optimising railways

SMA and partners Ltd

Gubelstrasse 28
8050 Zurich
Switzerland
Phone +41 44 317 50 60
info@sma-partner.com
www.sma-partner.com

sma

A Real-world Algorithm Implementation Using the Algorithm Platform Posted on GitHub

An implementation of an algorithm based on SPOT using Viriato's Algorithm Platform.

Abstract

SPOT [1] is a mathematical model for strategic passenger railway planning building on the well-known PESP (Periodic Event Scheduling Problem) [2]. The goal of the SPOT model is to obtain an automatically generated and workable timetable during the strategic planning phase as it aims to provide a passenger-centric timetable.

We want to provide an implementation based on **SPOT** using Viriato's Algorithm Platform to deliver a software prototype that can be actually used by a subject-matter expert in practice so that the model's results can be assessed by them without any mathematical or programming background. We demonstrate the benefits that come with our Algorithm Platform to the researcher.

Our Goals for this Implementation

We want to highlight the benefits that come with our Algorithm Platform to the researcher:

– **Data Acquisition and Provision.** The Algorithm Platform retrieves all data requested by the algorithm from Viriato's database and provides it via an interoperable REST interface. There is no need to write database queries.

– **Rapid Development.** The input data provision and the simple way of passing parameters in combination with the predefined domain data model (AIDM) reduce the development effort considerably.

– **Prevention from Misuse.** Relying on the Algorithm Platform reduces the chance for the algorithm developer to make errors, and also protects them from erroneous data due to the enforced invariants in Algorithm Platform's Algorithm Interface.

– **Visualisation of Results and Reports.** The user can easily explore the solution which was written back to Viriato, allowing them to inspect the structure of the results in the available modules and assess their correctness and quality.

In addition, reports in form of Excel sheets are generated to present the parameters used and a summary of the solution to the user giving them insights.

Note that our implementation of **SPOT** deliberately deviates in some aspects from the original model in order to enhance the applicability in practice. The main goal was to demonstrate the use of the Viriato Algorithm Platform rather than an analysis of the model.

Introducing SPOT

As the input for **SPOT** we are given a list of origin-destination ("OD") relations describing the travel wishes of a set of passengers, and a set of trains that have a list of defined commercial stop locations and minimum travelling times between these node pairs.

The **SPOT** model routes passengers through a train network from their origin to their destination by modifying the arrival and departure times of the trains, therefore impacting the total travel times (consisting of transfer times at stations and travel times on trains) of the passengers being routed. The objective is to minimise the total travel time for all passengers of all OD relations.

Figure 1 provides an example timetable generated with **SPOT**. In our small example, let there be only one OD relation with origin A and destination D. The given trains travel only from A to B, B to C and C to D. Therefore, passengers have to change twice, once in B and again in C. In this simple case where the only relation is for demand from A to D, **SPOT** has optimised the arrival and departure at both changes, such that the transfer time is minimal.

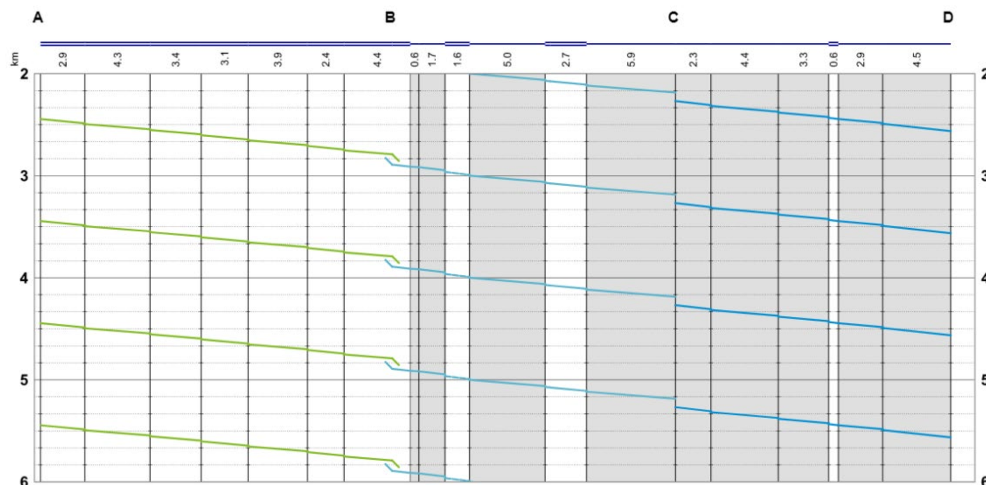

Figure 1 A graphic timetable for the example with two transfers. (Viriato screenshot).

