Solving the Operational Crew Rescheduling Problem

Miguel Alexandre Cerejo Frazão
miguel.cerejo.frazao@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

Abstract

This master's thesis presents the Operational Crew Rescheduling Problem (OCRSP), a problem of railways management that deals with the quick rearranging of railway crew schedules that were disrupted by an unforeseen event and proposes and implements a Reinforcement Learning algorithm and a Local Search algorithm to solve

Keywords: Operations Research, Crew Rescheduling, Reinforcement Learning, Local Search

1. Introduction

This thesis presents the Operational Crew Rescheduling Problem (OCRSP) and presents two solutions to it: one based on Reinforcement Learning and the other on Local Search.

Let's start by motivating the problem. Railways are a seemingly simple transportation mode: trains run trips on dedicated tracks to transport passengers or freight from one location to another, starting and ending at specific times. However, if a Train Operating Company (TOC) has to provide several hundred trips per day, considering overall client satisfaction and minimizing used resources, the problem of planning becomes incredibly intricate.

To control complexity, TOC's divide the planning task into different phases (long-term planning, short-term planning, dispatching and control) and types of resources (track and time, rolling stock, and crew), resulting in a set of smaller, and therefore less complex, dependent sub problems [4].

Some months prior to the actual day of operation, the TOC does long-term planning, a process to construct the timetable (the list of trips) for a generic week and to assign abstract resources (rolling stock and crew members) needed to perform all trips in the timetable.

The resulting plans must not only make efficient use of resources and oer supply to the demands of railway users: they should also be robust and recoverable, since railway operations are exposed to unforeseen events that disrupt the planned schedule. The schedule's robustness permits small perturbations to occur, without the need to modify the schedule. The recoverability allows the schedule, to be easily rearranged after a disruption.

A few months to a few days before the day of operation, the TOC does short-term planning. In this process, concrete resources are assigned to trips and the schedule is rearranged, if needed, due to some predicted irregularity, like an anticipated infrastructure maintenance or a football match.

During the day of operation, unpredictable events, such as problems with the infrastructure, rolling stock, crew, weather conditions and accidents, can disrupt the planned schedule, causing the planned schedule to be inoperable.

These disruptions, due to the strong interdependencies in the railway network, shared resources and to the cost efficient resource schedules, if not solved quickly, can propagate through space and time over the railway network causing some train trips to be delayed or even canceled, causing frustration on clients [5]. This scenario may have serious financial and reputation consequences for the TOC.

The process of rearranging the disrupted schedule, with the goal of obtaining a new feasible one, is called rescheduling. The rescheduling process, like the planning process, is divided into three separate smaller processes, completed in this order: firstly, rearrange the timetable, then reschedule rolling stock, and then, reschedule crew members. Iterations between the three steps may be necessary when it is not possible to assign resources to some of the trips from the proposed timetable.

In rescheduling, it is of most importance to minimize canceled trips, to satisfy clients. Deviations from the original planned schedule should also be minimized. A TOC that effectively and efficiently handles disruptions, benefits in terms of its public image and overall costs.

In this document, we focus on the last phase of

the rescheduling process, that is, rescheduling of crew members' tasks. From now on, we will call this problem, as in the literature, *Operational Crew Rescheduling Problem* (OCRSP).

2. Background

The *Operational Crew Rescheduling Problem* (OCRSP) deals with assigning and reassigning tasks to *crew members* after the timetable has been changed and the rolling stock has been rescheduled. A crew member can either be a train driver or a train guard. We abstract from this so, from now on, we will just refer to crew members.

In order to operate the timetable, trains are split into trips between *relief points*. A relief point is a station where a crew member can switch from one rolling stock unit to another.

A *task* is the unit of work of a crew member: a task for a driver corresponds to driving a certain trip, from a certain start location and start time to a certain end location and end time; a task for a guard is to inspect tickets in a certain trip. There are also *positioning trips* where the crew member travels in the train as a passenger or uses other means of transportation, such as taxis, to reach a location where he can either take a break, sign out and go home or perform productive work.

A *duty* is a sequence of tasks planned for a crew member, usually, for a single day. Each duty must end at the *crew base* where it started[1]. The set of crew bases is a subset of the set of relief points.

Depending on the TOC, several other constraints must be verified when checking the feasibility of duties, such as the existence of a meal break at an appropriate time and location, maximum duty duration, minimum time to transfer between rolling stock units and crew member appropriate skills for the tasks. A duty is said to be *feasible* if it verifies all constraints.

In this context, a *crew schedule* is the set of all duties for a calendar day. A crew schedule is said to be *feasible* if all the duties in it are feasible duties.

On the day of operation, the crew schedule is initially given by the *original duties*, that is, the duties conceived in the short-term planning phase, each assigned to a crew member. The original duties are either *active duties*, meaning that they are a sequence of tasks to be performed by the corresponding crew member, or *reserve duties*, meaning that the crew member is on standby at a crew base for a given time period, for handling the case that there are tasks that cannot be completed by the active duties due to a disruption.

