Mathematical Institute

Integer Programming B6.3, MT 2018

MSc Special Topic

Shift scheduling with non-stationary stochastic delivery demand

Candidate Number:
1032562

Lecturer:
Prof. Raphael Hauser

January 17, 2019

1 Introduction

The fast delivery market has been growing critically over the past years ([12]). Such service systems with limited opening hours face with stochastic and non-stationary[1] demand. In this context, determining staffing requirements is a challenge. However, up to date, shift scheduling with non-stationary stochastic demand has received limited attention in the academic literature. Much research has focused on scheduling problems in call centers (see, e.g, [1], [2] or [3]). In this paper, fast delivery shift scheduling problems with service-level constraints are formulated as mixed-integer programs and investigated through a simulation-based branch-and-bound approach. Section 2 specifies the problem formulation. Section 3 provides and discusses a technique to solve the problem. Section 4 applies the method to a real-world case study.

2 Problem Formulation

2.1 Context

Fast delivery systems must spread workers over predefined work shifts to satisfy demand. Demand is called *customer load.* Figure 1 displays the typical demand curve experienced by a food delivery system over a day. Time is cut into short periods called *staffing intervals* on which *capacity* (i.e number of workers) is assumed steady. Note that work shifts may overlap each other and be of varying durations. Shifts are the (longer) periods which workers are allocated to. Staffing intervals (or staffing periods) are simply the short time periods obtained by discretizing time (intervals of 5, 10, 15 min...).

Provided any customer load forecast, determining staffing requires to quantify the labor needed during each staffing interval to ensure satisfactory customer service and select staff shifts that cover this determined labor amount. The latter is solved through a set-covering integer program formulated in 2.2.

As proposed by [4] and [5], (i) customer load arrival is assumed to follow a Poisson process with time-dependent rate $\lambda(t)$. Quality of service (ii) is measured in terms of customer waiting time. More precisely, we require that the (virtual) waiting time W_t encountered by a customer arriving at time t and willing to wait infinitely satisfies:

[1]i.e. the demand rate varies over time

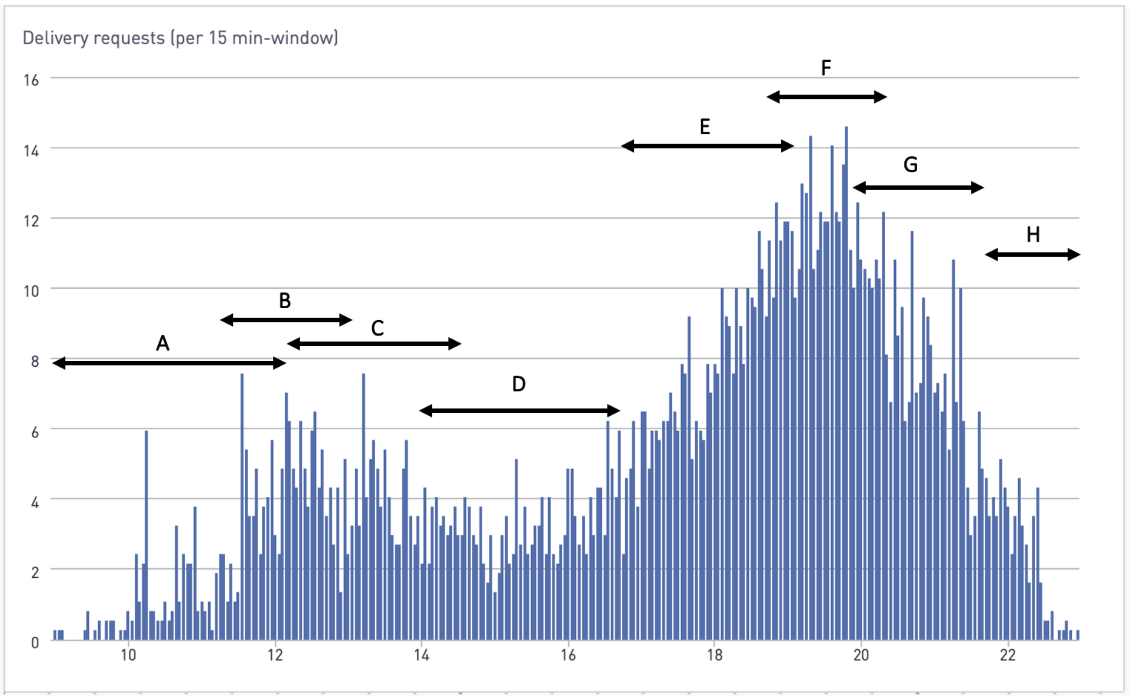


Figure 1: Schematic representation of time-varying customer load. In black, predefined work shits.

$$\mathbb{P}(W_t > \tau) \leq \alpha \quad \forall t \tag{1}$$

where τ is the maximum allowed waiting time and α the target probability of excessive waiting. Equation (1) is called *service level (or performance) constraint.* We define the staffing levels $s_1, ..., s_p$ as the numbers of workers required in the p staffing intervals. Note that the waiting time depends on the staffing levels, hence so does the constraint:

$$\mathbb{P}(W_t > \tau) = \mathbb{P}(W_t > \tau \mid \mathbf{s})$$

where $\mathbf{s} = (s_1, ..., s_p)$.

