. In our experiments, we took $k = 50$. One can now clearly see that the expected speed-up of the parallel version equals to $O(2^N) - \frac{n \cdot O(2^k)}{k}$.

VI. Computation Results

This work contributes, in total, five workload distribution algorithms: three versions of the "full set" algorithm, and two versions of the column generation algorithm. In order to validate them a number of experiments were conducted. This section describes the set up of the experiments and reports their results.

All suggested algorithms were implemented in C++ and used the IBM ILOG CPLEX V12.1 optimization engine in order to solve the linear programming problems. All algorithms were run on a Quad-core Intel Xeon E5450 3.00GHz machine with 8 GB of RAM. The experiments were conducted as follows. First, n workload items were obtained with a random number generator. In the experiments we used random number equally distributed in the range from 12 to 40. The capacity of a processing unit has been fixed at 100 in all experiments. Second, each of the five algorithms was run on the generated set of workload items and the execution time was measured. Table I contains the results of the conducted experiments. In addition to the execution time, the obtained solution (the number of required processing units) is presented. For the "full set" algorithms we also report N - the number of candidate blocks considered. For the column generation algorithms the number of generated columns is reported. Because basic and size-restricted versions of the "full set" algorithm fail to distribute more than 15 items, the statistics on them are not included in the table.

For example, during the ninth experiment 600 workload items were generated. The basic and size-restricted versions of the "full set" algorithm failed due to size of the set B. The load-restricted version distributed the items among 162 processing units, but took 4 minutes to complete, and processed 86390 candidate blocks. The basic version of the column generation algorithm distributed the same workload items among 155 processing units. The algorithm generated 1541 columns and took 6 minutes while the parallel version of the algorithm was able to achieve the same result in only 24 seconds.

VII. Conclusion

Cloud computing's pricing model creates an incentive for subscribers to minimize the consumption of rented resources. In the case of modularized software, multiple deployment options may exist, creating different possibible distributions of workload and resource consumption. The current research aims to developing a formal approach of distributing multiple workload items among a minimum number of processing units.

We designed and evaluated five algorithms that, given a set of workload items, distribute them among processing units of specified capacity. The algorithms can be classified into two different types: those that explicitly consider a fixed set of candidate options (the "full set" algorithms) and those that gradually improve the solution by considering dynamically generated options (column generation algorithms).

The combinatorial nature of the workload distribution problem makes any algorithm based on explicit enumeration of possible alternatives intractable. That is, even for reasonably sized input, the algorithms fail due to the overwhelming number of alternatives to be processed. The experimental results clearly demonstrated this phenomenon.

The results also showed that the basic version of the column generation algorithm produces the best solution. The solution found by the parallel version of the column generation algorithm is worse by approximately 1%. However, the speed of parallel version is much better. For this reason, we conclude that the best results are achieved with parallel version of the column generation algorithm.

Acknowledgments

The authors want to express special thanks to Nick Lanham for the numerous improvements he contributed to this paper.

 ISBN: 978-1-61208-153-3

Table I
EXPERIMENT RESULTS

		Full Set			Column Generation					
		Load-restr.			Basic			Parallel		
No	n	Sol.	N	Time	Sol.	Cols.	Time	Sol.	Cols.	Time
1	10	3	100	1 sec	3	15	1 sec	3	15	1 sec
2	15	4	137	1 sec	4	23	1 sec	4	23	1 sec
3	30	9	300	1 sec	8	25	1 sec	8	25	1 sec
4	50	14	473	2 sec	13	121	2 sec	13	121	2 sec
5	100	28	793	3 sec	26	233	6 sec	26	243	3 sec
6	150	40	2870	15 sec	39	322	21 sec	39	348	5 sec
7	250	71	8800	36 sec	69	672	45 sec	70	1523	11 sec
8	400	108	20000	1 min	104	912	3 min	104	938	17 sec
9	600	162	86390	4 min	155	1541	6 min	156	1523	24 sec
10	1000	275	133000	7 min	262	2637	22 min	263	2654	40 sec
11	1300	349	219240	19 min	330	3674	42 min	333	3243	60 sec
12	1500	405	303750	27 min	389	3853	93 min	390	3789	75 sec
13	2000	536	549000	31 min	514	5201	168 min	516	5017	84 sec
14	2500	670	846250	39 min	651	6435	274 min	654	6337	95 sec
15	3000	807	1200000	45 min	-	-	-	774	7519	112 sec
16	5000	1342	3365000	74 min	-	-	-	1296	12595	200 sec
17	10000	2698	12146396	193 min	-	-	-	2598	25170	378 sec

		Full Set					
		Basic			Size-restr.		
No	n	Sol.	N	Time	Sol.	N	Time
1	10	3	35673	5 sec	3	27990	3 sec
2	15	4	3012765	12 min	4	2366910	9 min
3	30	-	-	-	-	-	-

REFERENCES

[1] P. Murray, "Enterprise grade cloud computing," in *Proceedings of the Third Workshop on Dependable Distributed Data Management, European Conference on Computer Systems*, 2009, pp. 1–1.

[2] R. L. Grossman, "The case for cloud computing," *IT Professional*, pp. 23–27, March 2008.

[3] W. Tang and M. W. Mutka, "Load distribution via static scheduling and client redirection for replicated web servers," in *International Conference on Parallel Processing*, 2000.

[4] N. Nehra, R. B. Patel, and V. K. Bhat, "A framework for distributed dynamic load balancing in heterogeneous cluster," 2007.

[5] D. Grosu and A. T. Chronopoulos, "A game-theoretic model and algorithm for load balancing in distributed systems," in *16th International Parallel and Distributed Processing Symposium*, 2002, pp. 146–153.

[6] S. Iqbal and G. F. Carey, "Performance analysis of dynamic load balancing algorithms with variable number of processors," *Journal of Parallel and Distributed Computing*, vol. 65, pp. 934–948, 2005.

[7] J. Arabeyre, J. Fearnley, F. Steiger, and W. Teather, "The airline crew scheduling problem: A survey." *Transportation Science*, vol. 3, no. 2, p. 140, 1969.

[8] R. E. Marsten and F. Shepardson, "Exact solution of crew scheduling problems using the set partitioning model: Recent successful applications," *Networks*, vol. 11, no. 2, pp. 165–177, 1981.

[9] J. Ford, L. R. and D. R. Fulkerson, "A suggested computation for maximal multi-commodity network flows," *MANAGEMENT SCIENCE*, vol. 5, no. 1, pp. 97–101, 1958.

[10] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting-stock problem," *Operations Research*, vol. 9, no. 6, pp. 849–859, 1961.

[11] M. Minoux, "Column generation techniques in combinatorial optimization: A new approach to the crew pairing problems," in *24th AGIFORS Symposium*, 1984, pp. 15–29.

[12] D. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 3*. Addison-Wesley, 2005.

[13] G. Desaulniers, J. Desrosiers, and M. M. Solomon, *Column generation*. Springer, 2005.

[14] G. B. Dantzig and P. Wolfe, "Decomposition principle for linear programs," *OPERATIONS RESEARCH*, vol. 8, no. 1, pp. 101–111, 1960.

[15] M. E. Lübbecke and J. Desrosiers, "Selected topics in column generation," *Operations Research*, vol. 53, pp. 1007–1023, November 2005.

[16] D. Feillet, "A tutorial on column generation and branch-and-price for vehicle routing problems," *4OR: A Quarterly Journal of Operations Research*, vol. 8, pp. 407–424, 2010.

 ISBN: 978-1-61208-153-3