An Application Of Tabu-Enhanced Genetic Search To A Railway Optimization Problem

Team Off-by-one

Consisting of

Lennart Carstens-Behrens
lennart.carstens-behrens@stud.htwk-leipzig.de
Oliver Herrmann
oliver.herrmann@stud.htwk-leipzig.de

Students of the

HTWK Leipzig

January 10, 2022

Abstract

The informatiCup 2022 introduced a railway optimization problem in which trains should bring passengers from one station to another via a railway network with as little waiting time as possible. This paper describes an application of tabu-enhanced genetic search to the railway optimization problem.

Contents

1 Introduction

The 17'ths informatiCup[1] by the German Informatics Society (GI[2]) - started in october 2021 - introduced a railway optimization problem. The background to this is, that public rail transport in germany will gradually be changed to a coordinated timetable, which aims to reduce passenger waiting times by having local and long-distance trains arrive at key hubs at coordinated times. This timetable is described by Deutsche Bahn as "Taktfahrplan [3]" or "coordinated timetable".

1.1 Task

The task of the informatiCup 2022 includes the creation of a timetable for a rail network with the aim of minimizing the total delay of all passengers. The model and the timetable are provided and to be returned in a given format. The formats and more details about the task can be found in the GitHub repository[4].

2 Literature Review

The railway optimization problem falls under the computational time complexity class NP-Hard. Which means that no efficient algorithm exists for the problem. Knowing that no efficient solution exists is helpful, according to Steven S. Skienna [Skiena(2008)], as the algorithm designer can productively focus on finding an approximation instead of searching for an algorithm that solves the problem effiently in nondeterministic polynominal-time.

2.1 Approximation Algorithms

The goal of an approximation algorithm is to quickly find a good but not optimal solution which can be improved with longer runtime eventually. Al-

[1]informatiCup `https://gi.de/informaticup`

[2]Gesellschaft Für Informatik `https://gi.de/`

[3]Taktfahrplan`https://www.deutschlandtakt.de/blog/so-funktioniert-der-taktfahrplan/`

[4]Repository `https://github.com/informatiCup/informatiCup2022`

gorithms that prooved to work well for optimization problems are Genetic search, Simmulated Annealing and Tabu Search.

2.1.1 Genetic Algorithms

Genetic algorithms where first suggested as a search method for optimization problems in 1975 by J. Holland [Holland(1975)].

A Genetic algorithm borrows biologic concepts by emulating ”survival of the fittest” principle. The chance of a solution to ”survive” increases with its fitness which is given by an objective function. Furthermore surviving solutions are ”mutated” by applying random alterations.

2.1.2 Simulated Annealing

Simulated Annealing is inspired by a physical process of cooling in thermodynamics, where the energy state of a system is described by the energy state of each particle within the system. The energy state of a particle jumps randomly, this transition is determined by the temperature of the system. The probability that the energy state of a particle changes from e_i and e_j increases with decreasing difference $\Delta e = e_i - e_j$ and decreases with decreasing temperature T. The formula for this transition propability $P(e_i, e_j, T)$ is given by

$$P(e_i, e_j, T) = e^{(e_i, e_j)/k_b T}$$

where k_b is the so called Boltzmanns constant which defines the dependence of the temperature drop on the transition probability.

2.1.3 Tabu Search

Fred Glover and Eric Taillard specify in [Glover and Taillard(1993)] that Tabu Search can be viewed as an iterative technique which explores a set of problem solutions, denoted X by repeatedly making moves from one solution s to another solution s' located in the neighbourhood $N(s)$ of s. Moves are selected according to a certain criteria in order to find an optimal solution more quickly. Starting from the best solution s_{best}, the process is repeated, whereby previously executed moves are disalowed (tabu). This prevents cyclic behaviour and not getting stuck in suboptimal regions to which the creteria may lead.

2.2 Which One To Choose

Steven S. Skienna states in [Skiena(2008)], that he has never encountered any problem where genetic algorithms seemed to be the right way to attack these problems and he recommends to stick to Simulated Annealing or Tabu Search instead. The study [Francis Gorman(1998)], on the other hand, shows that genetic algorithms enhanced with tabu search can be used effectively specifically for railway optimization problems. The tabu restrictions prevents cyclic behaviour by searching different directions when a state is revisited. Thus, a tabu-enhanced genetic search algorithm, adapted to the task, can be suitable for solving the optimization problem.