2.2 Problem statement

The staffing problem can be formulated as:

$$\begin{aligned} \min \quad & \mathbf{c}^\intercal \mathbf{w} = \sum_{j=1}^{n} c_j w_j \\ \text{s.t.} \quad & \mathbf{A}\mathbf{w} = \mathbf{s} \\ & \mathbf{w} \geq \mathbf{0} \qquad\qquad \text{and integer} \end{aligned} \tag{2}$$

$$\text{while} \quad \mathbb{P}(W_t > \tau \mid \mathbf{s}) \leq \alpha \quad \forall t$$

where

- $\mathbf{c}$ is the shift cost vector (and c_j is the cost of shift j, expressed in worker hours)
- $\mathbf{w} = (w_1, ..., w_n)^\intercal$ where w_j is the number of workers in shift j
- $\mathbf{s} = (s_1, ..., s_p)^\intercal$ where s_i is the number of workers in staffing period i
- $\mathbf{A} = (a_{i,j})_{ij}$ where $a_{i,j}$ is 1 if staffing period i is in shift j, 0 otherwise
- n is the number of shifts
- p is the number of staffing periods

In reality, any shift scheduling is admissible as long as it covers the required minimal staffing levels. Therefore, as proposed in [6], the constraint $\mathbf{Aw} = \mathbf{s}$ can be relaxed into $\mathbf{Aw} \geq \mathbf{s}$. Thus, for any given staffing vector $\mathbf{s} = (s_1, ..., s_p)^\intercal$, the shift covering problem can be formulated as:

$$\min \quad \sum_{j=1}^{n} c_j w_j \tag{3}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{i,j} w_j \geq s_i \quad i = 1, ..., p \tag{4}$$

$$s_i \geq 0 \quad \text{and integer } i = 1, ..., n \tag{5}$$

$$\mathbb{P}(W_t > \tau \mid s_1, ..., s_n) \leq \alpha \quad \forall t \tag{6}$$

3 General Methodology

This section presents a method to solve the shift scheduling problem under stochastic non-stationary demand and with service-level constraints. It uses a branch-and-bound approach to decompose the problem into shift covering integer subproblems[2] (3)-(5) and simulation to check on the validity of constraint (6). Section 3.1 explains the construction and the exploration of the search tree. Section 3.2 describes how simulation is conducted. Section 3.3 provides a method to obtain the branch-and-bound algorithm inputs. Section 3.4 discusses the general method.

[2]The set covering problem was introduced by Dantzig (1954) [7].

3.1 A branch-and-bound algorithm

The method presented in this section is attributed to [5]. It requires three staffing vector inputs:

- an initial feasible solution $\mathbf{s}^{\mathbf{init}}$. It is any staffing vector that leads to a shift vector satisfying the service level constraint. The corresponding shift vector is obtained as the solution of problem (3)-(5).
- a lower bound vector $\mathbf{s}^{\mathbf{LB}}$ that contains a set of lower bounds on the staffing requirements for each staffing period $i = 1, ..., p$. We explain how to find $\mathbf{s}^{\mathbf{LB}}$ in 3.3.
- an upper bound vector $\mathbf{s}^{\mathbf{UB}}$ that contains a set of upper bounds on the staffing requirements for each staffing period $i = 1, ..., p$ (see 3.3).

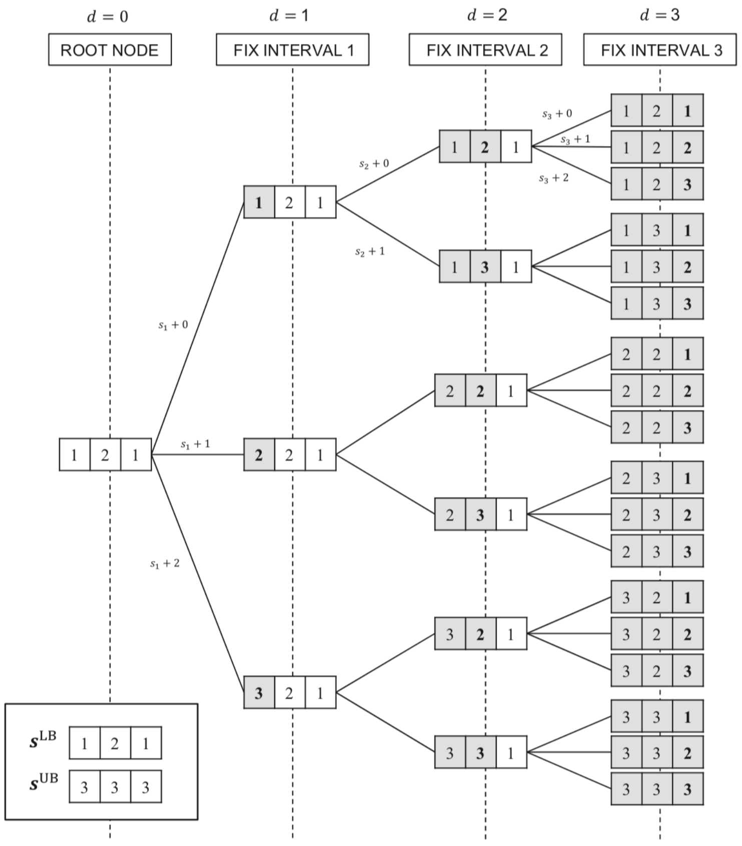


Figure 2: Example of tree structure. From [5].

