Does Laziness Pay Off? - A Lazy-Constraint Approach to Timetabling

Torsten Klug[1]
LBW Optimization GmbH, Berlin, Germany

Markus Reuther
LBW Optimization GmbH, Berlin, Germany

Thomas Schlechte
LBW Optimization GmbH, Berlin, Germany

Abstract

Timetabling is a classical and complex task for public transport operators as well as for railway undertakings. The general question is: Which vehicle is taking which route through the transportation network in which order? In this paper, we consider the special setting to find optimal timetables for railway systems under a moving block regime. We directly set up on our work of [8], i.e., we consider the same model formulation and real-world instances of a moving block headway system. In this paper, we present a repair heuristic and a lazy-constraint approach utilizing the callback features of Gurobi, see [3]. We provide an experimental study of the different algorithmic approaches for a railway network with 100 and up to 300 train requests. The computational results show that the lazy-constraint approach together with the repair heuristic significantly improves our previous approaches.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Moving Block, Railway Track Allocation, Timetabling, Train Routing

Digital Object Identifier 10.4230/OASIcs.ATMOS.2022.11

1 Introduction

We consider the general timetabling problem *GTTP* presented in [8]. In this problem the decisions to make are:

- Which train request is cancelled and not routed?
- Which route over time does a train take through the network to meet its requested time intervals?
- Where and how does overtaking take place, i.e., where does deceleration, waiting on a side track, and acceleration take place?

The literature on timetabling is overwhelming as the numerous Chapters in [1] document, see the surveys by [4] and [2] and the significant and recent works [5] and [6]. However, for moving block systems, i.e., trains running in braking distance, less papers are published, e.g., [9] and [8]. In this paper, we consider the model formulation presented in [8] without modifications. We directly setup on the multi-layer graph structure based on a velocity expansion and the definition of *competitions* in order to resolve conflicts between trains, i.e., by using disjunctive tandem and opposite headway constraints. We will give a very compact presentation of the mixed-integer programming model in Section 2, see [8] for the details. In the current follow-up paper, we exclusively focus on the solution methodology. The contribution of this paper is to discuss and compare different algorithmic approaches that we present in Section 3, i.e., using lazy-constraints and a primal repair heuristic. In particular,

[1] corresponding author

in Section 4 we show the effects of these enhancements on the entire solution process. We provide computational results for a railway network with 100, 200 and 300 requested trains. The results demonstrate that the new algorithmic add-ons halve the runtime on average and allow to solve larger instances to optimality.

2 Model

Let us revive the notation used in [8] for the binary variables: $y_a,\ a \in A$ to represent routing decision via arcs of the velocity-expanded graph $D = (V, A)$, $x_e^{r_1 \prec r_2}$ for the order of two trains r_1, r_2 on an infrastructure edge e, slacks u to allow cancelling train requests, and the continuous variables t to model departure and arrival times. Let A, B, C, D, I, N, G be appropriate matrices and d, f, g vectors, respectively, to represent the MIP model for the *GTTP* defined in [8]. A compact MIP formulation is then:

$$\begin{aligned}
\min\quad & c_u^T u + c_t^T t + c_y^T y && && (1)\\
& Iu \qquad + Ny = d && \texttt{(routing)} && (2)\\
& \qquad At + By \le f && \texttt{(timing)} && (3)\\
& \qquad Ct + Dy + Gx \le g && \texttt{(headway)} && (4)
\end{aligned}$$

The constraints partition into three parts: the `routing` part (2) that models train cancellations and the y-flow throw the velocity-expanded graph; the `timing` part (3) that combines time and flow variables to model running times meeting departure and arrival time windows at requested stops; the `headway` constraints (4) that ensure the minimal safety headway distances between the train's paths.

The headway constraints are the crucial part that couples the different train requests into one integrated optimization problem. Without these the problem decomposes into independent routing problems for each train.

In reality, most of the headway constraints are automatically satisfied by the network's shape and the request set. In fact, only a few of those constraints become intrusive, which motivate their handling by a lazy-constraint approach.

