The Trip Scheduling Problem

Claudia Archetti
Department of Quantitative Methods, University of Brescia
Contrada Santa Chiara 50, 25122 Brescia, Italy

Martin Savelsbergh
School of Industrial and Systems Engineering, Georgia Institute of Technology
Atlanta, GA 30332-0205, U.S.A.

Abstract

The hours of service regulations of the department of transportation severely restrict the set of feasible driver schedules. So much so that establishing whether a sequence of full truckload transportation requests, each with a dispatch window at the origin, can feasibly be executed by a driver is no longer a matter of simple forward simulation. We consider this problem and prove that the feasibility of a driver schedule can be checked in polynomial time by providing an $O(n^3)$ algorithm for establishing whether a sequence of full truckload transportation requests, each with a dispatch window at the origin, can be executed by a driver.

1 Introduction

Truckload transportation represents a significant portion of all land-based freight transportation. In truckload transportation, full truckloads have to be picked up at an origin location and delivered at a destination location. A load dispatch window specifies the earliest time a load is ready for pickup at the origin location and the latest time that the load can be picked up at the origin location and still reach the destination location at a desired time. The challenge in truckload transportation is minimizing the (empty) repositioning of vehicles between a delivery location and a subsequent pickup location, because no revenues are generated on such a deadhead move, only costs are incurred. Demand uncertainty (transportation requests become available dynamically over time) makes minimizing repositioning especially challenging. A body of literature exists on truckload transportation, see for example Powell ([5, 6]), Powell et al. ([7]), Yang et al. ([8]), Regan et al. ([9]), and Powell et al. ([10]). Unfortunately, most of the literature ignores a crucial real-life complexity: driver restrictions. Governments impose restrictions on truck drivers to ensure their safety as well as the safety of other drivers. In the United States, for example, the Hours of Service (HOS) regulations ([2]) of the Federal Motor Carrier Safety Administration mandate that a driver cannot drive for more than 11 hours and cannot be on duty for more than 14 hours before a mandatory rest period of at least 10 hours (there are additional

restrictions, but these two are the most relevant and most restrictive). It is obvious, especially given the fact that truckload transportation often involves loads moving over long distances, that decision technology for scheduling truckload transportation requests which ignores driver restrictions is of limited, if any, value. A truckload transportation company has to consider HOS rules whenever it prepares a driver schedule for a trip taking more than one day (which is common, especially in the US, where individual moves often take more than a day). One of the few papers explicitly acknowledging the importance and complexity of handling driver restrictions is Xu et al. ([4]). The paper contains illustrative examples of how HOS rules impact the timing of a trip (i.e., arrival and departure times at pickup and delivery locations). The authors observe, among other things, that due to the presence of dispatch windows at origin locations of loads and the HOS regulations governing drivers, it is not necessarily optimal for a truck to depart from its home base as early as possible in order to complete a trip as early as possible. In fact, they conjecture that the "trip scheduling" problem, i.e., determining a departure time that results in the earliest return time, is NP-hard. Their trip scheduling problem is more complex than the one we consider in this paper, as it considers HOS regulations as well as multiple time windows at both pickup and delivery locations (and a complex cost function involving fixed costs, mileage costs, waiting costs and layover costs), but our results indicate that their conjecture may not be true. More specifically, we show that it is possible to establish in polynomial time whether a given sequence of transportation requests with a single dispatch window at the origin of the request (but no delivery window at the destination) can be feasibly executed by a driver that is available during a certain time period, i.e., respecting HOS regulations.

There are several other non-trivial scheduling problems involving just a single route. In the context of inventory routing problems, Campbell and Savelsbergh ([1]) develop a linear time algorithm for determining a delivery schedule for a route, i.e., a given sequence of customer visits, that maximizes the total amount of product that is delivered on the route. The algorithm has to properly account for the two dueling effects of increased inventory holding capacity at customers as time progresses and increased delivery times as more product is delivered at customers. In the context of dial-a-ride problems, Hunsaker and Savelsbergh ([3]), develop a linear time algorithm for determining whether a feasible schedule exists for a route, i.e., for a given sequence of pickups and deliveries, in the presence of maximum wait time and maximum ride time restrictions.

The remainder of the paper is organized as follows. In Section 2, we formally introduce the Trip Scheduling Problem and we provide a few examples illustrating the complexities encountered when solving the Trip Scheduling Problem. In Section 3, we present a polynomial algorithm for the Trip Scheduling Problem, prove its correctness, and derive its run time complexity. In Section 4, we analyze more general cases and prove that the algorithm proposed in Section 3 can be adapted to handle these generalizations. In Section 5, we investigate a variant of the Trip Scheduling Problem. Finally, in Section 6, we offer some final remarks.