Figure 2 presents an illustration of the tree structure, for p=3. Each node represents a staffing vector $\mathbf{s}$, with a corresponding staffing cost c_s (expressed in worker hours) and a corresponding (3)-(5) problem. The tree is explored in a depth-first manner. The root node of the tree is $\mathbf{s^{LB}}$ and each level of the tree is denoted by its depth d which corresponds to the d-th staffing period. For any parent node of depth d, child nodes are generated by increasing the capacity in staffing interval d+1 (and leaving all the other capacities unchanged). Hence, the staffing level d+1 of the child nodes lies between the lower bound s_{d+1}^{LB} and the upper bound s_{d+1}^{UB}. Moreover, child nodes are explored in the increasing order of capacity c_s. With these elements in mind, the tree structure satisfies:

Theorem 1. *Let* $\mathbf{s}$ *be a node of depth* d *and* c_s *the corresponding staffing cost. The staffing cost of any child node (of depth* $\geq d$*+1) and any unexplored node of depth* d *is at least* c_s*.*

Proof. Recall that staffing costs are expressed in terms of worker hours. Any child node has the same staffing capacities as $\mathbf{s}$ for staffing intervals $\leq d$ and higher (or equal) staffing capacities on staffing intervals $> d$, so the overall staffing cost is higher or equal to c_s. Similarly, any unexplored node of depth d has the same capacities as $\mathbf{s}$ except on staffing interval d where it is strictly higher. □

Although all nodes are enumerated implicitly, they are not all explored explicitly. Indeed, for each of them, a problem (3)-(5) must be solved to find the corresponding shift vector and a simulation must be run to check on the performance criteria for this shift vector. As those two steps can be computationally expensive, rules are implemented to *fathom* nodes as soon as they fail at meeting specific requirements. A node is said to be fathomed if it is discarded from the search procedure, along with all its child nodes. Throughout the search tree procedure, the best feasible shift vector found so far is stored ($\mathbf{w}^*$ with shift cost c_w^*). At the start, $\mathbf{w}^*$ is initialized to $\mathbf{w^{init}}$ (the shift vector solution of the shift covering problem (3)-(5) solved with $\mathbf{s} = \mathbf{s^{init}}$). Any explored node (staffing vector) is assessed through 4 steps at most.

Step 1 *Evaluate staffing cost* c_s - the staffing cost c_s is the number of worker hours that corresponds to the staffing vector $\mathbf{s}$. If $c_s \geq c_{w^*}$, $\mathbf{s}$ can be fathomed along with all its child nodes and all unexplored nodes from the same parent node according to Theorem 1. The algorithm then proceeds to the next unexplored node in the tree: this can be a node at depth $d-1$ along the same branch as the parent node, or a node higher in the tree (if backtracking takes place). This rule is referred to as Fathom[$\mathbf{c_s}$].

Step 2 *Evaluate shift cost c_w* - Let c_w be the solution cost for integer program (3)-(5). If $\mathbf{s}$ has not been fathomed in Step 1, the LP relaxation of problem (3)-(5) is solved. According to Proposition 2.2 of lecture notes [8], the related shift cost c_w^{LP} satisfies $c_w \geq c_w^{LP}$. Hence, as before, if $c_w^{LP} \geq c_{w^*}$, $\mathbf{s}$ can be fathomed along with all its child nodes and all unexplored nodes from the same parent node and the algorithm then proceeds to the next unexplored node. This rule is referred to as Fathom[$\mathbf{c_w^{LP}}$]. Otherwise, problem (3)-(5) is solved with the integrality constraints included. Again, if $c_w \geq c_w^*$, the node $\mathbf{s}$ is fathomed along with its underlying nodes and unexplored nodes from the same parent. This rule is referred to as Fathom[$\mathbf{c_w}$].

Theorem 2. *Let $\mathbf{w}$ be the shift vector solution for a given staffing vector $\mathbf{s}$. Let us assume that $\mathbf{w}$ violates the performance constraint. Let $t^e = min\{t \,|\, \mathbb{P}(W_t > \tau) > \alpha\}$ the first time instant the constraint was violated. Let i^e denote the staffing interval that contains $t^e + \tau$. All $\mathbf{s}'$ for which $s'_i \leq s_i$ for all $1 \leq i \leq i^e$ are unfeasible as well, irrespective of the capacity in intervals $i > i^e$.*

Proof. To attain a waiting time $\leq \tau$, a customer arriving at t^e must enter the system at $t^e + \tau$ at the latest. If this condition is not satisfied for staffing vector $\mathbf{s}$, it cannot be satisfied neither for staffing vectors requiring fewer (or the same number of) workers ahead of time. $\square$

Step 3 *Check if $\mathbf{w}$ was simulated before* - Different $\mathbf{s}$ vectors can lead to identical vectors. Therefore, it is possible that $\mathbf{w}$ has already been simulated at a previous node. To avoid running unnecessary expensive simulations, unfeasible simulated shift vectors are stored in a set $\mathbf{B}$. Thanks to Theorem 2, we can define a fathom rule termed Fathom[$\mathbf{i^e}$]. We distinguish three cases:

- $i^e < d$: the parent node at depth i^e can be fathomed along with all its child nodes and the next unexplored node in depth i^e is proceeded.
- $i^e > d$: we branch to the next unexplored node at depth i^e.
- $i^e = d$: the node at depth d can be fathomed and the algorithm proceeds with the next unexplored node at level d.

If the shift vector $\mathbf{w}$ was not simulated before, we go to step 4.

Step 4 *Evaluate* **w** *through simulation* - The simulation procedure is detailed in 3.2. If **w** is feasible, a new optimum has been found and $\mathbf{w}^*$ and c_{w^*} are updated. All the child nodes and the unexplored nodes sharing the same parent node can be fathomed as they cannot improve the optimum. This rule is named Fathom[$\mathbf{w}^*$]. As in steps 1 and 2, the algorithm proceeds with the unexplored nodes in the search tree. Otherwise, if **w** is unfeasible, the Fathom[$\mathbf{i}^\mathbf{e}$] rule is applied and **w** is stored in **B**.

Appendix B presents the related pseudo-code proposed by [5]. We implement this code in Python.

3.2 Simulation

The planning horizon (typically a day) lasts T minutes and is divided into several staffing periods $i = 1,..,p$ which start at time t_i and end at t_{i+1} respectively. The staffing periods last Δ_p so that $\mathbf{t_p} = \{t_1, t_2, ..., t_p\} = \{0, \Delta_p, 2\Delta_p, ..., T - \Delta_p\}$.