Consider the following example representation of a headway constraint:

$$t^{r_1} + \sum_{\dots} h_a\, y_a^{r_1} \le t^{r_2} + M \cdot \left(3 - x_e^{r_1 \prec r_2} - \sum_{\dots} y_a^{r_1} - \sum_{\dots} y_a^{r_2}\right) \qquad (5)$$

The constraint (5) is defined for the request pair (r_1, r_2) and the infrastructure edge e. The big-M constraint becomes active if and only if r_1 goes first ($x_e^{r_1 \prec r_2} = 1$) and both requests route via edge e, i.e., $\sum y_a{}^{r_1} = 1$ and $\sum y_a{}^{r_2} = 1$. The minimal required headway time between time variables t^{r_1} and t^{r_2} is denoted by h_a. Thus, enough spacing between different train events is triggered if the constraint becomes active.

In that construction, three nearly independent situations, i.e., the specific request order on e and the two routing decisions need to come together, to activate the constraint. We exploit this structure by releasing one of three situations to deactivate, i.e., repair, a violated headway constraint. The repair heuristic presented in Section 3 is based on this idea.

3 Algorithmic Add-Ons

Due to the constantly evolving power of MIP solvers, solving sequences of relaxed MIP formulations, in that we temporarily give up some *missing constraints*, are a way to solve large scale models. In each *MIP iteration* we resume at least one or more of the missing constraints if we detect violations. See, e.g., [7] for the TSP. A similar concept is the iterative MIP approach used in [8] applied to timetabling. This seems to be a promising approach if two key properties are present:

- the missing set of constraints is too large (or even exponentially growing w.r.t. the input size) to be handled directly and
- only a few of the missing constraints are relevant to cut-off enough infeasibilities to obtain a feasible primal solution within only some MIP iterations.

In this Section we present a straight-forward enhancement of this approach by using lazy-constraints. The obvious motivation behind this is that if several MIP iterations are needed, the explored branch-and-bound trees sum up and the computational effort

3.1 Recall the Iterative MIP Approach (BC)

Let us briefly recall the branch-and-cut motivated sequential MIP approach (BC) used in [8]: The method starts with the MIP formulation as presented in Section 2, but without the ordering variables $x_e^{r_1 \prec r_2}$ and without the headway constraints (4). We denote this relaxed MIP by *RMIP*. Solving *RMIP* always provides an integer feasible routing which may violate some headway constraints. If no headway constraint is violated, we already found an optimal solution of GTTP. Otherwise, we determine all violated headway constraints including required train ordering variables and add them to the model (simultaneous constraint and variable generation). A violation, i.e., a headway conflict, belongs to a specific location (node or edge) and velocity. To save iterations, we always add all headway constraints for all possible velocity levels. We call the subset of headway constraints for a location and request pair with all possible velocity combinations a *competition*. We then continue solving the resulting *RMIP* and repeat the process until an optimal solution to GTTP is found. The computational results in [8] already shown that this approach terminates much faster and with a much smaller model than the full GTTP model.

3.2 Lazy-Constraint Approach (LAZY)

The lazy-constraint approach makes use of the Gurobi lazy-constraint callback. The method starts with the MIP formulation *GTTP* without the headway constraints (4). We denote these relaxed sub-models with only restricted subsets of headway constraints by *RMIP*. In contrast to BC, for technical reasons all ordering variables $x_e^{r_1 \prec r_2}$ are added in advance (even if they have effect). Whenever Gurobi determines a feasible integer solution for *RMIP* in its branch-and-bound algorithm (or already at the root node), the lazy-constraint callback is called. In the callback we check the current solution and determine the violated headway constraints in the same manner as described in 3.1 for the BC case. If no violated headway constraint is found then the solution is feasible for *GTTP*. Otherwise, all headway constraints of the superordinate competitions are added as lazy-constraints to *RMIP*. This procedure proceeds and terminates with an optimal solution for the *GTTP*.

3.3 Primal Repair Heuristic (PRH)

Both of the previous approaches check a given feasible solution of the relaxation *RMIP* and add violated headway constraints. We observed that after a couple of iterations the number of violated headway constraints decrease significantly. Nevertheless, the solution of the previous iteration is cut off and the MIP solver has problems to find a feasible primal solution satisfying the newly added headway constraints. To give a little help, we implement the following primal repair heuristic PRH that derives a global primal feasible solution as follows.

Let (u', y', x', t') be a feasible solution fo the relaxation *RMIP* and V be the set of train requests for that a violated headway constraint was detected. Let H be set of the violated headway constraints. We construct a conflict graph with a node for *all* train requests and an edge between two nodes if there exists a constraint in H for the corresponding train request pair. We chose a maximal stable set S in this graph. Then the following applies:

- each isolated node of the conflict graph is an element of S,
- for each pair $r_1, r_2 \in V$ that is part of the same headway constraint in V at most one of the corresponding nodes is an element of S.