The original crew schedule, the one constructed in the planning phase, is a feasible crew schedule and covers all tasks that were planned.

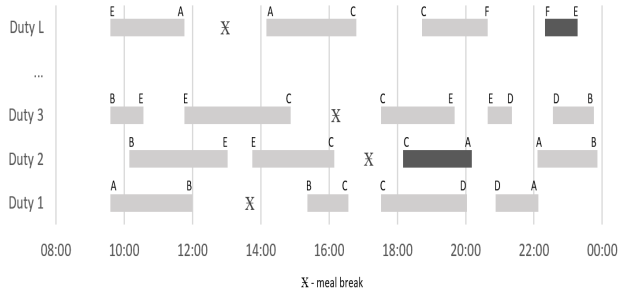

Figure 1: Example of an original crew schedule for a given day, with L feasible duties.

In Fig. 1, an original crew schedule is presented. Each grey block is a task. The two dark grey tasks represent positioning trips. Letters over the tasks indicate the departure/arrival station of the task.

After the occurrence of a disruption, dispatchers quickly proceed to rearrange the timetable and reschedule the rolling stock.

Due to the rearranged timetable and rescheduled rolling stock, some crew duties may become infeasible. Duties can be made feasible by removing specific tasks from them and, in some cases, also by adding positioning trips. However, this creates *unplanned tasks*, tasks that are not planned in any duty, which must be quickly covered (assigned a crew member), otherwise, the corresponding trips will be cancelled.

So, in this work we assume that the first step performed after a crew schedule becomes disrupted is an initial repair, which results in a set of feasible duties, which we call *initially repaired crew schedule*, and unplanned tasks.

In such a manner, the goal is to rapidly get a new feasible crew schedule, that covers as many tasks as possible.

This is achieved by inserting the unplanned tasks in the duties. However, planning a task in a duty might make the duty infeasible. Given this, it is also permitted to remove tasks from duties, which then also become unplanned tasks.

Often, considering only the directly affected duties is not enough to produce a feasible crew schedule that covers all tasks, after a disruption. It is usual to also consider a neighbourhood of the set of disrupted duties, that is, a set of duties that were not directly affected by the disruption but might be helpful to solve it by being available to cover tasks. For details about the construction of a neighbourhood, we refer to (D. Potthoff, 2010, pp 31-41). These duties bring more possibilities to plan an un-

[1]There exists the case of composing two duties, and, in between a rest period to sleep. Usually, those duties do not start and end at the same crew base

planned task.

Therefore, an *instance* of the OCRSP, consists in a set of feasible duties (the initially repaired crew schedule) and a set of unplanned tasks where the primary goal is to plan the maximum number of unplanned task in those duties, getting a new feasible crew schedule.

Let an instance of the OCRSP have N duties and M unplanned tasks. N is generally much smaller than the total number of duties in a crew schedule, in such a manner that computation times are small enough to be acceptable for a real-time approach. The primary goal is to plan the maximum number of unplanned tasks in the duties. The tasks that are being executed, or have already finished at the time of rescheduling, cannot be removed from the duties.

So, a *solution* of an instance of the OCRSP is a feasible crew schedule and a set of zero or more unplanned tasks. A *solution with full coverage* is a solution with no unplanned tasks.

Note that, given an instance of the OCRSP, we immediately have a solution: the given initially repaired crew schedule and the given unplanned tasks. However, this solution, generally, is much worse than most solutions.

In a solution, an unplanned task is extremely undesirable, such that, a solution with more tasks covered is always better then a solution with less tasks covered.

Changing a duty requires the crew member performing that duty to be informed, which may not happen due to some communication problem. So, changing a duty is also undesirable, but not as much as leaving a task unplanned.

When planning a task in a duty, the changed duty might contain overtime, changed breaks or the need for a paid transport for positioning purposes (e.g. taxi). This costs are also considered in the solution, but are much lower.

3. Related work

3.1. Dealing with the OCRSP

In the railway domain, crew rescheduling has been dealt with at least since 2005. Walker et al., in [10], were the rst who investigated this issue. The paper presents a model for an integrated approach to timetable rearrangement and crew rescheduling. Still, they considered only small instances.

A group of papers which based their approaches on set covering models are [2], [6], [5] and [7].

To be able to solve smaller disruptions, [9] developed an heuristic to solve crew rescheduling after small disruptions. This work is of special importance because some of the ideas of our solution were inspired by it. In this paper, the problem is solved as an depth-rst iterative-deepening (DFID) search in a tree [3]. This tree has, as nodes, solutions of the OCRSP, that is, a feasible schedule, plus zero or more unplanned tasks.

The heuristic works fast, delivers good and desirable solutions and outperforms [5], when it comes to OCRSP in small disruptions. The algorithm is also integrated in CREWS.