For this example, we defined five minutes as the required minimum transfer time. The connection clock in Figure 2 allows us to verify that the transfer time at B is minimal, since the train to C departs exactly five minutes after the arrival of the train from A.

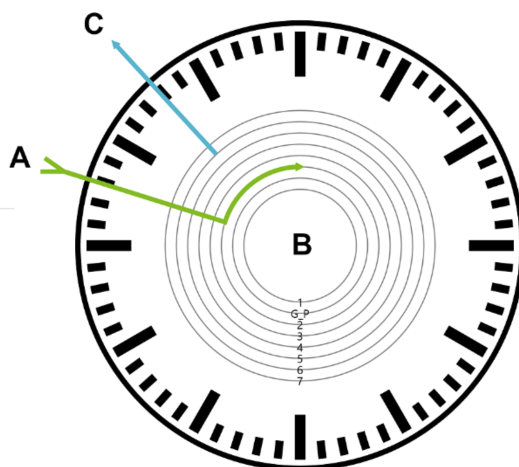

Figure 2 A connection clock for station B. (Viriato screenshot)

Deviations from the Theoretical SPOT Model

Our **SPOT** implementation deviates from the original in [1] in several places for sake of practical applicability.

Cancelling of Commercially Irrelevant Parts of Trains

The **SPOT** model optimises travel times only for those parts of a train run which are used on a passenger's route. Depending on the input data, it may happen that parts of a train run are not used by any passenger in a **SPOT** solution, and this implies that the travel time on these parts are not necessarily optimised. Therefore, it can happen that a train runs with an unrealistically slow speed on some track section in the solution. In practice, a train often spends a large part of this "idle" time stationary at a station. For sake of simplicity, we cancel these sub-parts in a solution if they occur at the beginning or at the end of a train run, and completely cancel and remove trains which not used at all by the algorithm. Figure 3 shows a timetable, where irrelevant parts of scheduled trains have been cancelled.

Using the Algorithm Platform's C# API wrapper methods CancelTrain(...), CancelTrainBefore(...) and CancelTrainAfter(...) this is a straightforward operation.

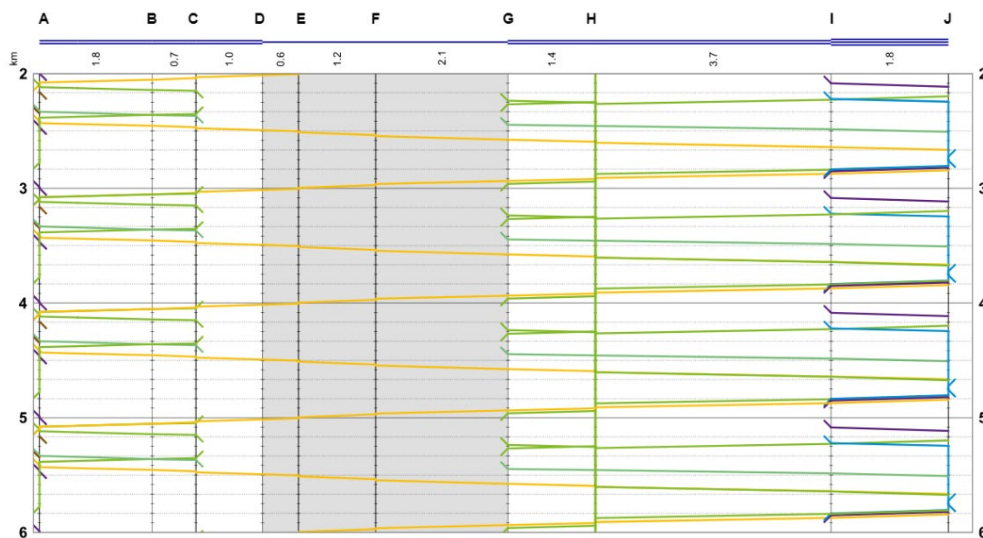

Figure 3 A graphic timetable where all commercially irrelevant parts of trains have been cancelled. (Viriato screenshot).

See 'Cancelling a Train' on GitHub for a code example

Simplified Passenger Routing

For sake of simplicity, in this case we chose to neglect the waiting time at a passenger's origin station. Moreover, at present we assume that all passengers of an *OD* relation select the same travel route. This assumption contrasts with the implementation in [1], where the waiting time at the origin *is* considered. Therefore, not all passengers of an *OD* relation travel via the same route. The difference between these two cases can be seen in Figure 4.

