Railway Time-Tabling Effort

Milind Sohoni, Narayan Rangaraj and others
http://www.cse.iitb.ac.in/~sohoni
The WR Network

- 28 stations
- Over 200 track segments
- around 1000 services daily
- 67 rakes (physical trains)
- over 300 junctions/points
Objectives

Inputs

- The Physical Network
  stations, lines, platforms.
- Operational Norms
  Headway, turn-around times.
- Patterns of Operation
  such as CCG-VR fast
- Requirements
  Specific as well as aggregates

Outputs

- TimeTable
detailed timings.
- Rake-Links
  alloting physical EMUs.
- Platform Charts.
The Network

Stations
- Name
- No. of Platforms
- No. of Stables
- Turn-around Time
- Push-In/Pull-Out Time

**Churchgate**

*Turn-around = 2 min*

*Number of Platforms = 4*

*In = 3 min, Out = 2 min*

*Stable = 3*

Lines-unidirectional
- Start/End Station
- Duration (time)
- Headway (time)
- Fork/Join List

**CCG - MEL - Ft**

*CCG*  

*length = 4 min, headway = 3 min*

**MEL**
Pattern

This encapsulates a typical and repeating pattern of operation.

<table>
<thead>
<tr>
<th>pattern-id</th>
<th>Station</th>
<th>Stoppage</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-CCG-BVI-Ft</td>
<td>Churchgate</td>
<td>-</td>
<td>CCG-BCT-Thru</td>
</tr>
<tr>
<td></td>
<td>BombayCT</td>
<td>[1,1]</td>
<td>BCT-DDR-Thru</td>
</tr>
<tr>
<td></td>
<td>Dadar</td>
<td>[1,2]</td>
<td>DDR-BAN-Thru</td>
</tr>
<tr>
<td></td>
<td>Bandra</td>
<td>[1,1]</td>
<td>BAN-KHR-Thru</td>
</tr>
<tr>
<td></td>
<td>Khar</td>
<td>[0,0]</td>
<td>KHR-ADH-Thru</td>
</tr>
<tr>
<td></td>
<td>Andheri</td>
<td>[1,4]</td>
<td>ADH-BVI-Cross</td>
</tr>
<tr>
<td></td>
<td>Borivili</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Thus, *pattern* is a *path with time prescribed flexibility* in the network.
The \textit{vptt}

The \textit{vptt} is the input as well as the output. Fields are
(i) service-id and desired pattern
(ii) required start-time interval
(iii) actual start and end-times
(iv) rake-links

<table>
<thead>
<tr>
<th>service-id</th>
<th>pattern-id</th>
<th>start-time</th>
<th>start</th>
<th>end</th>
<th>rake-link</th>
</tr>
</thead>
<tbody>
<tr>
<td>BVI-647</td>
<td>9-CCG-BVI-Ft</td>
<td>18:20</td>
<td>CCG</td>
<td>BVI</td>
<td>PROP-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18:26</td>
<td>18:23</td>
<td>19:24</td>
<td></td>
</tr>
<tr>
<td>PROP-3</td>
<td>9-BVI-CCG-Su</td>
<td>19:24</td>
<td>BVI</td>
<td>CCG</td>
<td>ADH-751</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19:30</td>
<td>19:30</td>
<td>19:24</td>
<td></td>
</tr>
<tr>
<td>ADH-751</td>
<td>9-CCG-ADH-Sw</td>
<td>20:19</td>
<td>CCG</td>
<td>ADH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>20:24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Constraints

- **line constraints** - services using the same track must be headway apart.

- **platform constraints** - these must be available for halts at stations.

- **continuity constraints** - train departs a station and enters a line and vice-versa, and follows the pattern.
Other Constraints

- Fork

- Join
**Solvers**

**Manual Aids**

- Check a TT
- Move services
- Order Services

**Automatic**

- based on CHIP C++ Constraint Solver. Allows constraints to be posted on variables. Follows clever branch-and-bound

- Partial Order on services \( S \) by *time*
  Partition into clubs \( S_1, S_2, \ldots, S_k \)

- Solve \( S_i, S_{i+1} \) together. Freeze \( S_i \) and move to \( S_{i+1}, S_{i+2} \)

**Compute-Instensive**: Takes 50 minutes for half (UP) service set.
How to define $S_i$’s?

- Organize the services in a **temporal partial order**.
- Pick bunch $S_1$ by peeling off the top few, then $S_2$ and so on.

- services $s_i$ and $s_j$.
- depart-times $t_i$, $t_j$.
- patterns $p$ and $p'$

If $d_i - d_j \geq T(p, p')$ then $s_j < s_i$. 
Rake-Linking

Once the services have been scheduled, they have to be provisioned: assign one of the 64 rakes available with WR.

**Step 1 Form the Service Graph.**

- Vertices: Services
- Edges: Possible Successor

```
<table>
<thead>
<tr>
<th></th>
<th>CCG 72</th>
<th>BVI 180</th>
</tr>
</thead>
</table>
```

Direct Edge

```
<table>
<thead>
<tr>
<th></th>
<th>BVI 192</th>
</tr>
</thead>
</table>
```

Indirect Edge

```
<table>
<thead>
<tr>
<th></th>
<th>CCG 72</th>
<th>BVI 180</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>ADH 221</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>CCG 302</th>
</tr>
</thead>
</table>
```
Step 2 Compute Chain Decomposition.

- DAG
- Min-Cost-Flow and its variants
- Extremely Fast and provably optimal: 2 minutes
Platform Allocation

Inputs

- Service In-Out times
- Service Platform Preferences
- Set of Platforms

<table>
<thead>
<tr>
<th>Service</th>
<th>In</th>
<th>Out</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rajdhani</td>
<td>18:56</td>
<td>18:57</td>
<td>4</td>
</tr>
<tr>
<td>Virar Local</td>
<td>18:54</td>
<td>19:05</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>Dahanu Shuttle</td>
<td>19:01</td>
<td>19:11</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Output

- Platform Allocation for each service
- Clash report if impossible
Undifferentiated and differentiated

Theorem: For the undifferentiated case, if at no point are there more than $P$ services, then all services can be assigned platforms.

Thus a necessary condition is sufficient. However there is no such theorem in the differentiated case.

<table>
<thead>
<tr>
<th>Service</th>
<th>Platform 1</th>
<th>Platform 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rajdhani</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>Virar Local</td>
<td>0 2 1,2</td>
<td></td>
</tr>
<tr>
<td>Dahanu Shuttle</td>
<td>1 2 2</td>
<td></td>
</tr>
</tbody>
</table>
The Algorithm-undiff

- Let $A = \{a_1, \ldots, a_n\}$ be the set of arrival instants. Similarly, let $D$ be the set of departure instants.
- Let $L = (l_1, l_2, \ldots)$ be the list $A \cup D$ sorted by time.
- Maintain Allotment table $at[i]$.

![Diagram showing the algorithm-undiff process]

$L[i]$:arrival $\rightarrow at[i-1]$  

$L[i]$:departure $\rightarrow at[i]$
The Algorithm-diff

- Form $L$ as before.
- Maintain $AT[i]$, the collection of all possible $at[i]$. This is essentially Dynamic Programming.

\[ AT[i-1] \rightarrow L[i]:arrival[2,4] \rightarrow AT[i] \]
Algorithm continued

- If $AT[i]$ is empty for some $i$ then declare infeasible.
- Otherwise, pick an $at \in AT[\text{last}]$ and trace back.

Complexity: Good for WR. Virar takes 20 seconds.