3.2. Reinforcement Learning on similar problems

RL has already been used to solve problems similar to the OCRSP. To solve real-time train rescheduling, [12] use RL. The problem as some clear parallels with the the OCRSP. The research utilizes Q-learning [8]. Some ideas of our solution approach were inspired by this paper. The RL approach is tested with real world case and the results are satisfactory compared to some basic algorithms, such as FIFO (First In First Out) and Random Walk. Still, the model has some limitations that prevent the algorithm to be used in real scenarios. The paper serves as proof-of-concept for future researchers.

In the airline industry, to deal with fleet management after a disruption, [1] apply RL. The approach deals with a disruption by swapping flights between the fleet. This research also uses Q-Learning. The results manifest some interesting features, namely the ability to recover from heavily disrupted traffic. This paper suggests future potential lines of research on applying RL.

As an innovative approach the scheduling of NASA space shuttle payload processing, [11] employ RL. In this paper, the agent uses TD(λ)-learning [8]. The results suggest that reinforcement leaning can provide a new method for constructing high-performance scheduling systems.

In the next section, we give a brief introduction to Machine Learning, present some Reinforcement Learning successes and formalize Reinforcement Learning.

4. Reinforcement learning

4.1. Formalizing Reinforcement Learning

The RL architecture consists of an *agent* and an *environment* which interact through a sequence of discrete timesteps, where at each timestep, the agent receives some representation of the environment's *state*, and, on that information, selects an *action*. One timestep later, in part as a consequence of its action, the agent receives a *reward*, and transits to a new state.

Thereby, the agent and the environment give rise to a sequence of states, actions, and rewards, which we will call an *episode*. The agent learns doing actions and receiving the rewards from the environment. This process is repeated throughout multiple episodes.

We use the following notation:

- $t \in \mathbb{N}_0$ represents the current time in a given

episode. Each episode starts with $t = 0$.

- S_t represents the *state* of the environment at time t.
- A_t represents the *action* taken by the agent at time t.
- $R_{t+1} \in \mathbb{R}$ is the reward received after performing action A_t in state S_t.
- $\mathcal{S}$ represents the set of environment's states.
- $\mathcal{A}$ is the set of all possible actions. $\mathcal{A}(S_t)$ is the set of actions the agent can perform when the environment is in state S_t. For every t, $\mathcal{A}(S_t) \subset \mathcal{A}$.
- $E_i = \{S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...\}$ is the i-th episode.

The goal of the agent, at each time t, is to maximize the *return*, G_t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate* and, R_{t+k+1}, is the reward at time $t+k+1$. The discount rate determines the present value of future rewards.

The agent's *policy* π is a map, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which gives the probability of taking a specific action, when in a specific state. A policy denes the learning agent's way of behaving at a given time.

With the goal of finding the *optimal policy*, that is, the policy that gives the agent the best expected return, it's useful to define the *value function* Q $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, that maps *state-action pairs* to the expected return starting in that state, doing that action, and following a policy π that is:

$$Q(S_t, A_t) = \mathbb{E}_\pi[G_t | \textit{Current-State} = S_t, \textit{Performed-action} = A_t] \quad (1)$$

The *Q-values* are initialized to arbitrary values.

After each action in each episode, the Q-values are updated according to a rule which varies conform the RL algorithm.

We will use the Q-Learning algorithm, whose update rule is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma \max_A Q(S_{t+1}, A) - Q(S_t, A_t)\big) \quad (2)$$

where α is the *learning rate*, $0 \leq \alpha \leq 1$.

Usually, to learn accurate Q-values, the agent does thousands of episodes.

If the policy followed by the agent is the *greedy policy*, the action chosen by the agent at each time step is the action with highest Q-value. If there is a tie, the agent chooses randomly between the actions with the highest Q-value.

To learn the value of the state-action pairs, the agent must explore the *state-action space*. However, the greedy policy does not allow much exploration of the state-action space. This problem is known as the *exploration-exploitation trade-off*. We solve this issue by letting the agent use an ϵ-greedy strategy, that is, at each time, with probability 1-ϵ, the agent chooses the action with highest Q-value (exploits), and, with probability ϵ, the agent chooses a random action (explores).

Usually, ϵ starts high and decreases over the learning, to favor exploitation of the learned policy in the final episodes.

Given all this, to characterize an RL problem, we need to define what is the agent and its actions, the environment and its states, the rewards and episodes. In the next section, we describe all this constituents of our RL approach to the OCRSP

5. Proposed Approach

5.1. Solutions and costs

In this section, we give notation about solutions and introduce the concept of cost function to evaluate solutions. Let:

- D_t: Set of duties at time t. D_t is the crew schedule at time t. D_t has always the same number of duties, N. Note that all duties in D_t are feasible.
- T: Set of tasks which have not started, at the time of rescheduling, i.e., the tasks that can be planned/unplanned. At each time step t, we have the following partition of T:
  - P_t is the set of planned tasks at time t: the tasks in duties of D_t.
  - U_t is the set of unplanned tasks at time t. It can be the empty set.