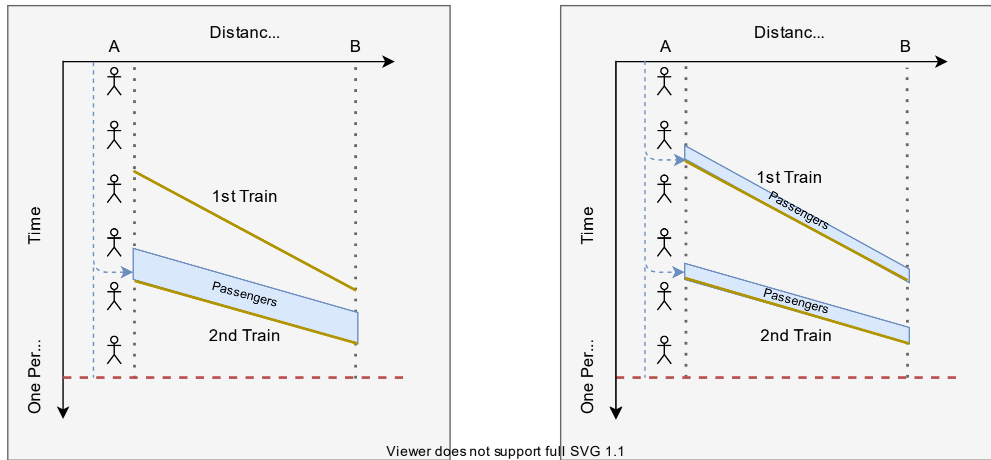

Figure 4 Simplified passenger routing (left), original **SPOT** implementation (right).

A solution of our implementation is shown on the left of Figure 4, where because waiting at *A* is disregarded, all passengers on the relation *A*-*B* use the faster 2nd train. In contrast, on the right hand side we can observe the solution that would be achieved given the full implementation of the **SPOT** algorithm as defined in [1]. As the waiting time at *A* is taken into account, some passengers use the slower train since in their case it leads to an overall shorter passenger trip time than waiting for the faster train.

Specific Features of the *Algorithm Platform* in Action

We will present here some features of the *Algorithm Platform* which assisted with the implementation of this prototype.

Reading Line Characteristics

Trains with their basic characteristics (travel route, minimum running and dwell times, etc.) from a pool of trains can be used by our prototype to build a **SPOT** solution. In Viriato, these *template trains* can be defined conveniently, as shown in Figure 5, and our prototype queries them directly from the *Algorithm Platform*.

Train run | Train configuration

Add | Remove | Enter departure times | Set arrival time fixed | Filter nodes | Modify run time reserves | Open platform / track graphic

Append | Reroute | Constant total run time | Set departure time fixed | Add joined train unit | Create graphic timetable | Calculate

Section	Sect. track	Node	Platform	Track info	ST	Arrival	Departure	Run time	Add Run	Sup run	Stop time	Add stop	Sup stop	km	v
		[illegible]					06:07.2							0.000	
75	35	[illegible]					06:09.6	2.2	0.2					1.43	6
75	3	[illegible]					06:10.5	0.9	0.0					2.64	0
75	304	[illegible]					06:12.7	1.6	0.1	0.5				5.05	6
75	1	[illegible]					06:13.1							5.65	
75	106	[illegible]					06:14.0	1.2	0.1					6.82	2
7	3	[illegible]					06:14.4	0.4	0.0					7.51	03
750	107	[illegible]			11	06:16.4	06:18.4	1.8	0.2		2.0			9.92	2
75	10	[illegible]					06:21.3	2.7	0.2					13.70	8
75	111	[illegible]					06:22.0	0.7	0.0					15.08	18
75	11	[illegible]					06:24.3	1.7	0.1	0.5				18.78	7
75	114	[illegible]	2				06:25.3	0.9	0.1					20.63	11
75	11	[illegible]					06:27.9	2.0	0.1	0.5				24.30	5
75	121	[illegible]					06:29.1	1.1	0.1					26.67	18
75	92	[illegible]					06:30.3	1.1	0.1					28.94	14
75	45	[illegible]				06:31.9		1.0	0.1	0.5				29.95	8

Figure 5 A train run (data anonymised) that can be used as a template train in SPOT. (Viriato screenshot).

See 'Reading Line Characteristics' on GitHub for a code example.

Parameter Passing

The *Algorithm Platform* and the C# API wrapper provide a convenient way to pass user-defined parameters to **SPOT**. For example, our implementation allows the setting of a bound on the number of allowed transfers for any passenger on their route. Before the algorithm is started, the user enters the desired value using a dialog, as shown in Figure 6.

