ORIE 5135 Computational Integer Programming
Final Project: Variations on Bin Packing

Abraham Hill

April 2022

1 Introduction

1.1 Problem Statement

We are given a set of n machines (indexed by $j \in \mathcal{M}$) that are tasked with completing a set of m jobs (indexed by $i \in \mathcal{J}$). Each job has an associated processing time $p_i \in \mathbb{R}+$ and memory requirement $r_i \in \mathbb{R}+$ which do not depend on the machine that completes the job. All machines are identical, and they are characterized by a processing time capacity P and a memory capacity R. The work of completing a job cannot be shared between different machines. We assume that each job can be completed without exceeding the time or memory limit of a machine, that is, $p_i \leq P$ and $r_i \leq R, \forall\, i \in \mathcal{J}$.

The primary objective of the problem is to determine the optimal assignments of jobs to machines in order to minimize the total number of machines used. This problem is a variation of the bin packing problem, in which the goal is to place items with weight w_i into the minimal number of identical bins with weight capacity b. An alternate objective, minimizing the completion time of the machine that finishes last, is discussed in section 6. This paper will discuss different mixed-integer linear programming (MILP) models and methods that accomplish these objectives.

1.2 Data

The data for this project were provided directly in csv files. For the objective of minimizing the number of machines used, there are four datasets, each containing ten instances of the problem. The quantities P, R, and n remain constant within each dataset, while r_i and p_i vary across instances. For each of these datasets, $m = n$. The fourth dataset only contains one instance, which is a challenge that contains much larger values of n. This challenge is discussed in section 5. The data are structured the same for the time minimization objective, except that there is a single dataset (with ten instances) that does not include a value of P, since the processing time is minimized in the objective.

1.3 Model Evaluation Metrics

The merits of a MILP are primarily based on the relative speed with with a mathematical optimization solver such as Gurobi can find the optimal solution. The most computationally intense task in doing so is often the execution of the branch and bound algorithm, which is used for imposing integrality requirements on the decision variables. The number of nodes explored in this tree is a measurable means of comparison between two models. One factor that predicts the run time of branch and bound is the quality of the linear relaxation of the model. This is measured by the percent difference between the optimal value z^* of the integer program and its continuous (or linear) relaxation z^*_{LP}. This quantity will be called the *solution gap* $\equiv (z^* - z^*_{LP})/z^*$.

2 Natural Formulation

The first integer program we will discuss is the easiest to formulate and understand. It has descriptive variables that clearly outline the decisions to be made by the model.

2.1 Inputs and Variables

The inputs to this model are the data described in section 1.2. No additional processing is necessary for this model.

The decision variables are x_{ij}. If job i is assigned to machine j, x_{ij} takes on the value 1, and otherwise it is 0. We also define a second set of variables y_j that indicate whether machine j is used in the solution. Similarly to the other variables, $y_j = 1$ when the machine is being used, and otherwise it takes on the value 0. Although y does not encode additional information about the assignment of jobs to machines, it is useful to include so that the objective function, minimizing the number of machines used, can be easily defined. Using these variables, we can write the following integer program.

2.2 Model

$$
\begin{aligned}
\text{minimize} \quad & \sum_{j \in \mathcal{M}} y_j \\
\text{subject to} \quad & \sum_{i \in \mathcal{J}} r_i x_{ij} \leq R y_j, \; j \in \mathcal{M}. \text{ Memory limit of each machine} \\
& \sum_{i \in \mathcal{J}} p_i x_{ij} \leq P y_j, \; j \in \mathcal{M}. \text{ Processing time limit of each machine} \\
& \sum_{j \in \mathcal{M}} x_{ij} = 1, \qquad i \in \mathcal{J}. \text{ Each job must be completed} \\
& x_{ij} \leq y_j, \qquad i \in \mathcal{J}, j \in \mathcal{M}. \text{ Logical constraint: } x_{ij} = 1 \Rightarrow y_j = 1 \\
& x_{ij}, y_j \in \{0, 1\}, \; i \in \mathcal{J}, j \in \mathcal{M}. \text{ Binary restrictions}
\end{aligned}
$$

The number of machines used in the solution is the number of entries in y that have value 1. Therefore, we minimize the sum of the entries of y so that the MILP will find the assignment of jobs to the fewest possible machines.