Each solution is a pair (D_t, U_t), i.e., a feasible crew schedule plus zero or more unplanned tasks.

To chose between solutions, there is a cost function c, that, given a solution, outputs a real number representing the quality of the solution: the higher the cost, the worse the solution. The cost of a solution is the sum of the costs of the duties plus the cost of the unplanned tasks[2].

The solution cost function c is such that

$$c(D_t, U_t) = \sum_{d \in D_t} cost_{duty}(d) + \sum_{u \in U_t} cost_{unplanned-task}(u) \quad (3)$$

[2]These functions are already provided by the CREWS product.

Given an instance, every duty already has a cost. Throughout the resolution, the duties are modified and this changes the cost of the duty. For instance, the act of changing the duty at least once, adds in the cost of the duty; other costs arise because of overtime, changed breaks or need for taxis.

The cost of an unplanned task is much larger than any other cost. For instance, adapting the costs from [9], we get
$cost_{unplanned-task}(u) = 1.000.000$, while changing a duty d adds 300 to $cost_{duty}(d)$. All the other costs are smaller.

Next, we specify the basic components of our RL approach to the OCRSP: the agent and environment, the states, the actions, the rewards and the episodes.

5.2. Agent and Environment

Our RL agent will behave as a TOC dispatcher. Given an instance of the OCRSP, the agents goal is to plan the maximum number of unplanned tasks, that is, to find a good solution.

The environment, which the agent interacts with, has information about the duties and the unplanned tasks.

5.3. States

Each *state* of the environment represents a solution: $S_t = (D_t, U_t)$, where D_t is the set of duties at time t and U_t is the set of unplanned tasks at time t.

The agent's purpose is to get to a state $S_t = (D_t, U_t)$ where $U_t = \emptyset$ (the empty set), explicitly, to a state with 0 unplanned tasks.

Sometimes, this might not be possible. In this case, the agent stops after a certain number of time steps, hopefully, with a state that represents a better solution, that is, a state with lower cost (for the sake of simplicity, since each state determines unequivocally a solution, we define the cost of a state as the cost of the solution associated).

Given a state S_t, the agent has the core information to decide on an action A_t and get to state S_{t+1}. This construction mimics the practice of human dispatchers, who rely on the information concerning duties and unplanned tasks to determine their decisions.

The initial state $S_0 = (D_0, U_0)$ represents the solution automatically given by the OCRSP instance. It contains the original duties with some unplanned tasks, that is, a solution without full coverage (every OCRSP instance has unplanned tasks to solve; we never try to improve a solution with full coverage).

Observe that we can also see our approach as a OCRSP solution improvement heuristic: the initial state can be a solution found by other approaches of the OCRSP, that needs to be improved. Everything works the same way, only the origin of the initial state is different (raw solution after a disruption vs solution found in other approach), but that is irrelevant for the computation.

The tasks that are being performed on the current time cannot be manipulated, such as the past tasks. So, we number only the tasks that can be planned/unplanned by the agent. The current time does not change in the computation, as the computation is supposed to last some seconds. Therefore, the set of tasks able to be planned/unplanned is static in a given instance of the problem.

5.4. Actions

An action by the agent is to try plan an unplanned task in a duty. So, if the environment is in state $S_t = (D_t, U_t)$, the set of actions the agent can perform is $\mathcal{A}(S_t) = D_t \times U_t$.

Hence, for each t, $A_t = (d, u)$, where $d \in D_t$ is a duty and $u \in U_t$ is the unplanned task which the agent will try to insert in d.

Some actions cannot be executed due to the fact that some tasks cannot be planned in certain duties. This can happen for a variety of reasons (e.g., the train driver doesn't have the skills). This actions are *impossible actions*. Verhaegh and co-authors define the *Basic Feasibility Check* to quickly discard some of the duties where a given unplanned task cannot be planned [9]. For instance, if the driver associated with the duty does not have route or rolling stock knowledge to perform the task, the task cannot be planned in that driver's duty, so it fails the *Basic Feasibility Check*.

We will also use a *Basic Feasibility Check*, based on [9], to, at each step, get rid of most impossible actions. Note that some impossible actions may pass the *Basic Feasibility Check*.

When it is possible to do the action, it might be the case that the unplanned task overlaps some tasks already in the duty; if the minimum transfer time between two consecutive tasks is violated, this is considered an overlap as well.

In this case, we say that the action is a *conditionally possible action*. The tasks overlapped in the duty become unplanned tasks. Positioning trips do not have to be rescheduled and are simply canceled.

When an action can plan a task in a duty without removing tasks from the duty (there are no overlaps), we say this is an *unconditionally possible action*.

In summary, we divide the actions into three categories: impossible actions, conditionally possible actions and unconditionally possible actions.