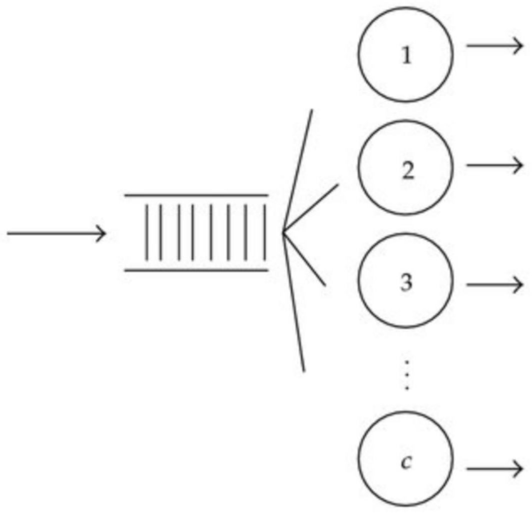


Figure 3: Representation of a first-come first-served queueing process. From [11].

We use a $M_t/M/c$ queuing model with first-come first-served discipline (see Figure 3): multiple servers serve jobs that arrive according to a $\lambda(t)$-rate Poisson process[3] and have exponentially distributed service times ([13]). For simplicity, we assume that the customers entering the system during staffing interval i all arrive at t_i. Performance is evaluated at $\Delta_p, 2\Delta_p, ..., T$ and customer abandonments that occur before τ are not

[3]Typically, $\lambda(t)$ is sinusoidal and mimics customer load trends seen in Figure 1

counted. We also assume that servers will work overtime at their shift end to finish the ongoing service instance if any.

For any shift vector $\mathbf{w}$, the queueing process is simulated repeatedly over the planning period. $\mathbf{w}$ is a satisfactory candidate if it violates the performance constraint less than α times (in proportion of the number of simulations). Otherwise, the first t_i such that $\mathbb{P}(W_{t_i} > \tau) > \alpha$ is stored in the set $\mathbf{B}$ of unfeasible. We implement a Python module provided in Appendix A. To the best of our knowledge, no simulation script algorithm taking this set of assumptions into consideration is provided in the literature, even though [9] presents a sophisticated mathematical approach to check on the performance constraint.

3.3 Inputs

Using the method proposed by [5], the lower bound is determined as follows: in each staffing interval, s_i^{LB} is set equal to the smallest capacity level that is needed to meet the performance constraint, assuming that infinite capacity is available in all other staffing intervals. Indeed, for staffing interval i, the only customers to serve are those who just arrived in the system, as the ones arrived at earlier times were all served (by an infinite number of servers).

We design a straight-forward method to find $\mathbf{s^{UB}}$. At any t_i, the customer load follows a Poisson process of rate $\lambda(t_i)$. Let k_i be the smallest integer k that satisfies $\mathbb{P}(X_i \leq k) \geq 1 - \alpha$, where X_i is the random variable referring to the customer load at t_i. Then, we choose $s_i^{LB} = k_i$. This way, as long as the customer load X_i is smaller or equal to k_i, every customer is served without waiting time. Otherwise, some customers experience a delay but it happens with probability α. Thus, the performance criteria is satisfied.

Finally, we choose $\mathbf{s^{init}} = \mathbf{s^{UB}}$.

3.4 Discussion - Suboptimality and efficiency

One limitation of the branch-and-bound method attributed to [5] is that it selects only one minimum shift vector at each node. Thus, possible alternatives to problem (3)-(5) are not taken into consideration. While the proposed minimum-cost shift vector might be unfeasible, alternative shift vectors with identical costs could satisfy the performance condition.

On the other side, the efficiency of the method heavily relies on $\mathbf{s^{UB}}$ and $\mathbf{s^{LB}}$. Specifically, the higher $\mathbf{s^{UB}}$, the higher the number of nodes in the tree. The total number of unique nodes is:

$$\prod_{i=1}^{p}(s_i^{UB} - s_i^{LB} + 1) \tag{7}$$

This quantity grows with $\mathbf{s^{UB}}$. As we estimate $\mathbf{s^{UB}}$ roughly (for simplicity reasons), we have to hope that the fathom rules are strong enough to silent tree nodes efficiently.

Moreover, determining $\mathbf{s^{LB}}$ is computationally intense as it requires to run simulations repeatedly for each staffing interval i from 1 up to $\mathbf{s_i^{LB}}$, especially when customer load in so-called staffing period is high. This step can be viewed as exploring a tree but without fathoming rules.

At last, we choose $\mathbf{s^{init}} = \mathbf{s^{UB}}$ for simplicity reasons. A more efficient way would be to find a more accurate initial solution. To do so, [10] proposes an algorithm that could be investigated in further research. This would enable us to use fathom rules at the maximum of their efficiency.

4 Case study

Let Agnes be the Operations manager of a fast delivery company. She needs to plan the number of couriers needed next week in London. The company runs every day from 9am to 10pm. As in Figure 1, the daily demand curve shows two peaks representing food demand increases at lunch and dinner. The company is meant to satisfy this demand by getting couriers to deliver the items.

The couriers work in *slot mode*: a schedule with limited amount of slots (determined by a working area and a time window) is communicated to the fleet one week ahead of time. The drivers can register to the slots they want as long as some slots remain unfilled (see [14]). Then, the rider has to be connected on the company app during the selected shifts and will be allocated to incoming deliveries thanks to a back-end dynamic dispatch logic (including distance to pickup locations, among others). More importantly, during these shifts, s/he is eligible to a minimum hourly salary called "base fare" but is paid per delivery: if s/he completes many deliveries and earns more than this fare, s/he keeps the extra money earned. Otherwise, s/he gets the base fare.

