Report 2

Fedir Kovalov

June 17, 2024

1 Program

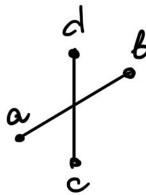
The current program is designed to solve the scheduling problem by using constraint programming and simple graph algorithms. It defines the problem as a job-shop scheduling problem and solves it using an integer programming solver.

1.1 Input

The program takes three arguments – track graph, routes, and so called "optroutes".

1.1.1 Track graph

The graph is represented as an adjacency list in plain text, with vertices represented as numbers. Vertices in the graph are supposed to represent parts of railroad track. The graph both contains information important to the scheduling and routing algorithms. For the routing, the graph maintains an adjacency lists for all vertices. This sounds simple enough, and this does work for routing, but this is not enough scheduling. Consider the following figure. It shows a



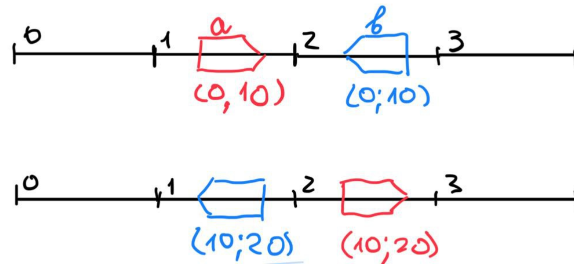
schematic representation of the real x-crossing. In this case, if two train would be crossing a to b and c to d then they would crash, so these parts of the track must be understood as conflicting to the scheduler. At the same time however, trains cannot go from b to d or a to c. However, in our simplistic model this relationship would be impossible to represent: if we make this entire track as a single vertex(making the scheduler to avoid already mentioned position), we would inevitably have to make b and d adjacent. On the other hand, if we make

$\{a, b\}$ and $\{c, d\}$ as single vertices, the graph would not contain any information about the fact that they must be conflicting.

So, in my model, the graph assigns conflict list to each vertex. These are lists of vertices that the scheduler would not put trains at the same time together (e.g. it defines two vertices as mutually exclusive). In our example, $\{a, b\}$ and $\{c, d\}$ would be single vertices, but $\{a, b\}$'s conflict list would include $\{c, d\}$, and vice versa. They feature allows for more flexible graph definitions than more simplistic representation.

1.1.2 Routes

Routes are basically paths in the track graph. Routes represent, well, train routes. They contain all important data for the scheduler to decide on their modification and scheduling. For this they contain a list of so called route vertices, which contain two things: vertex(track), and time that the train must spend in the track to *exit* it. However this data alone isn't enough to do proper scheduling. Consider the following figure. Suppose for both routes a and b they



both must spend 10 in both sections 1 and 2. In this case, since intervals do not overlap, it can schedule trains to go through each other, which is obviously undesirable. Additionally, it indicated the fact that this model would have no ability to express that train don't immediately transition from one track to another.

To mitigate this, routes feature one more parameter, simply called *overlap*. It indicated the amount of time will the train spend in both sections. This must roughly correspond to the time it takes for the train to travel it's own length. The track time indicates the time from when the train's front to enters the section to when it's front leaves it.

The program however does not have to recieve full description of routes, it can construct them on it's own. The routes that are constructed by the program itself are called optroutes. All they are, is a description of start, end and overlap. The program then constructs possible routes and passes them to the scheduler to construct a proper schedule.

1.2 What it is doing?

After receiving all the input, program first finds all possible routes for the optroutes. It collects them in an array and passes down to the actual scheduler program. The scheduler then constructs a cp-sat model. The cp-sat model is constructed as any job-shop model would be. The only modifications of traditional job-shop are that the no overlap constraints on jobs also include intervals on conflicting vertices and we make the sat choose between different variants of optroute to decide for best schedule.

1.3 Limitations

1.3.1 Efficiency

There are two main things the program spends time on: finding routes for optroutes, and scheduling by cp-sat. The path finding is done via a simple BFS on the graph, which does take $O(n^n)$ time (where n is the number of verticies). However I would argue, this is plausible for the task, and here is why:

- Track graph doesn't change much: we can cache results of the searches and reuse them over and over. It is safe to assume that movement on railroads it not too chaotic, so recalculation of paths in a graph is rarely needed. (though this is not actually implemented)
- Track graph is not that connected: providing train cannot reverse it's direction easily (which is true for most of them) there is only a portion of the graph that the train can ever traverse. For one, track layouts typically do not include loops and can be subdivided in biconnected components, which would make the graph traversal more efficient. (this is also not implemented)

Overall, though time complexity of the BFS is far from efficient, it is plausible for the task. The scheduling itself is done by cp-sat, which basically is a solver for Pure Integer Linear Programming problem, which is NP-complete. This is not great, but it is limited by the fact that the scheduling problem itself is NP-complete.