The memory and processing time limit constraints would be valid without the inclusion of y_j on the right hand side. However, including y_j improves the performance of the model significantly: the quality of the continuous relaxation and overall solve time are both much better in this version.

2.3 Qualitative Evaluation

The advantage of this model, as mentioned previously, is that it is relatively easy to formulate and understand. However, there are two notable problems with this model.

First, the optimal value of the continuous relaxation of this model (the model without binary restrictions) is not a very effective lower bound on that of the integer solution. In the branch and bound algorithm, the variables that are not restricted to be binary will naturally take on fractional values. Therefore, finding the optimal integer solution will require the depth of the branch and bound tree to approach the number of variables n, because integrality will need to be imposed on each variable individually. For that reason, this model is generally considered to be "weak."

Second, and even more importantly, this model is very symmetric, meaning that the variables are highly interchangeable. For each integer solution, we can construct an equivalent solution by changing the machine indices that jobs are assigned to. In fact, for each integer solution with value k, there exist $\binom{n}{k}k!$ equivalent solutions to the continuous relaxation. Due to the symmetry of the model, the branch and bound tree will be relatively dense because the algorithm will need to explore many different (yet often interchangeable) solutions at high depth before finding the optimal one.

This formulation works well enough for small problem instances, but as the size increases, the time to solve it quickly becomes impractically long.

3 Strong Formulation

The shortcomings of the natural formulation can largely be overcome by including a variable for each feasible assignment of jobs to a machine.

3.1 Inputs and Variables

In this model, we begin by finding all possible combinations of jobs that could be assigned to a single machine without violating the time and memory limits of the machine. The set of feasible job assignments $\mathcal{S}$ is a subset of all job assignments $\mathcal{S}'$, which itself consists of all subsets of the job indices $i = 1 \dots n$. A simple method was used to enumerate $\mathcal{S}$: first, find $\mathcal{S}'$ (this is done using a standard "powerset" function that returns all subsets of an input set), then for each candidate subset of indices $S \in \mathcal{S}'$, add it to $\mathcal{S}$ if it satisfies the memory and time constraints. This can be done more efficiently by "remembering" which combinations of jobs are infeasible and then never considering any further combinations that contain those jobs together. The size of the problem could also be reduced by only considering maximal subsets. However, the previously described algorithm proved to run quickly enough (see section 4) that no such improvements were necessary.

Once the set of feasible job assignments has been found, a model must be devised to choose which assignments to use. The decision variables are x_S, $S \in \mathcal{S}$ which take on the value 1 if subset S is assigned to a machine, and 0 otherwise.

3.2 Model

$$\begin{aligned}
\text{minimize} \quad & \sum_{S \in \mathcal{S}} x_S \\
\text{subject to} \quad & \sum_{S \in \mathcal{S}:\, i \in S} x_S \geq 1, \; i \in \mathcal{J}. \text{ Each job must be assigned to a machine} \\
& x_S \in \{0, 1\}, \qquad S \in \mathcal{S}
\end{aligned}$$

The number of machines used in the solution is the number of entries of x with value 1, because if $x_S = 1$, then the subset of jobs S was assigned to a machine. Therefore, we minimize the sum of the entries of x in the objective.

The job completion constraint in this model is not especially intuitive. On the left hand side, we take the sum over all subsets S for which the job i is in S. Requiring this sum to be at least 1 ensures that each job will be included in one of the subsets of jobs used in the solution.

3.3 Qualitative Evaluation

This model does not specify which to machine each subsets of jobs is assigned, so the problem of symmetry is completely eliminated. This MILP also benefits from incorporating much of the complexity of the problem into the definition

of the variables themselves, which leads to a much more compact formulation, and it has a stronger continuous relaxation for most instances.

However, as the number of jobs, m, increases, the number of feasible assignments of jobs to machines grows exponentially, so for large m it is impractical to enumerate all feasible assignments and solve a MILP with so many variables. This challenge can partially be overcome, and the solution is discussed at length in section 5.