2 Problem Definition

Truckload transportation problems are characterized by a set of transportation requests I, with for each transportation request $i \in I$ a pickup location i^+, a delivery location i^-, and a dispatch window at the pickup location $[e_{i^+}, \ell_{i^+}]$. If a truck arrives before the opening of the dispatch window, it has to wait. A truck must arrive before the closing of the dispatch window, because otherwise it will not reach the delivery location in time. A truck can serve one transportation request at a time. We assume that pickup and delivery of a load happen instantaneously, i.e., loading and unloading times are ignored. A truck driver can drive at most τ_{drive} hours and can be on duty at most τ_{duty} hours before a mandatory rest of at least τ_{rest} hours. Duty time includes driving time and waiting time.

The Trip Scheduling Problem (TSP) is to determine, given a sequence of transportation requests S, whether a trip exists that feasibly executes the transportation requests in S, i.e., whether or not dispatch times at the origins of the transportation requests in S can be found that correspond to a driver schedule respecting the HOS regulations. The ideas and techniques presented can easily be adapted to handle a central facility where the driver starts and ends. Since the sequence of transportation requests is given and there are no time restrictions at the delivery locations, it is possible to focus completely on dispatch times at the origin locations of the loads. In the absence of driver restrictions, the arrival time at the origin location of the next load is completely determined by the departure time at the origin of the current load (departure time plus travel time to the destination location plus the relocation time from the destination location to the origin location of the next load). Therefore, in the remainder when we refer to the driving time between two locations j and k, denoted by t_{jk}, we mean the driving time from the pickup location j^+ to the delivery location j^- plus the driving time from the delivery location j^- to the pickup location k^+.

The following examples illustrate that in the presence of driver restrictions it is unlikely that simple dispatching rules, such as "always departing as early as possible" and "always rest as late as possible," will allow us to establish the existence of a feasible trip.

Consider the situation depicted in Figure 1. Each horizontal line in the figure represents a time-line for the origin location of a transportation request. A set of consecutive transportation requests is represented by a set of consecutive time-lines with the first transportation request at the bottom and the last transportation request at the top. A pair of square brackets on a time-line represents the dispatch window associated with the corresponding request. The numbers above or below a square bracket represent the opening or closing time of the dispatch window. The numbers next to the double arrows on the right of the figure indicate the driving time between two locations. The colored path from the first to the last location represent a driver schedule: a slanted segment (blue segment) represents that the vehicle is driving, whereas a horizontal segment represents waiting time (red segment) or rest time (green segment). Numbers along the path represent either an arrival or a departure time at a location or the start or end time of a rest. The values of the

parameters of the HOS regulations are: $\tau_{duty} = 14$ hours, $\tau_{drive} = 11$ hours and $\tau_{rest} = 10$ hours. The top part of Figure 1 shows that when we depart as early as possible and rest as late as possible, we arrive at the origin of the third transportation request too late. (Note that the rest between the second and third location is a result of the duty time limit τ_{duty}.) At first, we may think that departing later from the first location, thereby avoiding the waiting time at the second location, may help, but we soon find that it does not, as it will not eliminate the rest between the second and third location. The only way to feasibly visit all three locations is to rest early, as soon as we arrive at the second location, as shown in the bottom part of the Figure 1.

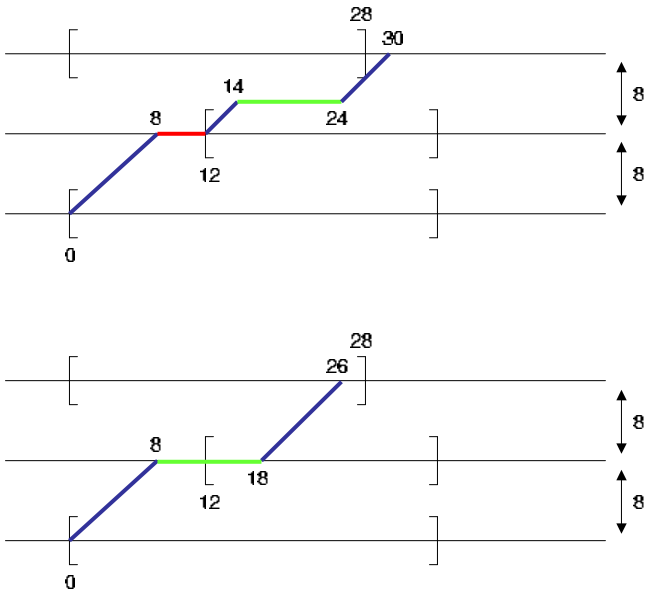

Figure 1: Rest earlier

Next, consider the situation depicted in Figure 2. The top part of Figure 2 shows that, again, departing as early as possible and resting as late as possible leads to a late arrival at the origin of the third transportation request. However, resting immediately upon arrival at the origin of the second transportation request, which lead to a feasible solution in the previous situation, does not produce a feasible solution. A feasible solution does exist though. Because the travel time between the second and third location is smaller than in the previous example, it is possible to eliminate the rest between the second and third

location by departing later, as shown in the bottom part of the Figure 2.