If we cancel all train requests, i.e, by setting the slack variable u_r to 1, that are not in the stable set S, then a primal feasible solution (u, y, x, t) of *GTTP* is defined as follows:

$$t = t',\ x = x',\ y = \begin{cases} y_a^r = (y_a^r)', & r \in S \\ y_a^r = 0, & \text{otherwise} \end{cases},\ u = \begin{cases} u_r = (u_r)', & r \in S \\ u_r = 1, & \text{otherwise} \end{cases} \tag{6}$$

It is easy to see that the `routing` and `timing` constraints are satisfied. In Section 2 we already mentioned that a headway constraint can be deactivated if the variables y_a of one of the involved requests are set to zero. This is guaranteed by the stable set construction and hence all determined headway constraints are satisfied. Since all violated headway constraints are determined the constructed solution is also globally feasible.

4 Computational Results

In this section we discuss the computational results and give answers to the following questions:

- What is the impact of the primal repair heuristic PRH?
- Does the lazy-constraint callback outperform the iterative approach?

We consider three sets of instances with an increasing number of train requests from 100 to 300. The OneHND testset is exactly the same as in [8] and consists of 10 scenarios with 100 trains. The TwoHND and ThreeHND testsets consist of 5 scenarios with 200 trains and 3 scenarios with 300 trains, respectively. The testsets have the same data basis with an increasing time horizon. Concrete numbers and more details on the railway application can be found in [8]. All tests were executed on a Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz with 90 GB RAM. We use Gurobi 9.51 as MIP solver with up to 4 threads. The time limit was 12 hours. The maximal optimality gap is set to 10^{-4}.

We compare four algorithms. These are the approaches BC and LAZY without the primal repair heuristic and its variants with the primal repair heuristic PRH. The variants with PRH are denoted by BC-P and LAZY-P, respectively. Table 1 shows the aggregated results for the different algorithm variants and all testsets. The first column lists the considered algorithm followed by the testset in column two. The third column indicates the number of scenarios that could be solved to optimality and the number of scenarios in the testset. The next four columns give the minimum, maximum, average and summed up computation times in seconds. Finally, the last two columns denote the average number of branch-and-bound nodes and the average number of generated headway constraints. In the case of BC the number of branch-and-bound nodes is the sum over all BC iterations.

The results of Table 1 highlight the performance boost by the primal repair heuristic, i.e, comparing the summed up and average computation time in seconds of BC with BC-P and LAZY with LAZY-P, respectively. We restrict the evaluation of the repair heuristic to the OneHND testset because without PRH the major part of the greater scenarios could not

Table 1 Summary of the results for the different algorithm variants and testsets.

algorithm	testset	# optimal	computation time				# B&B nodes	# headways
			min	max	average	sum	average	average
BC	OneHND	10/10	6	837	207	2070	3731	2293
BC-P	OneHND	10/10	6	389	111	1111	**1760**	**2042**
LAZY	OneHND	10/10	7	862	261	2606	22485	3270
LAZY-P	OneHND	10/10	8	151	**99**	**991**	4463	2393
BC-P	TwoHND	4/5	127	43201	8125	48753	123623	**8651**
LAZY-P	TwoHND	5/5	217	17247	**3765**	**22589**	**68415**	10691
BC-P	ThreeHND	2/3	1857	43201	13701	54804	201874	**15471**
LAZY-P	ThreeHND	3/3	1203	12691	**5147**	**20587**	**70133**	17334

be solved to optimality within the 12 hour time limit. Both approaches benefit from using PRH, so that the total and average runtime is halved. For the lazy-constraint approach the repair heuristic is crucial, since LAZY is notable slower than BC. In all scenarios where BC wins Gurobi needs a reasonable time to provide a primal feasible solution and therefore the branch-and-bound tree becomes large. The repair heuristic fixes this issue and significantly reduces the tree size especially for the larger and more complex scenarios.