4 Quantitative Model Comparison

Both formulations were implemented in Julia, and all instances in the three datasets were solved using Gurobi. The code and raw data on the solutions are attached in the min_machines folder. The solution gap, branch and bound nodes explored, and solve time were computed for each instance and averaged across datasets. To ensure a reasonable run time, a solve time limit of 5 minutes per instance was imposed. For the strong formulation, the time to enumerate the set of feasible assignments $\mathcal{S}$ is included in solve time. For the natural formulation, the solution gap was computed using the IP objective value of the strong formulation because the strong formulation consistently solves the problem within the time limit. The number of instances (out of ten) solved to optimality within the time limit were also counted for each dataset. These data are shared in tables 1 and 2:

m	Avg Solution Gap	Avg Nodes	Avg Solve Time	Time $\leq$ 300s
15	0.141	1.0	0.01888s	10
30	0.0761	442.7	0.4875s	10
50	0.0466	98884.7	102.8s	7

Table 1: Model Performance Data for the Natural Formulation

m	Avg Solution Gap	Avg Nodes	Avg Solve Time	Time $\leq$ 300s
15	0.0460	1.0	0.00185s	10
30	0.0197	1.0	0.00622s	10
50	0.0206	1.0	0.0409s	10

Table 2: Model Performance Data for the Strong Formulation

The natural formulation can solved reasonably quickly for small problem sizes, but as m increases, the average number of branch and bound nodes and average solve time increase rapidly. This MILP was only able to find a confirmed optimal solution for seven out of the ten instances with $m = 50$, which is not an especially large problem. Interestingly, the solution gap actually decreased with increasing m, which is evidence that the biggest issue with this model is its symmetry, not its weak linear relaxation. This model is only practical to use for instances with m not much greater than 30.

The strong formulation clearly dominates the natural formulation across all metrics. For each dataset, the average solve time was less than 0.05 seconds, and did not show the same extreme growth with increasing m. It has an average solution gap of less than 5% for all datasets. Due to the narrow solution gap and lack of symmetry in the model, only a single branch and bound node was needed to impose integrality for every instance. The enumeration of $\mathcal{S}$ did not take long at all for instances of this size. However, with much larger m, the exponential number of variables becomes an issue.

5 Bonus: Solving Instances with Large m

As discussed previously, both the natural and strong formulations are not practical to solve in their current form for instances with large m. For the natural formulation, $m = 50$ is already large enough that imposing a time limit becomes necessary. For the strong formulation, the algorithm to enumerate the set of feasible job assignments $\mathcal{S}$ has exponential time complexity with respect to n, as does the exploration of the branch and bound tree. The goal of this section is to discuss how to solve instances of the job assignment problem with size $m \geq 1000$.

5.1 The Column Generation Algorithm

Column generation is an algorithm for solving linear programs with large numbers of variables (or columns of A). At a high level the algorithm is to (1) initialize the problem with a small, feasible subset $\tilde{A}$, $\tilde{c}^T$, and $\tilde{x}$ of the columns of A, c^T, and x, (2) solve the reduced LP to get the solution $\tilde{x}^*$ and associated dual solution y^*, and (3) use the dual solution to check for optimality, and if necessary, add more columns and return to step 2. This way, the size of the problem can be kept small enough to be practically solvable at each iteration, and ideally, an optimal solution can be found using a small fraction of the columns in the original problem.

5.2 Initialization

The initial variables were selected by creating maximal subsets of indices using a simple algorithm: starting with the set of all the job indices, remove the first index from the set and place it in a new subset, then find the next index for which the subset of indices, if assigned to a machine, would satisfy the processing time and memory constraints. Keep doing this until there are no more indices that would "fit" (meaning the subset is maximal). Continue creating maximal subsets in this fashion until each job index has been assigned to a subset. The collection $\tilde{\mathcal{S}}$ of these subsets constitutes the initial assignments of jobs to machines. We have already checked this solution for feasibility by imposing the time and memory constraints on each subset and requiring that each index to belong to a subset. This initialization places a tighter upper bound on the solution than the previous approach, which should result in a faster run time.

5.3 Solving the Reduced Linear Program

Retrieving the primal optimal solution $\tilde{x}^*$ to the reduced linear program is as simple as solving the model in section 3.2 with $\mathcal{S} = \tilde{\mathcal{S}}$ and relaxing the integrality constraints. The dual optimal solution y^* can easily be found from the primal solution after the reduced linear program is solved.