Without loss of generality, the planning horizon and the working zone can be restricted to one day and one area (say West Central London) respectively. The company policy in terms of quality of service is as follows: "A courier must be appointed within the first 9 minutes following the delivery request for at least 90% of the requests". Thanks to a data analysis, Agnes knows that the demand flow can be assumed steady over 20-min windows and thus decides to divide the day into 20-min staffing intervals. By using predictive techniques, she claims that demand rate $\lambda(t)$ follows a sinusoidal pattern fluctuating around the average λ_0:

$$\lambda(t) = \lambda_0(1 + R\cos(c_1 t + c_2))$$

where t expressed in minutes (9am = 0 min, 12am = 180 min,...), $c_1 = 1.496 * 10^{-2}$, $c_2 = -2.917$ (scaled for two peaks near 12pm and 7pm), R a variation amplitude factor. The service time follows a 16-min average exponential process. Note that the service involves a pick up time (go to the store), a waiting time at pick up (wait the item at the store) and a drop off time (deliver the item to end-customer).

4.1 Fixed shift costs

4.1.1 Framework

We start with the classic case for which shift costs are expressed in worker hours. Shifts are pre-defined and depicted on Figure 4 along with the demand rate curve fitted for $\lambda_0 = 50$, $R = 0.6$. Figure 5 displays the corresponding shifts and costs.

4.1.2 Computational effort

We implement a Python script to solve the shift scheduling problem. We use PuLP library to solve the linear programs at each node. As discussed in 3.4, determining $\mathbf{s^{LB}}$ can be computationally intense: all staffing intervals are initially set to 1 and increased following the determination rule explained in 3.3. However, we observe by simulation that the only staffing intervals i such that $s_i^{LB} > 1$ are $i = 7, 10$ i.e between 11:00am-11:20am and 12:00pm-12:20pm. This result can be seen by analyzing Figures 4 and 5. Shift 2 is the only active shift between 11:00am-11:20am. The only way to satisfy the demand entering the system at 11:00am is thus to fill in this shift. The

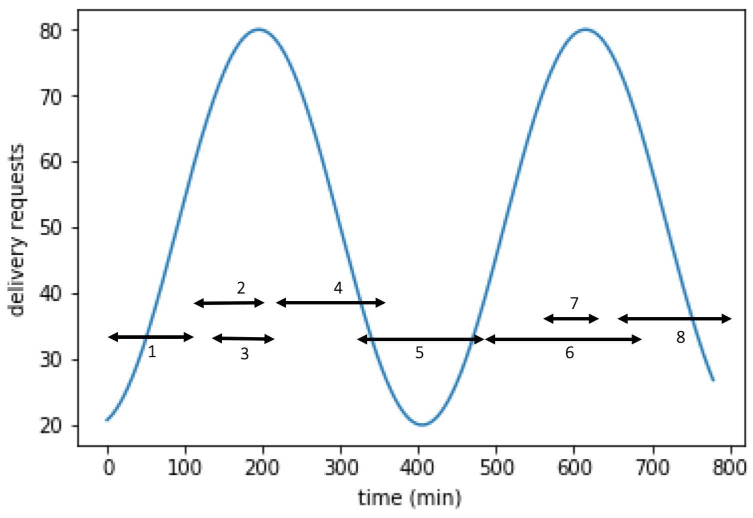
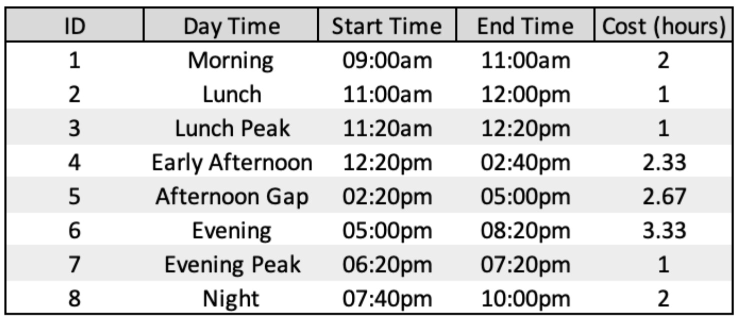


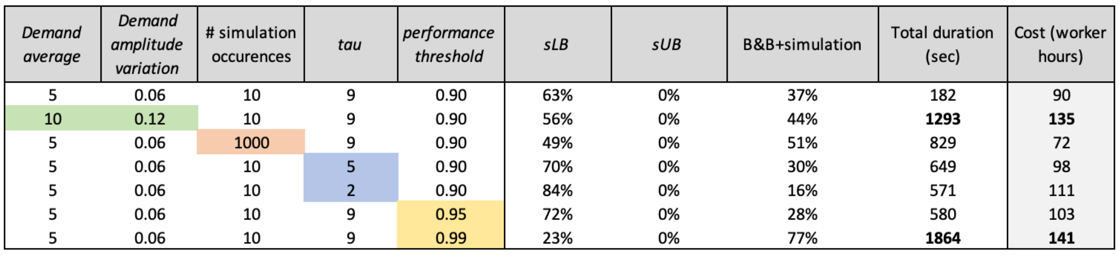
Figure 4: Demand curve and pre-defined shifts



ID	Day Time	Start Time	End Time	Cost (hours)
1	Morning	09:00am	11:00am	2
2	Lunch	11:00am	12:00pm	1
3	Lunch Peak	11:20am	12:20pm	1
4	Early Afternoon	12:20pm	02:40pm	2.33
5	Afternoon Gap	02:20pm	05:00pm	2.67
6	Evening	05:00pm	08:20pm	3.33
7	Evening Peak	06:20pm	07:20pm	1
8	Night	07:40pm	10:00pm	2

Figure 5: Shift times and costs

same logic applies for shift 3 at 12pm. Note that this does not apply for any other shift. Indeed, let us have a look at shift 5 at 3pm. It is actually the only active shift at this time. s_i^{LB} is the lowest staffing level satisfying the performance constraint at t_i, assuming that all the other capacities are infinite. Hence, staffing levels is infinite on 3:20pm-3:40pm. As shift 5 is also the only active shift on 3:20pm-3:40pm, the only way the problem (3)-(5) satisfies the infinite constraint on 3:20pm-3:40pm is by filling in the shift 5, regardless of staffing level between 3:00pm-3:20pm. Therefore, the staffing level at 3pm stays equal to 1.