3 Models

This section describes the models and its designations used in the further course of this paper. The input model, denoted Θ, holds initial information about all entities. The output model, is referred to as timetable, denoted Φ. Φ is given by a set of states φ for each point in time. A state φ holds information about the capacities of each station C^s, line C^l and train C^r and the location of all trains L^r and passengers L^p.

A list of the indices used is given below. These may change depending on need or collision.

Indices

t : index for a point in time
s : index for a station
l : index for a line
r : index for a train
p : index for a passenger

3.1 Timetable Evaluation

A timetable Φ is evaluated by a fitness function $F(\Phi)$ which sums the total delay for all passengers. This function is given by

$$F(\Phi) = \sum_{p=0}^{\rho} max(0, d_p) + \sum_{p=0}^{\varrho} t_{max}$$

where ρ is the number of arrived passengers, ϱ is the number of travelling passengers, d_p is the delay of the passenger p and t_{max} is the latest possible arrival time.

3.2 Moves

Furthermore, moves $m \in M_r^{\varphi}$ where M_r^{φ} is a set of all legal moves, can be executed for the train r at state φ. The following moves are possible:

- $m^{board} \equiv (t, p, s)$ Boarding passenger p to the train t from station s
- $m^{detrain} \equiv (t, p, s)$ Detrain passenger p from the train t to station s
- $m^{depart} \equiv (t, f, d, l)$ Detrain train t from station f to station d via line l
- $m^{start} \equiv (t, s)$ Start train t on station s
- $m^0 \equiv ()$ Null move - nothing happens

4 Algorithm

The algorithms that are used to find an optimal feasable solution are described below. The algorithms form a tabu-enhanced genetic search which has been adapted to the specific needs.

Starting from time $t = 0$, a neighbourhood $N(\varphi)$ is scanned for the state φ_t. This is done by evaluating all legal moves M_r for each train r. The best move m_{best} is picked for each train and pushed to φ. When the best move for each train has been found, t is increased by 1 and the neighborhood is scanned for the next state. This procedure is repeated until any of the following events occur:

- $t \geq t_{max}$
- φ_t is illegal
- all passengers have arrived

Then the genetic search concept of selection is applied by remembering Φ when it is better than all previously visited timetables. Starting from a randomly selected state $\varphi \in \Phi_{best}$ the neighourhood is scanned again. Executed moves are remembered in a Tabulist T and are excluded from all neighbourhoods in the following iterations.

The pseudocodes for search 1 and neighbour 2 summarize the procedure.

Algorithm 1 search

```
Create initial Solution $\Phi$
Set best solution $\Phi_{best} = \Phi$
Set initial state $\varphi$
Set start time $t = 0$
while $F(\Phi_{best}) \neq 0$ do
    while $t \leq t_{max}$ do
        Find the best neighbour $\varphi = neighbour(\varphi)$
        Push neighbour to solution $\Phi.push(\varphi)$
        Set the next null state $\varphi = \varphi.next\_null()$
        if $\varphi.is\_illegal()$ or All passengers arrived then
            break
    if $F(\Phi) < F(\Phi_{best})$ then
        $\Phi_{best} = \Phi$
    else
        $\Phi = \Phi_{best}$
    $t = random\_number\_between(0, t)$
    $\varphi = \Phi_{best}.at(t).clone()$
    $\Phi.drain(t..)$
return $\Phi_{best}$
```

4.1 Evaluating Moves

An important process of the algorithm is the evaluation of moves. When moves that lead to a bad result are excluded early on, a desirable result can be found much faster.

Algorithm 2 neighbour

```
for $r \in \Theta.train\_ids$ do
    Initalize best move $m_{best}$
    Get list of legal moves $M_r = legal\_moves(\varphi, r)$
    for $m \in M_r$ do
        if $m > m_{best}$ and $m \notin T$ then
            $m_{best} = m$
    if $m_{best}$ then
        $\varphi.push(m_{best})$
        $T.push(m_{best})$
return $\varphi$
```

A common evaluation practise is, to choose an objective function - mostly reffered to as "cost" or "fitness" function - which for example, maps any tuple of legal moves m and possible states φ to $x \in \mathbb{R}$:

$$c : (\varphi, m) \to \mathbb{R}$$

Another abstraction is to use a set of rules R to define the order of moves. We ended up choosing this method as it made surveying the evaluation much easier. Pseudocode 3 shows how the comparison is defined:

Algorithm 3 is_greater

```
for $r \in R$ do
    Get result $\psi = r(a, b, \varphi)$
    if $r \neq None$ then return $\psi$
return $false$
```

A rule r is a function $r(a, b, \varphi)$ given by

$$r(a,b,\varphi) = \begin{cases} true, & \text{if } a \text{ should be prefered to } b \\ false, & \text{if } b \text{ should be prefered to } a \\ None, & \text{otherwise} \end{cases}$$

where a and b are the compared moves.

To simplify the abstraction, rules are divided into categories, each of which fills a single purpose. The following sets in the given order are optimized for finding an optimal solution for the *large* data set given by the GI.

4.1.1 Avoid Station Overload

The purpose of the rules in this set is to prevent too many trains from arriving at stations at the same time, thus overloading the station.

The first rule compares a departure a^{depart} with the null move b^0 and is given by

$$r(a^{depart}, b^0, \varphi) = \begin{cases} false, & \text{if } (c_{est} - 1) \leq -c_{max} \\ None, & \text{otherwise} \end{cases}$$

where c_{est} is the estimated station capacity of the arrival station of a^{depart}, i.e. the capacity of destination at the arrival time of the departing train and c_{max} is the maximum capacity of the destination. This avoids trains from departing towards overloaded stations.

The second rule compares a train start a^{start} and the null move b^0 and is given by

$$r(a^{start}, b^0, \varphi) = \begin{cases} false, & \text{if } c_{est} \leq 0 \\ None, & \text{otherwise} \end{cases}$$

This avoids station overloading caused by starting a train at a station to which other trains are en route.

4.1.2 Detrain Arrived Passengers

The purpose of the single rule in this set is to detrain passengers from trains that have arrived in the destination station of the passenger.

The rule compares a detrain move $a^{detrain}$ to any of $b \in \{m^{board}, m^{depart}, m^0\}$ and is given by

$$r(a^{detrain}, b, \varphi) = \begin{cases} true, & \text{if } s = dest_p \\ None, & \text{otherwise} \end{cases}$$

where s is the station to which the passenger p should be detrained and $dest_p$ is the destination station of p.

4.1.3 Board By Arrival

The purpose of the single rule in this set is to preferentially board passengers with a short arrival time.

The rule compares boarding a^{board} to boarding b^{board} and is given by

$$r(a^{board}, b^{start}, \varphi) = \begin{cases} true, & \text{if } arrival_a < arrival_b \\ false, & \text{otherwise} \end{cases}$$

4.1.4 Board By Travel Path

The purpose of the single rule in this set is to board passengers together with other passengers that travel the same path.

The rule compares boarding a^{board} to any of $b \in \{m^{depart}, m^0\}$ and is given by

$$r(a^{board}, b, \varphi) = \begin{cases} true, & \text{if } \exists (t, s, q) \in \{(t, s, q)_a\} \exists p \in P_t(P(p) \subseteq P(q) \vee P(q) \subseteq P(p)) \\ None, & \text{otherwise} \end{cases}$$

where t is the train, s the destination station and q the passenger to be baorded, P_t is the set of passengers travelling in t and $P(p)$ is the shortest path from the current location to the destination of p.

4.1.5 Board To Empty Train

The purpose of the single rule in this set is to board passengers on trains that are empty.

The rule compares boarding a^{board} to any of $b \in \{m^{depart}, m^0\}$ and is given by

$$r(a^{board}, b, \varphi) = \begin{cases} true, & \text{if } c^t_\varphi = c^t_{max} \\ None, & \text{otherwise} \end{cases}$$

where c^t is the capacity of the train t at state φ, that should be boarded to and c^t_{max} is the maximum capacity of the train.

4.1.6 Depart To Exact Destination

The purpose of the single rule in this set is to prefer departures that bring the train to the destination station of any of its boarded passengers.