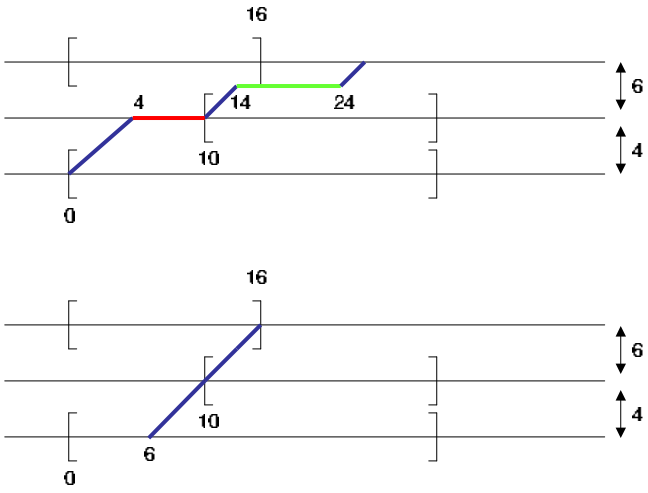

Figure 2: Depart later

Finally, consider the situation depicted in Figure 3. As before, the top part of Figure 3 shows that, departing as early as possible and resting as late as possible leads to a late arrival at the origin of the third transportation request. However, because of the tight dispatch window at the second location and the travel times between the different locations "resting earlier" (as shown in the middle part of Figure 3) and "departing later" (as shown in the bottom part of Figure 3) do not lead to a feasible solution either.

3 A Polynomial-Time Algorithm for the Trip Scheduling Problem

In this section, we present a polynomial-time algorithm for the Trip Scheduling Problem, i.e., an algorithm, called SMARTRIP, that produces a feasible dispatch schedule in polynomial time if one exists. For ease of presentation, we initially focus on the variant in which a rest takes exactly τ_{rest} hours.

Discussion
Before presenting the details of the algorithm, we provide the intuition behind it. Observe that the trip time is the sum of the travel time, the waiting time, and the rest time. The travel time, of course, is fixed as the sequence of transportation requests is given. Therefore, the challenge is to determine when to rest (and thus when to wait). Among the set of feasible dispatch schedules there is one that has a minimum number of rests. Furthermore, among

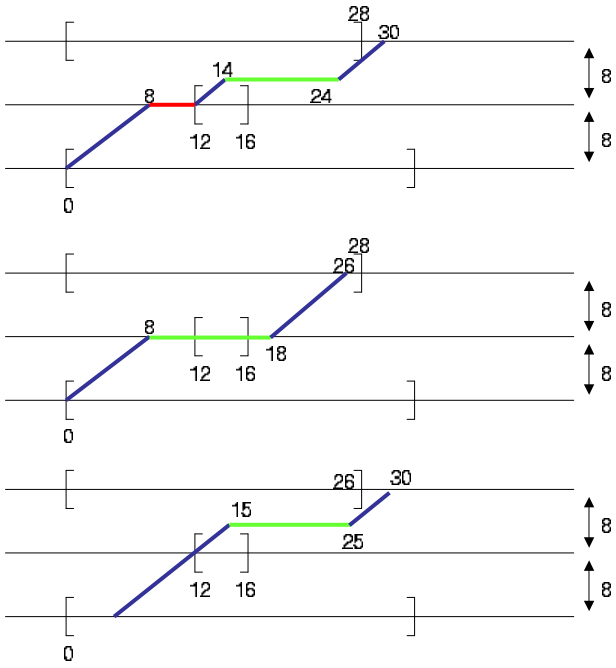

Figure 3: Infeasible

the set of feasible dispatch schedules with a minimum number of rests there is one where the rests are taken as early as possible. Our algorithm finds such a dispatch schedule, if it exists. We construct the dispatch schedule in reverse order, i.e., we start from the last location visited in the trip. The reason for this is that dispatch window feasibility is determined by the closing of the window and therefore we know that the last time we can visit the last location is at the closing of its dispatch window. We work backwards from there.