5.4 The Pricing Problem

Checking for optimality and choosing which columns to add can be done by examining properties of the dual solution y^*. The dual of the reduced linear program is

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in \mathcal{J}} y_i \\ \text{subject to} \quad & \sum_{i \in S} y_i \leq 1, \quad S \in \tilde{\mathcal{S}} \\ & y_i \geq 0 \quad i \in \mathcal{J} \end{aligned}$$

The primal solution $\tilde{x}^*$ is optimal exactly when the corresponding dual solution y^* is dual feasible. If it is, then the algorithm terminates at an optimal solution: $x^* = \tilde{x}^*$. If y^* is not dual feasible, then $\tilde{x}^*$ is not primal optimal. This implies that there exists some subset S that would improve the solution if it were added to the objective. Therefore, it is necessary to choose "some" subsets to add to $\tilde{\mathcal{S}}$. The choice of which subsets to add is called the pricing problem because we evaluate the reduced cost, or potential improvement to the objective of the reduced linear program, of each variable not in $\tilde{\mathcal{S}}$.

The method used in this implementation is to add the subset S^* corresponding to the "most violated" dual constraint, which is associated with the variable with the largest reduced cost. This subset can be found be solving the knapsack problem with profits y^*, and the usual memory and time constraints:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in \mathcal{J}} y_i z_i \\ \text{subject to} \quad & \sum_{i \in \mathcal{J}} r_i z_i \leq R \\ & \sum_{i \in \mathcal{J}} p_i z_i \leq P \\ & z_i \in \{0, 1\} \quad i \in \mathcal{J} \end{aligned}$$

where z_i are decision variables that specify whether job index $i \in S^*$. The objective function finds the maximum value $\bar{z}$ of the left hand side of a dual constraint. The two constraints ensure that S^* is a feasible assignment of jobs to a machine. Then, any maximal subset $S' \in \mathcal{S} : S^* \subseteq S'$ is the dual constraint that is farthest from being satisfied. We then add S' to $\tilde{\mathcal{S}}$, and return to step 2.

An advantage of this approach is that it also allows us to conveniently check if y^* is dual feasible. If the optimal value of $\bar{z}$ is less than or equal to one, then there does not exist a subset $S \in \mathcal{S} \backslash \tilde{\mathcal{S}}$ such that

$$\sum_{i \in S} y_i > 1$$

Therefore, every dual constraint is satisfied. This implies that $\tilde{x}^*$ is optimal, so we stop and return the solution.

Instead of using MILP, the knapsack problem can also be solved combinatorially. However, this approach is simpler with only one constraint (on either processing time or memory). I implemented a combinatorial algorithm that finds a heuristic solution faster than the MILP solver finds the optimal one, but did not use this approach in the final algorithm because the solve time of the knapsack problem is relatively fast ($\leq$ 1s) and does not increase with $|\tilde{\mathcal{S}}|$.

The method by which S^* is expanded to a maximal subset is meaningful for the performance of the algorithm. The basic method is to assign job indices to the subset until adding any additional index would violate the processing time or memory constraint. My initial approach for implementing this was to create a set T^* such that $S^* \cup T^*$ is the set of all job indices. Then, loop through T^* and add any indices to S^* until it is maximal. An issue with this approach is that T^* is ordered, so at each iteration, the same job indices are being added, which limits the progress of the algorithm in adding variables. I found that randomizing the order of T^* resulted in a quicker convergence to near-optimality, although actually reaching optimality took the approximately the same amount of time and number of iterations. For problem sizes for which it is necessary to limit the run time of the algorithm, this is useful because the algorithm will take less time to reach a given objective threshold.

5.5 Reimposing Integrality Requirements

The column generation algorithm results in an optimal solution to the continuous relaxation of the integer program we want to solve. In order to convert this solution to an optimal one, the resulting set $\tilde{\mathcal{S}}$ of column generation can be used to solve the reduced linear program with integrality constraints imposed. Even with column generation, however, the size of this final problem is still large, so limiting the solve time was necessary.

5.6 Results

This algorithm was implemented in Julia with Gurobi using the method described in previous sections, and the code is attached in the column_generation folder. To verify correctness, column generation was executed for each of the thirty smaller instances, and the results agreed with those produced by the models in sections 2.2 and 3.2. The data on this can be found in the attachment col_gen_solns.csv.