The superiority of the lazy-constraint approach increases with the problem size. LAZY-P is 12 seconds faster on average for the OneHND testset. For the TwoHND and ThreeHND testset LAZY-P requires less than half of the computing time of BC-P. Furthermore LAZY-P is able to solve all scenarios to optimality. This is not the case for BC-P with two scenarios that cannot be solved to optimality within the time limit. The number of generated headway constraints are in the same range for both approaches. In comparison to BC-P the number of branch-and-bound nodes of LAZY-P is on average 45% smaller for the TwoHND testset and 65% smaller for the ThreeHND testset. Since the iterative branch-and-cut approach restarts the branch-and-bound procedure at each iteration the restart overhead sums up with the problem size. The lazy-constraints approach add the generated headway constraints within the branch-and-bound procedure and do not have to create already explored subtrees again.

The detailed results for the single scenarios can be found in Table 2 and Table 3. The first column gives the unique scenario id. It follows the algorithm; the average computation time in seconds; the number of routed requests; the final objective value; the final optimality gap in percent and; the number branch-and-cut iterations. Finally, the last two columns denote the number of branch-and-bound nodes and the number of generated headway constraints. As before the number of branch-and-bound nodes is the sum over all iterations of BC.

The following two extreme cases give a hint when which algorithm should be used. Considering scenario 35 in Table 3 the number of iterations and the number of branch-and-bound nodes for algorithm BC-P is 19. This means Gurobi can find the optimal solution of the relaxed problem at the root node of each iteration and is therefore faster than LAZY-P. In contrast to that scenario 42 could be solved by LAZY-P within the time limit and needs less than 25% of the branch-and-bound nodes of BC-P. Furthermore BC-P only provides a solution with a gap of about 25% within the time limit.

Table 2 Detailed results for the OneHND testset.

scenario	algorithm	computation time	# requests routed	objective	gap in percent	# BC iterations	# B&B nodes	# headway constraints
25	BC	35	99	12657.77	0.00	15	15	869
25	BC-P	**16**	99	12657.83	0.00	7	**7**	**791**
25	LAZY	184	99	12662.75	0.04		8579	2177
25	LAZY-P	83	99	12660.46	0.02		2887	1434
26	BC	**6**	96	41821.69	0.46	5	3	131
26	BC-P	**6**	96	41821.69	0.46	5	**2**	**131**
26	LAZY	7	96	41821.69	0.00		168	290
26	LAZY-P	8	96	41821.69	0.00		157	441
27	BC	69	98	22049.13	0.27	15	**10**	1449
27	BC-P	**46**	98	22049.13	0.00	15	13	**1401**
27	LAZY	98	98	22050.33	0.01		4581	2638
27	LAZY-P	118	98	22051.19	0.01		6221	2323
28	BC	137	98	22112.66	0.00	23	1572	2471
28	BC-P	**25**	98	22116.17	0.02	10	**10**	**1592**
28	LAZY	94	98	22113.25	0.00		3577	2545
28	LAZY-P	141	98	22114.91	0.01		7201	3270
29	BC	**63**	97	32025.42	0.01	16	**16**	2530
29	BC-P	87	97	32022.49	0.00	24	23	2833
29	LAZY	91	97	32024.66	0.01		6116	2664
29	LAZY-P	72	97	32023.32	0.01		2526	**2079**
30	BC	234	96	42241.08	0.01	25	5358	**2695**
30	BC-P	389	96	42238.54	0.00	39	9975	2895
30	LAZY	251	96	42239.90	0.00		16111	3798
30	LAZY-P	**107**	96	42238.75	0.00		**5310**	2784
31	BC	119	97	32001.54	0.00	22	1946	1853
31	BC-P	**25**	97	32003.22	0.01	10	**10**	**1305**
31	LAZY	622	97	32004.12	0.01		65958	5313
31	LAZY-P	120	97	32006.99	0.02		6624	2724
32	BC	520	96	42086.50	0.01	43	8344	5604
32	BC-P	**110**	96	42081.67	0.00	24	**144**	4016
32	LAZY	328	96	42085.64	0.01		28405	5405
32	LAZY-P	116	96	42081.06	0.00		4968	**3552**
33	BC	**49**	98	22120.64	0.00	15	**244**	**1724**
33	BC-P	108	98	22120.52	0.00	21	1117	2052
33	LAZY	70	98	22121.11	0.00		2880	2200
33	LAZY-P	75	98	22123.11	0.01		3006	2205
34	BC	837	97	32249.30	0.01	32	19800	3604
34	BC-P	297	97	32249.54	0.01	23	6299	3399
34	LAZY	862	97	32252.74	0.02		88479	5668
34	LAZY-P	**151**	97	32251.90	0.02		**5730**	**3121**