Demand average	*Demand amplitude variation*	# simulation occurences	*tau*	*performance threshold*	*sLB*	*sUB*	B&B+simulation	Total duration (sec)	Cost (worker hours)
5	0.06	10	9	0.90	63%	0%	37%	182	90
10	0.12	10	9	0.90	56%	0%	44%	**1293**	**135**
5	0.06	1000	9	0.90	49%	0%	51%	829	72
5	0.06	10	5	0.90	70%	0%	30%	649	98
5	0.06	10	2	0.90	84%	0%	16%	571	111
5	0.06	10	9	0.95	72%	0%	28%	580	103
5	0.06	10	9	0.99	23%	0%	77%	**1864**	**141**

Figure 6: Computational efforts - Time processing shares. Parameters: λ_0 = demand average; R = demand amplitude variation; τ = tau.

Our test computer has a 2.7 GHz Intel Core i5 processor with 8GB RAM. Figure 6 presents the time shares for each step of the algorithm for different parameter instances. All results are compared to the first row configuration $\lambda_0 = 5$; $R = 0.06$. As we could expect, costs are increased by more restrictive performance requirements. The limitation mentioned in section 3.4 regarding $\mathbf{s^{LB}}$ is proven. Finding suitable lower bounds accounts for more than half the processing time in most cases. The processing time increases tenfold when waiting time tolerance decreases from 5% to 1%. Setting such a low tolerance seems to be more costly than providing twice more costumers ($\lambda_0 = 10$; $R = 0.12$). This implies that excessive performance requirements might be detrimental. The processing times are quite high even for such small-scale instances and we see that it can quickly skyrocket with respect to the customer load size. Finally, increasing the number of simulation occurrences leads to higher quality solution, at the expense of a higher computation time.

4.2 Weighted stochastic shift costs

We now focus on a more applicable case. As mentioned earlier, the costs are not proportional to the number of worked hours in practice: indeed, they depend on the couriers' efficiency. Anytime a delivery is completed, the company earns a commission

and gives the remaining to the courier. However, if the courier's hourly earnings fall below the base fare, the company compensates and therefore, loses money. Moreover, shifts that overlap do not necessarily have the same cost. This is particularly the case for peak hours slots. As a matter of fact, peak hours are important for the company's business as good quality service during these periods are critical for customer retention. Such fast delivery companies are willing to give extra incentives to couriers (such as extra pounds per delivery) especially when the demand is unexpectedly high or the operating fleet size is low (on rainy days for instance).
Therefore, rather than being linearly correlated to their duration, shifts depend on parameters whose impact on fleet is hardly predictable: wind, temperature, rain intensity, traffic congestion, etc. Again, with some data analysis, Agnes concludes that shift costs $c_1, c_2, c_3, ...$ follows Normal laws of parameters $(\mu_1, \sigma_1), (\mu_2, \sigma_2)....$

How does this change our model?
In the model developed above, we assumed that the costs were expressed in worker hours. With this assumption, we have been able to compare staffing costs with shift costs in Step 1 (section 3.1) because they were expressed in the same unit. It enabled us to use the Fathom[$\mathbf{c_s}$] rule and to fathom nodes inexpensively. As shift costs are now uncorrelated to staffing costs (still expressed in worker hours), we only keep the fathom rules that compare shift costs to each other i.e all the rules except the Fathom[$\mathbf{c_s}$] applied in Step 1. Apart from that, the model remains the same. All (shift) costs are compared to the lower feasible shift cost so far. However, we can expect longer processing durations. First, we removed one fathom rule. Secondly, the shift costs are not deterministic anymore and we must therefore simulate the whole shift scheduling method repeatedly. Each replication uses a set of shift costs generated to follow the above-mentioned Normal distributions.

5 Conclusion

We have presented a branch-and-bound method that breaks down a general shift scheduling problem into shift covering integer subproblems. It uses simulations to check on the performance quality of the subproblem solutions. Tree exploration is boosted by the means of 5 fathom rules. Although the method does not guarantee an optimal solution, we have observed that higher quality solutions could be obtained by increasing the number of simulation occurences.
Finding staffing vector inputs is a demanding task but can speed up the search tree

process. More particularly, determining a lower bound vector represents a large part of the overall processing time. To be able to apply the method to larger scales, especially for industrial applications, further research should be dedicated to questions of optimality, robustness and input vector search efficiency. In this sense, a discussion may be found in [5].

References

[1] Athanassios N. Avramidis , Wyean Chan, Pierre L'Ecuyer (2009), *Staffing multi-skill call centers via search methods and a performance approximation*, IIE Transactions, 41:6, 483-497.

[2] Stefan Helber, Kirsten Henken (2010) *Profit-oriented shift scheduling of inbound contact centers with skills-based routing, impatient customers, and retrials*, OR Spectrum, Volume 32, Issue 1, p 109–134.

[3] Júlíus Atlason, Marina A. Epelman, Shane G. Henderson (2007), *Optimizing Call Center Staffing Using Simulation and Analytic Center Cutting-Plane Methods*, Management Science, Vol. 54, No 2.

[4] Armann Ingolfsson, Fernanda Campello, Xudong Wu, Edgar Cabral (2010), *Combining integer programming and the randomization method to schedule employees*, European Journal of Operational Research, Vol. 202, Issue 1, p 153-163.