An impossible action ensues when the action task cannot be planned in the action duty. This can happen for a variety of reasons, such as:

- The crew member associated to the duty does not have the necessary route or rolling stock

knowledge to execute the task;

- The minimum travel time between current location and start location of the unplanned task is more than the available time between current time and start time of the unplanned task;

Note that the environment has all this information about the tasks, duties and about the crew members to which these duties are assigned.

When an agent performs an *impossible action*, the state remains the same. Before each action, it is made the *Basic Feasibility Check*, to quickly discard most impossible actions.

A *conditionally possible action* arises when it is possible to plan the task in the duty, but it is necessary to create new unplanned tasks. This happens when the original unplanned tasks overlaps tasks or transfer times in the duty.

An *unconditionally possible action* is an action that can be planned without the need to create more unplanned tasks.

5.5. Rewards (RL only)

Given the previous architecture, we now need to define what is the environment's feedback to each action performed by the agent.

Since our goal is, given an instance of the OCRSP, to find a solution with the lowest cost, which corresponds to a solution with the fewest unplanned tasks, we need do design the rewards in such a way that, given those rewards, our agent is encouraged to find a good solution.

The model for rewards is the following: the reward is the difference between the costs of the current state S_t and the next state S_{t+1}, that is, after action A_t, the agent receives the reward $R_{t+1} = c(S_t) - c(S_{t+1})$.

The reward is positive if the cost of the next state is smaller then the cost of the current state and the reward is negative if the cost of the solution increases in that time step. This design leads the agent to follow states with decreasing cost, which is adequate, given the goal.

The reward is zero if the next state is the same as the current state (it's very improbable that two distinct states have the same cost), that is, the action taken was impossible.

If A_t is conditionally possible, then R_{t+1} can be either positive or negative, depending on the cost of S_{t+1}.

If A_t is unconditionally possible, then R_{t+1} is positive, because S_{t+1} has one less unplanned task, which is the component with most weight in the solution cost.

5.6. Episodes (RL only)

Given an instance of the problem, every episode starts in the state representing the first solution given.

The episode ends when the environment gets to a state that represents a solution with full coverage (without unplanned tasks) or, possibly, after a predetermined number of timesteps.

After each action in each episode, the agent updates the Q-values, according to the Q-Learning update rule. To achieve convergence of Q-values, it is usual to simulate thousands of episodes.

6. Implementation

6.1. Trying Reinforcement Learning

The implementation was made in C++ and used code previously built in SISCOG, such as classes and functions that represent useful concepts for the problem.

Firstly, we defined new classes, such as Agent, State, Action, Transition, Episode and Solver. The class that was most modified during software development was the Agent and the way it interacted with State and Action: firstly, when the Agent found a new State, it created all State' actions, that is, all the possible pairs (*template*, *uncovered task*), where the action is to try to plan *uncovered task* in the duty assigned to *template*, to then choose a random action among all actions. This approach didn't work because of the time and memory needed to create and store all the actions.

The next strategy was the following: when the Agent needed to find a random action in a State, it would do the following loop: within each iteration, the agent selects a random action, that is, a random template and a random uncovered task. Then, it checks if that action was previously verified as a possible action: if the action was already checked and is an impossible action, the agent will restart the loop; if the action was already checked and it's a possible action, the agent executes that action. If the action wasn't already checked, the agent will check if the action passes the *feasibility check*. If it passes the *feasibility check*, we then verify if it is a possible action. If it is, the agent does that action. This function, which we called *GetRandomPossibleAction(state)* will be thoroughly analyzed in the next section, since it's essential for the Local Search algorithm.

The *feasibility check* verifies quickly if an action is impossible by analyzing some proprieties of the template and the uncovered task, such as:

- The driver has the skills to do the task;
- The time interval of the task is in the time interval of the template.

The *feasibility check* is, on average, 10^4 times faster than the *possibility check*, which verifies if an action is possible. That's the reason the *feasibil-*

ity check comes first: it quickly discards impossible actions.

This new strategy radically improved the algorithm's performance, but the results were still much below the benchmark. Next, we implemented Watkins's Eligibility Traces on the tabular Q-Learning algorithm [8]. This improved the performance by 22%: however the results were still not enough. We tried various heuristics to solve the problem using Reinforcement Learning, but the results never improved significantly.

With hindsight, we could see that, the graph of states for big problems, was weakly connected, that is, the number of ways of going from state A to state B, were almost always 0 or 1. That way, Reinforcement Learning strategies are much slower to converge. The results were unsatisfactory for production and a new approach was needed.

6.2. Local Search

This realization led to some experimenting with new heuristics, which led to a big pivot: we got to the *Local Search* algorithm.

Local search reuses almost all the structure created for the RL algorithm, but the algorithm is not an RL algorithm. It has entities such as Agent, State, Action, but there is no learning. The results of this algorithm were satisfactory, even better than the benchmark on some instances of the problem.

As with RL, the agent starts in the initial state, the state that contains the initial solution. The variable T, which is the maximum number of tasks an agent can unplan in an action, is set to 0, that is, in the beginning, an action is possible if and only if, to plan the uncovered task in the duty associated with the template of the action, the agent doesn't need to unplan any task (an unconditionally possible action). The agent performs random possible actions, planning a task per action, without unplanning any task.