Table 3 Detailed results for the TwoHND and ThreeHND testset.

scenario	algorithm	computation time	# requests routed	objective	gap in percent	# BC iterations	# B&B nodes	# headway constraints
35	BC-P	**127**	194	65095.85	0.01	19	**19**	**1761**
35	LAZY-P	217	194	65096.47	0.00		3782	2061
36	BC-P	**1133**	194	64318.40	0.00	24	**15660**	**6480**
36	LAZY-P	1577	194	64318.14	0.00		30173	9938
37	BC-P	1701	191	94415.08	0.00	27	**23445**	**6291**
37	LAZY-P	**1306**	191	94414.63	0.00		28094	8543
38	BC-P	2038	192	84379.15	0.00	35	**24424**	**8097**
38	LAZY-P	**1597**	192	84382.17	0.00		34120	9792
39	BC-P	43201	193	74940.93	13.31	55	554568	**20628**
39	LAZY-P	**17247**	194	64974.44	0.01		**245908**	23121
40	BC-P	3664	291	96779.69	0.00	29	44368	**9585**
40	LAZY-P	**1731**	291	96789.00	0.01		**21190**	10995
41	BC-P	6083	286	146914.49	0.00	26	79033	**13465**
41	LAZY-P	**4962**	286	146910.75	0.00		**60731**	17082
42	BC-P	43201	289	117369.01	25.42	55	482222	**23363**
42	LAZY-P	**12691**	292	87548.56	0.01		**128479**	23926

Finally, we answer the question: Does Laziness Pay Off? Our computational experiments indicate that the answer is yes with two minor restrictions. First, the problem size or complexity must be large enough and second a primal heuristic is needed, such as the presented simple repair heuristic PRH. Under these conditions, we conclude that the lazy-constraint approach outperforms the iterative branch-and-cut approach on the large instances considered.

References

1 Ralf Borndörfer, Torsten Klug, Leonardo Lamorgese, Carlo Mannino, Markus Reuther, and Thomas Schlechte, editors. *Handbook of Optimization in the Railway Industry*, volume 268. Springer, 2018. doi:10.1007/978-3-319-72153-8.

2 Valentina Cacchiani, Dennis Huisman, Martin Kidd, Leo Kroon, Paolo Toth, Lucas Veelenturf, and Joris Wagenaar. An overview of recovery models and algorithms for real-time railway rescheduling. *Transportation Research Part B: Methodological*, 63:15–37, 2014. doi:10.1016/j.trb.2014.01.009.

3 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL: https://www.gurobi.com.

4 Steven S. Harrod. A tutorial on fundamental model structures for railway timetable optimization. *Surveys in Operations Research and Management Science*, 17(2):85–96, 2012. doi:10.1016/j.sorms.2012.08.002.

5 Leonardo Lamorgese and Carlo Mannino. An exact decomposition approach for the real-time Train Dispatching problem. *Operations Research*, 63:48–64, 2015. doi:10.1287/opre.2014.1327.

6 Bianca Pascariu, Marcella Samà, Paola Pellegrini, Andrea D'Ariano, Joaquin Rodriguez, and Dario Pacciarelli. Effective train routing selection for real-time traffic management: Improved model and aco parallel computing. *Computers & Operations Research*, 145:105859, May 2022. doi:10.1016/j.cor.2022.105859.

7 Ulrich Pferschy and Rostislav Staněk. Generating subtour elimination constraints for the TSP from pure integer solutions. *Central European Journal of Operations Research*, 25(1):231–260, February 2016. doi:10.1007/s10100-016-0437-8.

8 Thomas Schlechte, Ralf Borndörfer, Jonas Denißen, Simon Heller, Torsten Klug, Michael Küpper, Niels Lindner, Markus Reuther, Andreas Söhlke, and William Steadman. Timetable optimization for a moving block system. *Journal of Rail Transport Planning & Management*, 22:100315, June 2022. doi:10.1016/j.jrtpm.2022.100315.

9 Peijuan Xu, Francesco Corman, Qiyuan Peng, and Xiaojie Luan. A train rescheduling model integrating speed management during disruptions of high-speed traffic under a quasi moving block system. *Transportation Research Part B*, 104:638–666, 2017. doi:10.1016/j.trb.2017.05.008.