[5] Mieke Defraeye, InnekeVan Nieuwenhuyse (2016) *A branch-and-bound algorithm for shift scheduling with stochastic nonstationary demand*, Computers and Operations Research, Vol. 65, p 149-162.

[6] Mehmet Tolga Cezik, Pierre L'Ecuyer (2008) *Staffing Multiskill Call Centers via Linear Programming and Simulation*, Management Science, Vol. 54, No 2.

[7] George B. Dantzig, (1954) *A Comment on Edie's "Traffic Delays at Toll Booths".*, Journal of the Operations Research Society of America 2(3):339-341.

[8] Raphael Hauser *Integer Programming Notes*, Lecture Notes for B6.3 Integer Programming module at University of Oxford, MT 2018-2019.

[9] Linda V. Green, João Soares (2007) *Computing Time-Dependent Waiting Time Probabilities in M(t)/M/s(t) Queuing Systems*, Manufacturing and service operations management, Vol. 9, No 1.

[10] Mieke Defraeye, InnekeVan Nieuwenhuyse (2013) *Controlling Excessive Waiting Times in Emergency Departments: An Extension of the ISA Algorithm*, Available at SSRN: https://ssrn.com/abstract=1967004 or http://dx.doi.org/10.2139/ssrn.1967004

[11] Frederico Samartini et al. (2011) *Upper Bounds on Performance Measures of Heterogeneous M/M/c Queues*, Mathematical Problems in Engineering, Vol. 2011.

[12] *https://www.mckinsey.com/industries/travel-transport-and-logistics/our-insights/how-will-same-day-and-on-demand-delivery-evolve-in-urban-markets*, McKinsey and co., 2017.

[13] $https : //en.wikipedia.org/wiki/Queueing_theory$, Queueing theory. See "Kendall's notation" part.

[14] $https : //www.youtube.com/watch?v = Ww69GUEIgVI$, Staffomatic, Shift Planner Deliveroo.

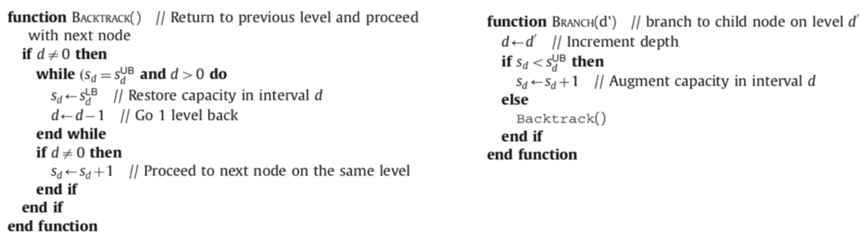
A Simulation module

```python
import numpy as np
from copy import deepcopy

def generate_service_time(shape, scale=1):
    """ Generates 1 realisation of a exponential-distributed random
    variable. This counts as the duration for a customer service. """

    return  np.random.gamma(shape, scale, 1)

class Customer:

    def __init__(self, shape, scale, arrival_time, service_start_time=None,
                 service_end_time=None, wait=None):

        self.arrival_time = arrival_time
        self.service_start_time =  service_start_time
        self.service_time = generate_service_time(shape, scale)
        self.service_end_time = service_start_time
        self.wait = wait

    def print_details(self):
        """ Prints the details of the node. """

        print("arrival_time:", self.arrival_time)
        print("service_start_time:", self.service_start_time)
        print("service_time:", self.service_time)
        print("service_end_time:", self.service_end_time)
        print("wait:", self.wait)

def Simulate(node, rates, shapeGamma, scaleGamma, times, shiftMatrix,
             tau=9, thres=0.1, nbOccurences=10, interval_length=20):
    """ Simulates operations over the planning horizon with specified shift
    vector (repeats 'nbOccurences' times). Returns True if the quality service
    constraints is satisfied. Otherwise, returns False and the index of the
    first staffing period at which the constraint has been violated. """

    waiting_times_summary = {}
    # Simulate 'nbOccurences' times
    for occurence in range(nbOccurences):
        poissonParams = {}
        Customers=[]
        count_per_interval = {}

        # Set workforce for each staffing period
        for time in times:
            n = int(time/interval_length)
            count_per_interval[str(time)] = sum(shiftMatrix[:,n]*node.shift_vector)

        # Generate customer load for each staffing period
        for time in times:
            # +1 to avoid no demand situation (demand = 0)
            poissonParams[str(time)] = np.random.poisson(
                    rates[times.index(time)]) + 1

        # Generate incoming customers (service time follows exponential law)
        for time in times:
            demand = poissonParams[str(time)]
            for i in range(demand):
                Customers.append(Customer(shapeGamma,
                                          scaleGamma,
                                          float(time)
                                          )
                )

        # Simulate queuing
        time = times[0]
        count = count_per_interval[str(time)]
        workers = [time for i in range(int(count))]

        for i in range(len(Customers)):
            if Customers[i].arrival_time > time:
                time = int(Customers[i].arrival_time)
                count = count_per_interval[str(time)]
                workers = [time for t in range(int(count))]
            workers = sorted(workers)

            if workers[0] >= time + interval_length:
                k = (workers[0] - time)//interval_length
                time += k*interval_length
                try:
                    count = count_per_interval[str(int(time))]
                    workers = [time for t in range(int(count))]
                except:
                    for j in range(i,len(Customers)):
                        Customers[j].wait = tau + 10000
                    break

            start_time = workers[0]
            Customers[i].service_start_time = start_time
            Customers[i].service_end_time = Customers[i].service_start_time+Customers[i].service_time
            Customers[i].wait = Customers[i].service_start_time - Customers[i].arrival_time
            workers[0] = Customers[i].service_end_time

        # Flag customers with exceeding waiting times
        demand_amplitude = deepcopy(poissonParams)
        waiting_times = dict.fromkeys(list(demand_amplitude), 0)
        for customer in Customers:
            if customer.wait > tau:
                waiting_times[str(int(customer.arrival_time))] += 1

        # Generate performance summary for each staffing interval
        for time in waiting_times:
            waiting_times[time] /= demand_amplitude[time]
            try:
                waiting_times_summary[time] += waiting_times[time]
            except:
                waiting_times_summary[time] = waiting_times[time]

        # Compare performance to threshold for each staffing interval
        # Return lowest under-performing staffing interval if any.
        for time in waiting_times_summary:
            waiting_times_summary[time] /= nbOccurences
        if waiting_times_summary[time] > thres:
            return False, int(int(time)/interval_length)

    return True, None
```