After some iterations, the number of possible actions gets small and the agent needs too much time to find a possible action in the current state (there's also the case that some states don't have possible actions for some T).

This time is a function of *aggressiveness*, a parameter of the *Local Search* algorithm: the higher the *aggressiveness*, the more actions the agent will check in the current state until it gives up.

If the *aggressiveness* is set to 1, the agent will check every combination of uncovered task and template to check if there is some possible action.

At this point, T is incremented to 1, that is, a possible action possible can uncover one task. Now, the current state has more possible actions, actions that need to uncover one task.

The number of uncovered tasks of two consecutive states can stay the same or decrease by 1, whereas, when $T = 0$, the number of uncovered tasks is always decreasing by 1 in each transition.

The agent iterates until it gets to a state where it can't find any additional actions. It halts here, outputting the solution of the last found state, the one with less uncovered tasks.

The pseudocode for *Local Search* is displayed in **Algorithm 1** .

The algorithm as 2 parameters:

- *aggressiveness*. This is a double between 0 and 1 which tells how aggressive the search is. This parameter determines the maximum number of actions an agent will try in a given state, until it finds a possible one. The role of this parameter will be more analyzed in a following.
- *maxTime*. This is the maximum number of seconds the algorithm will run until It halts and returns the best solution found.

On lines 3 and 4 of **Algorithm 1**, these parameters are set according to the user. For production, we usually set *maxTime* to 60, that is, the run will take 1 minute, and *aggressiveness* to 0.8.

On line 6, we initialize the variable *time*, which saves how much time has passed since the start of the execution.

On line 8, the variable T is set to 0. T is the maximum number of tasks the agent can uncover when performing an action. For example, if an action uncovers 3 tasks and $T = 2$, the action is not possible.

Algorithm 1 Local Search

```
1:  procedure :
2:
3:      set agressivness
4:      set maxTime
5:
6:      time ← 0
7:
8:      T ← 0
9:      state ← initialState
10:
11:
12:     while time < maxTime do
13:
14:         action ← GetRandomPossibleAction(state)
15:
16:         if action exists then
17:             newState ← Transtition(state, action)
18:             state ← newState
19:         else
20:             if T=1 then
21:                 return best solution found
22:             if T = 0 then
23:                 T ← 1
24:
25:         time ← GetCurrentTime()
26:
```

This variable starts on 0, that is, in the beginning an action is possible if it doesn't uncover any tasks. Through the course of a run, T will be increased to 1, when there are no more possible actions with $T = 0$.

On line 9, we initialize the variable *state* with the value *initialState*, which is constructed with the initial solution of the problem. The body of the algorithm consists in a *while* loop which stops when the run is going for more than *maxTime* seconds.

Every iteration of the algorithm consists in a state transaction. On line 13, the agent tries to find a possible action for the current state, and sets *action* to the result.

The function *GetRandomPossibleAction(state)*, which will be analyzed next, returns a random possible action or, if it doesn't find any, returns a null action.

On line 15, there's an *if-else* statement: if *action* exists (not null), then the agent will find *newState*, applying *action* on *state*. Then, *state* becomes *newState*, that is, *state* becomes the previous state when *action* is executed.

Otherwise, if *action* is null, that is, if the agent didn't find any possible actions on state, the algorithm follows: if $T = 1$, then the algorithm halts and returns the best solution found, that is, it compares all states found and outputs the best solution (note that every state corresponds to one and only one solution). If *action* is null and $T = 0$, this means that the agent can't find any possible actions in *state* that don't uncover tasks. To keep searching for better states, the algorithm allows the agent to remove 0 or 1 tasks per action. That is, $T \leftarrow 1$.

In line 24, the variable *time* is actualized and if $time < maxTime$, a new iteration starts.

The function *GetRandomPossibleAction(state)* was coded for the RL algorithm, but it is fundamental for *Local Search*. As explained, the function receives *state*, the current state of the run, and tries to find a possible action in that state, that is, an action that plans an uncovered task in a template, without unplanning more than T tasks. If a possible action is found, that action is returned. Otherwise, the agent returns a null action.

The pseudocode for *GetRandomPossibleAction(state)* is displayed on **Algorithm 2**.

The pseudocode for the algorithm for this function consists in the following:

In line 3, there's a while loop whose condition is "the number of actions already checked in the *state* is less than L". An action is already checked if the agent knows that the action is possible or impossible. If it's the first time visiting *state*, then the number of actions already checked is 0. With each iteration of the while loop, the agent checks a different random action.