The rule compares one departure a^{depart} to another b^{depart} and is given by

$$r(a^{depart}, b^{depart}, \varphi) = \begin{cases} true, & \text{if } \forall(t, f, d, l) \in \{a, b\} \exists p \in P_t(D(d, d_p) = 0) \\ None, & \text{otherwise} \end{cases}$$

where t is the train and s the destination station of the corresponding departure. I.e. For all moves $m \in \{a, b\}$, there exists one passenger p in train t so that the distance between destination d of the departure and destination d_p of passenger p equals 0.

4.1.7 Depart Towards Destination

The purpose of the single rule in this set is to prefer departures that bring the boarded passengers of a train closer towards their corresponding destination.

The rule compares one departure a^{depart} to another b^{depart}, and is given by

$$r(a^{depart}, b^{depart}, \varphi) = \begin{cases} true, & \text{if } D(d_a, p_0) < D(d_b, p_0) \\ false, & \text{otherwise} \end{cases}$$

where d_m is the destination of the corresponding move m and p_0 is the first passenger in the departed train.

4.1.8 Depart Passenger Train

The purpose of the single rule in this set is to depart non empty trains.

The rule compares a departure a^{depart} to the null move b^0 and is given by

$$r(a^{depart}, b^0, \varphi) = \begin{cases} true, & \text{if } \exists(t, f, d, l) \in \{(t, f, d, l)_a, ()_b\}(c^t_\varphi < c^t_{max}) \\ false, & \text{otherwise} \end{cases}$$

where c^t_φ is the capacity of train t at state φ and c^t_{max} is the maximum capactiy of t.

4.1.9 Depart To Pickup Passenger

The purpose of the single rule in this set is to depart trains towards passengers that have not arrived at their destination station so far.

The rule compares a departure a^{depart} with the null move b^0 and is given by

$$r(a^{depart}, b^0, \varphi) = \begin{cases} false, & \text{if } \exists (t, f, d, l) \in \{(t, f, d, l)_a, ()_b\}(P_t = \varnothing \vee P_f \neq \varnothing) \\ None, & \text{otherwise} \end{cases}$$

4.1.10 Choose Train Start

The purpose of the rules in this set is to start trains at stations with passengers. Passengers with low arrival times are prefered.

The first rule compares a train start a^{start} to the null move b^0 and is given by

$$r(a^{start}, b^0, \varphi) = \begin{cases} true, & \text{if } \exists (t, s) \in \{(t, s)_a, ()_b\} \exists p \in P_s(|p| \leq c_t) \\ None, & \text{otherwise} \end{cases}$$

where $|p|$ is the size of the passenger group p. This leads to trains starting at stations with passengers that can be boarded on train t.

The second rule compares one train start a^{start} to another b^{start} and is given by

$$r(a^{start}, b^{start}, \varphi) = \begin{cases} true, & \text{if } min(arrival_{Pa}) < min(arrival_{Pb}) \\ false, & \text{otherwise} \end{cases}$$

5 Implementation

The theory of an algorithm alone is not sufficient to find solutions in a desirable time. The implementation must be carried out with the help of suitable data structures that make it possible to quickly change and access information. This section describes the programming language chosen, design decisions and the methods used to improve performance.

5.1 Rust

The implementation was done in the multi-paradigm programming language *Rust*[5]. It is designed for performance and safe concurrency. *Rust* uses borrow checkers to validate references and thus gurantee memory safety. Due to the high performance it is well suited for optimization problems. The well written documentation, the rich ecosystem and the integrated package manager *Cargo*[6] make it possible to work productively with it.

5.2 Architecture

The structure of the implementation follows the layout conventions[7] specified for *Cargo* projects. Except for tests, which are located at the point of implementation of an abstraction. This design decision was made because separating components that involve the same abstractions can increase the complexity of a program. Furthermore, abstractions are located following the single-responsibility principle in the project root. Rules and the definitions of their order are located in the *rules* submodule.

5.3 Data Structures

Preparing data through appropriate data structures enables significant performance improvements in finding optimal timetables. The following data structures proved to work well for the corresponding use cases.

- *vector*: A contiguous growable array with an access and update complexity of $O(1)$. It is suitable for holding and adjusting all data of a state φ.
- *hashset*: A hashmap with no value. The complexity $O(k)$, where k is the capacity of the set, allows the quick addition and removal of data. Thus, for a state, there are two vectors of hashesets that make it possible to quickly access and update the passengers in a train t or on a station s.