I ran my implementation on the fourth dataset (with $m = 1000$) and found the optimal solution to the continuous relaxation! The algorithm added 3322 variables and terminated in about 90 minutes with an optimal objective value of 405.2. I ran branch and bound for 30 minutes, and found an integer solution that uses 408 machines, which is only two away from the best possible bound, 406, for a gap of 0.05%.

It is unreasonable in most practical applications to run this algorithm for two hours. I imposed limits on the iteration count of column generation and time spent doing branch and bound to evaluate the quality of solutions obtained in different time frames. These data are displayed in table 3:

Iteration Limit	BB Time Limit	Total Time	LP Objective	IP Objective
∞ (3000)	30 mins	120 mins	405.20	408
2000	5	21	405.97	411
1000	2	5	416.6	422
500	1	1	455.3	456

Table 3: Column Generation Solution Values with Different Algorithmic Limits

The job assignments corresponding to each solution can be found in the attachment named (IP objective value)_assignments.csv.

As expected, there is a trade-off between the limits imposed on the algorithm and resulting solution quality. However, this algorithm converges somewhat quickly, given that a solution within 4% of the best possible optimal value can be found in five minutes. The column generation algorithm enables us to solve very large problems to near-optimality in reasonable time frames.

6 Minimizing Completion Time

An alternate objective for this problem is to minimize the completion time of the machine that finishes its assigned jobs last. In this problem, we no longer have a maximum processing time constraint for each machine.

The natural formulation from section 2 is a good starting point for this model. We can remove the y_j variables since they were defined specifically so that the objective function could be easily specified. We can also remove the processing time limit constraint. The challenge in formulating this model is to devise a linear objective function, which is discussed in the next subsection.

6.1 Formulating the Objective as a Linear Expression

For each feasible solution to this problem, every machine j will have an associated time P_j that it takes to process the jobs that are assigned to it. The objective value is the maximum of these times: $max(P_1, P_2, \ldots, P_n)$. However, this expression is nonlinear, so we must find a different way to evaluate the objective value. To do this, we can define a new variable $t \in \mathbb{R}+$ which will be constrained to be greater than the completion time of each machine. Then, when we minimize t in the objective, it will naturally take on the value of the largest completion time of all of the machines.

6.2 Model

$$
\begin{aligned}
\text{minimize} \quad & t \\
\text{subject to} \quad & \sum_{i \in \mathcal{J}} p_i x_{ij} \leq t, \ \ j \in \mathcal{M}. \text{ Objective value lower bound} \\
& \sum_{i \in \mathcal{J}} r_i x_{ij} \leq R, j \in \mathcal{M}. \text{ Memory limit} \\
& \sum_{j \in \mathcal{M}} x_{ij} = 1, \quad i \in \mathcal{J}. \text{ Every job must be assigned to a machine} \\
& x_{ij} \in \{0, 1\}, \quad i \in \mathcal{J}, j \in \mathcal{M}. \\
& t \geq 0
\end{aligned}
$$

6.3 Model Evaluation

This MILP was implemented in Julia and solved with Gurobi over ten instances, each with 30 machines and 50 jobs. As in section four, I collected data to evaluate the performance of the model. The code is attached in the min_time folder, and the raw data can be found in min_time_solutions.csv. The averages of the model performance data are displayed in table 4.

Qualitatively, this model does not have any glaring weaknesses. Although it looks similar to the natural formulation of the machine minimization objective, there are important differences. This model is not as symmetric as

n	m	Solution Gap	Nodes Explored	Time	Time $\leq$ 120s
30	50	0.093	12.5	0.00034	10

Table 4: Model Performance Data: Minimizing Completion Time

the natural formulation, because due to the exclusion of y, there are fewer variables that correspond to a given machine j. Additionally, the objective function does not rely as heavily on the integrality of the variables because t is not a binary variable. These factors should significantly improve the performance of this MILP over the natural formulation.

Although this model was only evaluated over ten instances, its performance does appear relatively strong. It explores far fewer nodes than the natural formulation does on instances of similar size, and the solution gap is reasonably low. Its average solve time across this dataset is the lowest we have seen in this report, including datasets with smaller instances.