Algorithm 2 GetRandomPossibleAction(*state*)

```
1:  procedure :
2:
3:      while state.numberOfActionsAlreadyChecked < L do
4:
5:          action ← GetRandomAction(state)
6:
7:          if action already checked and is not possible then
8:              restart loop
9:          if action already checked and is possible then
10:             return action
11:
12:         if action does not pass feasability check then
13:             add action to state.setOfActionsAlreadyCheckedAndNOTpossible
14:             restart loop
15:         else
16:             if action is possible then
17:                 add action to state.setOfActionsAlreadyCheckedAndPossible
18:                 return action
19:             else
20:                 add action to state.setOfActionsAlreadyCheckedAndNOTpossible
21:                 restart loop
22:     return null
```

L is an solving constant which depends on the size of the problem and the aggressiveness of the local search: larger problems have larger L's. Higher aggressiveness means higher L. L represents the maximum number of actions an agent will check in a state before giving up on *state*.

The total number of actions, possible or impossible, in a state is $numberOfTemplates \times state.numUncoveredTasks$, which is too large.

Using L, which is much smaller than the total number of actions in the state, the algorithm doesn't get stuck finding a possible action in a state. In the first states and with $T = 0$, about 10% of the actions are possible. This number decreases and can hit 0%. When T get incremented to 1, in an intermediate state, only about 4% of actions are possible. This number decreases again, as the solution gets better.

The first step of each iteration is in line 5. It consists in choosing a random action, *action*, which will be checked, that is, verified if it is possible or impossible. The function *GetRandomAction(State)* simply returns a random action, that is, a pair composed of a random template and a random uncovered task.

Every state has, as attributes, two sets of actions:

- *state.setOfActionsAlreadyCheckedAndPossible*; the actions that are already checked and are possible.
- *state.setOfActionsAlreadyCheckedAndNOTpossible*; the actions that are already checked and are impossible

If *action* is an already checked action and is not possible, that is, *action* is in

	Number of templates	Number of uncovered tasks
Instance 1	28	33
Instance 2	180	713
Instance 3	942	734
Instance 4	1143	1591

Table 1 - Description of the instances

	Instance 1	Instance 2	Instance 3	Instance 4
RL (4 min)	33/33 (1 seg)	56/713	21/734	11/1591
RL (60 min)	33/33 (1 seg)	378/713	294/734	105/1591
Local Search (1 min)	33/33 (1 seg)	449/713	322/734	320/1591
Local Search (4 min)	33/33 (1 seg)	459/713	360/734	423/1591
Local Search	33/33 (1 seg)	356/713 (7 seg)	322/734 (62 seg)	371/1591 (120 seg)
Thjis	33/33 (1 seg)	331/713 (7 seg)	349/734 (62 seg)	380/1591 (120 seg)

Table 2 - Number of solved tasks of the different algorithms on the 4 instances

state.setOfActionsAlreadyCheckedAndNOTpossible, the loop is restarted. If *action* is on the set of already checked and possible actions, the function returns *action*. If the iteration hits line 12, means that *action* wasn't already checked. The next step is to test if *action* is possible or impossible.

In line 12, we verify if the action passes the *feasibility test* (this test quickly discards impossible actions. However, some impossible actions pass the *feasibility check*). If the action doesn't pass the *feasibility check*, means *action* is an impossible action: *action* is added to the set of actions *state.setOfActionsAlreadyCheckedAndNOTpossible* and the loop is restarted.

If *action* passes the *feasibility test*, then we test if *action* is indeed possible (line 16). Testing if an action is possible is much more time consuming than the *feasibility check*: this is the reason the *feasibility check* is executed first, so it quickly rejects most impossible actions.

If *action* is possible, then we add action to *state.setOfActionsAlreadyCheckedAndPossible* and return *action*.

If *action* is not possible, the same code is executed as in lines 13 and 14: *action* is added to the set of impossible action in the state and the loop is returned.

If the agent checks more than L actions in a state, it gets out of the while loop and returns null.

In the next chapter, we show and discuss the results of the RL algorithm, the Local Search algorithm and *Thjis* algorithm [9].

7. Results

We tested the various algorithms on 4 instances of different sizes (see Table 1). The first instance is a small problem. In production, the instances are usually bigger, such as Instance 2, Instance 3 and Instance 4.

The runs were performed on an Intel Core i7-7700K 4.20 GHz with 32 GB of RAM.

We tested the RL algorithm with various times and parameters. As said before, the results were unsatisfactory for production. Either way, we experimented runs of 60 min of the RL algorithm to check if it converges to some good solution. Since changing the parameters of the RL algorithm barely affected the results, we opted to fix the parameters.[3]

We tested *Local Search* (with 1 minute of *maxTime*, 4 minutes of *maxTime* and with the same running time as *Thjis*) and *Thjis*. The next table presents the results for 6 algorithms:

1. RL running 4 minutes;
2. RL running 60 minutes;
3. *Local Search* running 1 minute;
4. *Local Search* running 4 minutes;
5. *Local Search* running the same time as *Thjis*;
6. *Thjis*.

The implementation of *Thjis* algorithm available tried to plan the most tasks and halt, being impossible to specify the time of the run.