B Branch-and-bound pseudocode

```
function ExploreTree()
    Set initial feasible staffing vector $\mathbf{s}^{\mathsf{init}}$
    Determine $\mathbf{w}^{\mathsf{init}}$

    $\mathbf{w}^* \leftarrow \mathbf{w}^{\mathsf{init}}$    // Initialize best feasible solution so far
    $\tilde{\mathbf{s}}_{\mathbf{w}^*} \leftarrow \tilde{\mathbf{s}}_{\mathbf{w}^{\mathsf{init}}}$
    $\tilde{c}_{\mathbf{w}^*} \leftarrow \tilde{c}_{\mathbf{w}^{\mathsf{init}}}$

    Set bounds $\mathbf{s}^{\mathsf{LB}}$ and $\mathbf{s}^{\mathsf{UB}}$

    $\mathbf{s} \leftarrow \mathbf{s}^{\mathsf{LB}}$    // Initialize root node
    $d \leftarrow 0$    // Initialize depth

    repeat
        $\boxed{\text{Determine } c_{\mathbf{s}}}$
        if $c_{\mathbf{s}} \geq \tilde{c}_{\mathbf{w}^*}$ then
            Fathom[$c_{\mathbf{s}}$]:
            $s_d \leftarrow s_d^{\mathsf{LB}}$    // Restore capacity in interval $d$
            $d \leftarrow d - 1$    // Go to parent node.
            Backtrack()
        else
            $\boxed{\text{Solve Problem (2-4)}}$ to obtain $\{\mathbf{w}, \tilde{c}_{\mathbf{w}}, \mathbf{s}_{\mathbf{w}}\}$
            if $\tilde{c}_{\mathbf{w}} \geq \tilde{c}_{\mathbf{w}^*}$ then
                Fathom[$\tilde{c}_{\mathbf{w}}$]:
                $s_d \leftarrow s_d^{\mathsf{LB}}$    // Restore capacity in interval $d$
                $d \leftarrow d - 1$    // Go to parent node.
                Backtrack()
            else
                $\boxed{\text{Check if } \mathbf{w} \text{ has been simulated before}}$
                if $\mathbf{w} \in \mathbf{B}$ : then
                    $i^e \leftarrow i^e$ of previously simulated solution
                    if $d < i^e$ then
                        Branch($i^e$)    // Proceed to child node on level $i^e$
                    else    // $d \geq i^e$ so branching to underlying nodes is useless
                        while ($d > i^e$) do
                            $s_d \leftarrow s_d^{\mathsf{LB}}$    // Restore capacity in interval $d$
                            $d \leftarrow d - 1$    // Go 1 level back
                        end while
                        Backtrack()
                    end if
                else
                    $\boxed{\text{Simulate } \mathbf{s}_{\mathbf{w}}}$ to obtain $i^e$
                    if Feasible then
                        $\mathbf{w}^* \leftarrow \mathbf{w}$    // Update best solution so far
                        $\mathbf{s}^*_{\mathbf{w}} \leftarrow \mathbf{s}_{\mathbf{w}}$
                        $widetildec_{\mathbf{w}^*} \leftarrow widetildec_{\mathbf{w}}$

                        Fathom[$\mathbf{w}^*$]:
                        $s_d \leftarrow s_d^{\mathsf{LB}}$    // Restore capacity in interval $d$
                        $d \leftarrow d - 1$    // Go to parent node.
                        Backtrack()
                    else    // Infeasible solution
                        if $d < i^e$ then
                            Branch($i^e$)    // Proceed to child node on level $i^e$
                        else    // $d \geq i^e$ so branching to underlying nodes is useless
                            Fathom[$\overline{i^e}$] :
                            while ($d > i^e$) do
                                $s_d \leftarrow s_d^{\mathsf{LB}}$    // Restore capacity in interval $d$
                                $d \leftarrow d - 1$    // Go 1 level back
                            end while
                            Backtrack()
                        end if
                    end if
                end if
            end if
        end if
    until ($d = 0$)
end function
```

Figure 7: Branch-and-bound pseudocode. See Figure 8 for side functions. From [5]



```
function BACKTRACK()   // Return to previous level and proceed
   with next node
  if $d \neq 0$ then
    while ($s_d = s_d^{UB}$ and $d > 0$ do
      $s_d \leftarrow s_d^{LB}$   // Restore capacity in interval $d$
      $d \leftarrow d - 1$   // Go 1 level back
    end while
    if $d \neq 0$ then
      $s_d \leftarrow s_d + 1$   // Proceed to next node on the same level
    end if
  end if
end function
```

```
function BRANCH(d')   // branch to child node on level $d'$
  $d \leftarrow d'$   // Increment depth
  if $s_d < s_d^{UB}$ then
    $s_d \leftarrow s_d + 1$   // Augment capacity in interval $d$
  else
    Backtrack()
  end if
end function
```

Figure 8: Side functions. From [5].