[5]Rust `https://rust-lang.org/`

[6]Cargo `https://doc.rust-lang.org/cargo/guide`

[7]Cargo project layout `https://doc.rust-lang.org/cargo/guide/project-layout.html`

- *linked-hashset*: A hashset that preverses the insertion order and allows popping the first inserted element in $O(1)$ time. It is used to remove the first state from the tabu list when the maxmium number of states in the set is reached.
- *fxhash* [8]: The fxhash has its strength in hashing 8bytes on 64-bit platforms and is significantly faster than the standard sip hash which is desgined to withstand ”hash flooding” DoS attacks. fxhash is used as a hasher for all hashmaps and hashsets.
- *decimal*: For precise calculation with real numbers. It is used to represent the speed of trains and distances between stations.

5.4 Performance

The precalculation of data allows quick access later on. This saved time by initially finding the shortest path and distance between all stations using the djikstra algorithm. This information is accessed by the rules 4, 6 and 7.

Furthermore, time could be saved when hashing a state by reducing the attributes used for hashing. The location of all trains L^r and passengers L^p, the time t and the pushed moves M are sufficient to uniquely identify a state, since all other attributes can be derived from it. Breaking down the attributes used for hashing a state gave a significant performance improvement on large datasets, as it made adding and accessing the tabu list less costly.

In addition, benchmarks showed that cloning a state takes a lot of time, especially for large models. Thus, performance could be improved by undoing changes to an instance of a state instead of remembering a duplicate of its old version.

6 Results

Table 1 shows the results of the algorithm and the implementation in Rust. Two datasets given by the GI are compared, one small named ”unused-wildcard-train” and a ”large” dataset.

[8]fxhash `https://docs.rs/fxhash/latest/fxhash/`

The tests were performed in a Docker container given 2gb memory and 2 cpus. This corresponds to the given competition circumstances.

If the search does not improve for 25.000 iterations or does not find the optimum within 3 seconds it is discontinued and considered a failure. An iteration is defined as a complete search of a neighborhood.

Table 1: Model Results

	unused-wildcard-train	large
Model:		
Stations	5	215
Lines	8	18426
Trains	10	37
Passengers	41	721
Compared moves:		
Average	257.089	900.465
Maximum	1.293.599	1.362.345
Minimum	1.578	723.708
Search time:		
Average	665ms	264ms
Maximum	3.001ms	450ms
Minimum	2ms	234ms
Compared moves / millisecond:		
Average	458	3.256
Maximum	780	3.427
Minimum	417	2.997
Optimum reached:	95%	100 %

7 Conclusion

The evaluation of moves through carefully crafted rules combined with tabu-enhanced genetic search can be used to precisely generate desired timetables.

The results show that despite the size of the *large* set, results are found relatively fast compared to the $unused - wildcard - train$ set. On the one hand, this shows that desirable solution times can easily be reached for large problem sizes, but on the other hand, that the rule set is only optimized for a series of inputs similar to the *large* set. Thus, rules can be found that apply better for a largerer range of data sets.

Furthermore, it is noticeable that "transit" stations where passengers transfer from one train to another are not necessary in order to find optimal solutions. This may be different for other data sets. For that case, other rules for transfering passengers can be easily added in order to adapt this behavior for the timetable as well.

The tabu-enhanced genetic search algorithm thus has the potential to find desirable solutions not only for the informaticup but also for the Deutsche Bahn's "Taktfahlplan".

References

[Francis Gorman(1998)] Michael Francis Gorman. An application of genetic and tabu searches to the freight railroad operating plan problem. *Annals of Operations Research*, 78(0):51–69, Jan 1998. ISSN 1572-9338. doi: 10.1023/A:1018906301828. URL `https://doi.org/10.1023/A:1018906301828`.

[Glover and Taillard(1993)] Fred Glover and Eric Taillard. A user's guide to tabu search. *Annals of Operations Research*, 41(1):1–28, Mar 1993. ISSN 1572-9338. doi: 10.1007/BF02078647. URL `https://doi.org/10.1007/BF02078647`.

[Holland(1975)] John H Holland. *Adaptation in Natural and Artificial Systems.* MIT press, 1975.

[Skiena(2008)] Steven S. Skiena. *The Algorithm Design Manual.* Springer, London, 2008. ISBN 9781848000704 1848000707 9781848000698 1848000693. doi: 10.1007/978-1-84800-070-4.