Local Search has a variable user defined execution time.

Each algorithm, except *Thjis*, was ran 4 times in each instance, since they contain some randomness, albeit the variability of results isn't large. The number of tasks planned shown in Table 2 is the average of planned tasks in the 4 runs.

8. Discussion

We can see that the performance of RL (4 min) and RL (60 min) is much worse than *Local Search* and *Thjis*. As referenced before, the program must execute fast to be useful: if it takes much more than 4 minutes, it's not good enough for real world usage.

Instance 1 is a small instance that every algorithm can totally solve, that is, plan all unplanned tasks, in less than 1 second.

For the bigger problems, there are clear differences in the results obtained by the various algorithms.

The RL algorithm, even running for 60 minutes, is still worse than *Local Search* and *Thjis*, running less than 2 minutes.

[3] The RL algorithm used the following parameters for every run: $\alpha = 0.6$; $\gamma = 0.4$; $\epsilon_{initial} = 0.5$; $numberOfTransitionsPerEpisode = 500$

The results obtained by *Local Search* are at the same level as those obtained by SISCOG's implementation of *Thjis* algorithm. In Instance 2, *Local Search* (1 min) planned 449 out of 713 uncovered tasks, while *Thjis* planned 331, in 7 seconds. Adding 3 more minutes to *Local Search* only plans 10 more tasks, suggesting that the bulk of the results is in the first 60 seconds.

In Instance 3, a similar scenario occurs: RL (4 mins) plans only 21 tasks out of 734. RL (60 min) plans 294 tasks which is less than *Local Search* (1 min) which plans 322. In this instance, *Thjis* plans 394 tasks in 62 seconds.

In Instance 4, the RL algorithm running for 4 min only plans 11 out of 1591 tasks; running for 1 hour increases the number of planned tasks to 105. Here again, the RL results are poor when compared to *Local Search* and *Thjis*.

As expected and desired, the relative performance of an algorithm doesn't depend on the instance.

The failure of RL is attributed to the weakly connected states of the big problems. For state S_1 to be connected with state S_2, the solution in S_2 must be obtained by inserting a task in some duty in the solution of S_1. For big problems, most of the states only connect to one state, which makes the agent's learning redundant.

This results show great potential on *Local Search*, but further testing must be taken. Some additional code optimization is also suggested.

9. Conclusions

This project started with the goal of solving the OCRSP with Reinforcement Learning techniques. We tried tabular Q-learning with Eligibility Traces but the results were not good enough: in hindsight, we can see that the agent's learning is slow because the states for production-size problems are weakly connected. To be able to produce something commercially useful, various turns were made. Finally, we reached the *Local Search* algorithm which gave good results for the OCRSP.

The making of this project was challenging and rewarding. I learned a lot about areas I'm passionate about, such as, Artificial Intelligence, Operations Research and Software Development.

Trying to solve OCRSP with tabular Q-learning with Eligibility Traces proved impossible. In future research, it might be explored the use of Q-learning with state features and Q-value function approximation.

The Local Search algorithm is powerful enough to be tried in production. It was integrated in SISCOG's CREWS code base for further testing and optimization.

References

[1] G. Hondet, L. Delgado, and G. Gurtner. Airline Disruption Management with Aircraft Swapping and Reinforcement Learning. In *SESAR Innovation Days 2018 (SID 2018)*, Salzburg, Austria, Dec. 2018.

[2] D. Huisman. A column generation approach for the rail crew re-scheduling problem. *European Journal of Operational Research*, 180(1):163 – 173, 2007.

[3] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97 – 109, 1985.

[4] E. Morgado and J. P. Martins. Automated real-time dispatching support. In *Proceedings of the 2012 APTA rail conference*, 2012.

[5] D. Potthoff, D. Huisman, and G. Desaulniers. Column generation with dynamic duty selection for railway crew rescheduling. *Transportation Science*, 44(4):493–505, 2010.

[6] N. J. Rezanova and D. M. Ryan. The train driver recovery problem—a set partitioning based model and solution method. *Computers Operations Research*, 37(5):845 – 856, 2010. Disruption Management.

[7] K. Sato and N. Fukumura. Real-time freight train driver rescheduling during disruption. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E94.A(6):1222–1229, 2011.

[8] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[9] T. Verhaegh, D. Huisman, P.-J. Fioole, and J. C. Vera. A heuristic for real-time crew rescheduling during small disruptions. *Public Transport*, 9(1):325–342, Jul 2017.

[10] C. G. Walker, J. N. Snowdon, and D. M. Ryan. Simultaneous disruption recovery of a train timetable and crew roster in real time. *Computers Operations Research*, 32(8):2077 – 2094, 2005.

[11] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer, 1995.

[12] D. Šemrov, R. Marsetič, M. Žura, L. Todorovski, and A. Srdic. Reinforcement learning approach for train rescheduling on a single-track railway. *Transportation Research Part B: Methodological*, 86:250 – 267, 2016.