1.3.2 Graph incompleteness

Track graph in the present version does not feature any description of how long the tracks are. This results in an inaccurate estimation (or rather lack thereof) of track time. This is partly because it wouldn't really change the point of the program, and I didn't have an actual schematic with lengths that this feature may be fully tested on. However, this feature would be trivially easy to implement from the program architecture standpoint.

1.3.3 Integer programming

Since CP-SAT is a pure integer solver, all the values inputted are also integers. This however shouldn't be a huge limitation, since the granularity needed for an accurate solution shouldn't be more than simply seconds.

1.3.4 Ignorance of inertia

Probably the biggest limitation of this program is that it ignores inertia of the trains. Because the routes are calculated independently from the scheduling, it cannot take into account occasional stops scheduler might generate. This means, schedule might contain rapid accelerations and decelerations that might not be possible in reality. This is unfortunately the limitation of the linear programming nature of the scheduler, that complicates implementation of (non-linear) speed calculation. Though this still leaves a lot of uses even for such crude scheduler (for example depots, where trains rarely accelerate faster that even 30km/h), this remains a significant issue.

2 Examples

In the program files there are two examples. One of them is a simpler track model(useful for testing the conflict constrains), and another is a partial model of Oslo's main station layout.

Oslo's model has a bit unusual graph. In fact, the numbers on the schematic do not correspond to the actual graph vertex numbers. The thing is, that simple graph vertices do not convey any information about the direction of the train. If a track vertex is adjacent both to tracks a and b, no matter the direction the train arrived at it. Avoiding over complication of code, Oslo example divides each vertex i into two – vertex $i * 2 - 1$ and $i * 2$. Then, all vetrices that are adjacent to this track from the left.

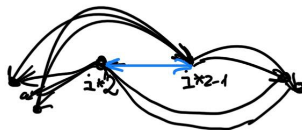


Figure 1: Visualisation of the double vertex scheme. All the connection coming from the left connect to the vertex $i * 2 - 1$ and all connection from the right come into $i * 2$ vertex.

You can run examples by passing these arguments to the program:

```
./RailScheduler --routes tests/simple1/routes.txt \
--track_graph tests/simple1/track_graph.txt \
--optroutes tests/simple1/optroutes.txt

./RailScheduler --routes tests/large/routes.txt \
--track_graph tests/large/track_graph_gen.txt \
--optroutes tests/large/optroutes.txt
```

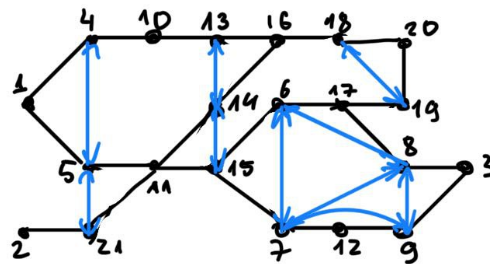


Figure 2: Schematic of the "simple1" example. Blue lines represent conflicting vertices relationship.

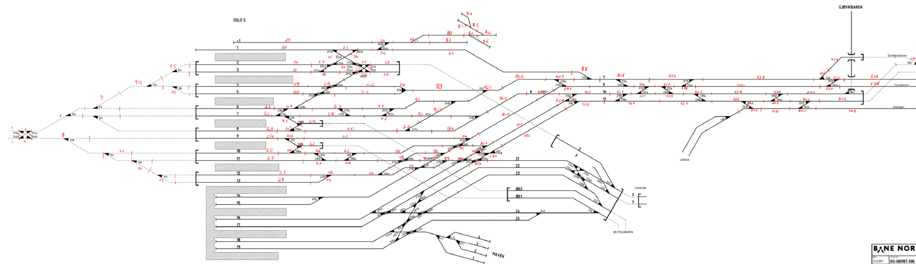


Figure 3: Schematic of Oslo of "large" example. (full resolution picture is on the project website)