Choose Algorithm
Algorithm
SPOT
Port
8082
Parameters
Show advanced options
Template Trains From Scenario for SPOT
Scenario SPOT
Validity 01.01.2018, 02.01.2018
Filepath to CSV-File with OD-Routes for SPOT
exported.csv
Time Window of First Cycle
01.01.2018 00:00:00 01.01.2018 01:00:00
Maximal Number of Transfers per Route
2
Default Minimal Duration of a Transfer [Minutes]
5
Maximal allowed additional Running Time [%]
75
Number of Cycles
8
Timeout for MIP Solver [Seconds]
60
Set to Default OK Cancel

Figure 6 The dialog to enter the parameters before starting the algorithm. (Viriato screenshot).

See Section 'Passing Parameters' on GitHub for a code example.

Writing Back Planned Trains to Viriato

If a template train defined in the section 'Reading Line Characteristics' on GitHub is selected by the algorithm as forming part of the the solution of the **SPOT** model, then the external algorithm will write it back to the *Algorithm*

Platform. This consists of two steps. First, the template train is declared to be in the solution set. Then its arrival and departure times, including reserves, are updated, and written back to the *Algorithm Platform*. The resulting timetable can be inspected with Viriato's graphic timetable functionality, see Figure 7.

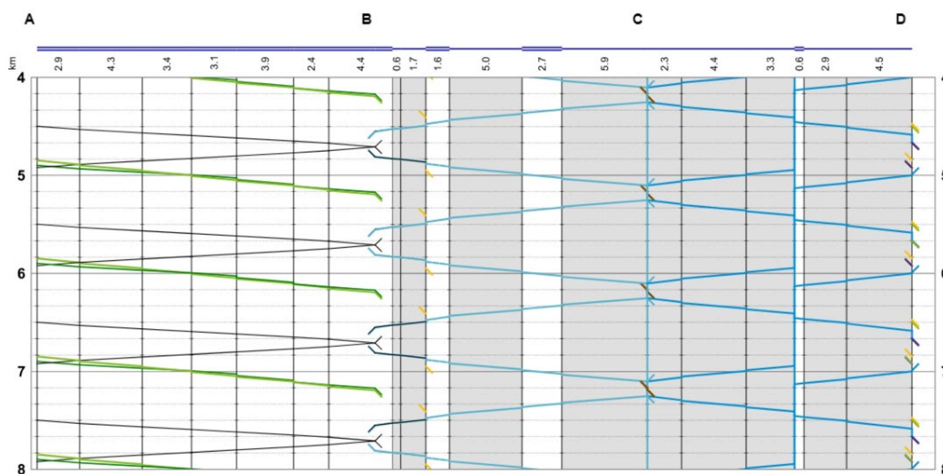

Figure 7 A resulting graphic timetable. (Viriato screenshot)

See the section 'Writing Trains Back to Viriato' on GitHub for a code example.

Excel Reports

In addition to the persisted timetable results, we also provide Excel sheets that our implementation populates with KPIs, allowing subsequent evaluation and comparison of different solutions. See Figure 8 and Figure 9 for a sample of this output. Generating these reports programmatically is straightforward using the *CreateTable(...)* and related methods with the C# API wrapper.

	A	B	C	D	E
1	Origin Node	Destination Node	Total Travel Time	Transfer Time	Total Time
2	A	B	0:15:18	0:05:00	0:20:18
3	A	C	0:09:12	0:05:48	0:15:00
4	A	D	0:17:36	0:05:48	0:23:24
5	A	E	0:20:12	0:00:00	0:20:12
6	A	F	0:09:48	0:05:00	0:14:48
7	B	A	0:16:24	0:05:00	0:21:24
8	B	C	0:03:42	0:00:00	0:03:42
9	B	D	0:12:06	0:05:00	0:17:06
10	B	E	0:37:06	0:05:00	0:42:06
11	B	J	0:57:42	0:24:42	1:22:24
12	B	K	0:25:36	0:00:00	0:25:36
13	C	A	0:09:12	0:05:00	0:14:12

Figure 8 Travel time composition per relation generated by Viriato and exported to Excel

	A	B
1	Total Travel Time	Passengers without a Route
2	57:12:12	4512
3		

Figure 9 A summary sheet generated by Viriato and exported to Excel

References

[1]: 2021 G. J. Polinder, M. Schmidt and D. Huisman, *Timetabling for strategic passenger railway planning*, Transportation Research Part B: Methodological, DOI: https://doi.org/10.1016/j.trb.2021.02.006

[2]: 1989 P. Serafini and W. Ukovich, *A mathematical model for periodic scheduling problems*, SIAM Journal on Discrete Mathematics DOI: https://doi.org/10.1137/0402049