The key component of the algorithm is a backward search through the trip that starts from the last location and constructs a path respecting the dispatch windows and the driving and duty time limits in which rests are taken as late as possible (and thus as early as possible in a forward direction). If by doing so, we reach the first location, then we have found the feasible dispatch schedule we are looking for; simply traverse the constructed path in the opposite direction. On the other hand, if we encounter an infeasibility, i.e., we arrive at a location before the opening of the dispatch window, we conclude that we have been too aggressive and backtrack. If a feasible path exists, then at least one of the rests should have been taken earlier (and thus later in a forward direction). Therefore, we backtrack, adjust the start time of a rest, and start the backward search again. This process is repeated until either a feasible path has been found (and thus a feasible dispatch schedule) or we have established that no feasible path, and thus no feasible dispatch schedule, exists.

Algorithm

As mentioned above, the key component of the algorithm is a backward search through the trip. Assume that the trip is given by a sequence of (origin) locations $(1, 2, ..., n, n+1)$. For each i, the backward search attempts to construct a path from $n+1$ to i respecting driving and duty time limits, with a minimum number of rests, and with maximum remaining duty and drive time upon arrival at i. The latter implies that we have taken rests as late as possible. Such a path may not be unique, as the same path may be feasible for different start times; we say there is slack on the path. Slack occurs as a result of the width of the dispatch windows and the difference between the duty time and driving time limits. For convenience, we choose to work with the path that starts as early as possible at every location (among those feasible paths with a minimum number of rests and with maximum remaining duty and drive time upon arrival at i), which can be done by monitoring and using the slack appropriately.

A detailed description of SMARTRIP is given in Algorithm 1 where the following variables are used:

- *t*: the arrival time,
- *remDuty*: the remaining duty time,
- *remDrive*: the remaining driving time,
- *slack*: the amount by which the arrival time can be adjusted without affecting the remaining duty and remaining driving time.

In the discussion below, which focuses on the backward search, notation and terminology are consistent with the backward nature of the search. Thus, the arrival time t at location i is with respect to the path from $n+1$ to i. Similarly, the departure time at location i is the departure from i to $i-1$.

Since the slack is used extensively to avoid waiting time along the path, we elaborate on it before explaining the details of the backward search. Its initial value at location $n+1$ is $l_{n+1} - e_{n+1}$; we can change the departure time l_{n+1} by as much as $l_{n+1} - e_{n+1}$ without affecting the remaining duty and remaining driving time (the driver always departs fresh from $n+1$). We cannot change the departure time more than $l_{n+1} - e_{n+1}$ because then we would no longer depart within the dispatch window. The slack on the path may grow when the driver takes a rest. A driver starts a rest as soon as he reaches his maximum driving time limit. If there is any remaining duty time, then that remaining duty time has to be added to the slack. Postponing the start of the rest (up to the remaining duty time) does not affect the remaining duty and remaining driving time at the next location. The slack on the path may decrease for two reasons. First, we may use the slack to avoid waiting time. This occurs when we arrive at a location after the closing of the dispatch window ($t > l_i$). We exploit slack on the path, if any, to reduce the waiting time. Second, we may have to

adjust the slack because of the opening of the dispatch window. We can never adjust the departure time at a location beyond the opening of the dispatch window. Therefore, the slack can never be more than the difference between the departure time and the opening of the dispatch window. Consider the instance depicted in Figure 4. The top part of the

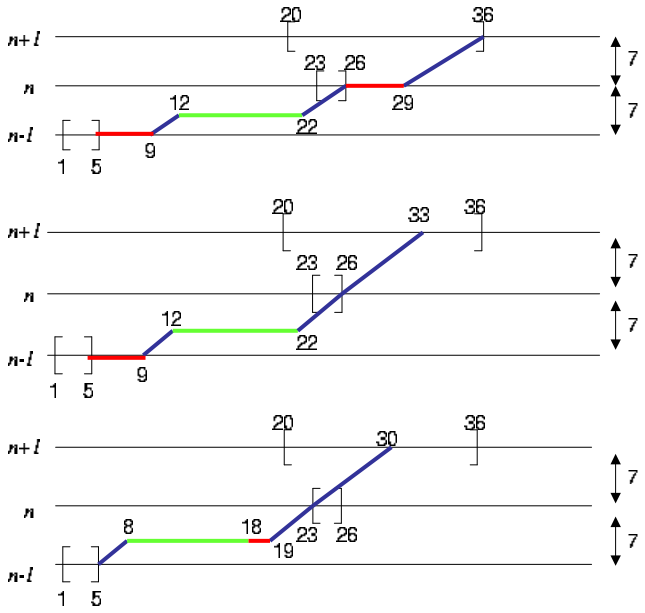

Figure 4: Exploiting slack

figure shows the path constructed by the backward search if we would not exploit slack along the path. The driver starts fresh from location $n+1$ at time $l_{n+1} = 36$. After 7 hours of driving, he reaches location n at time 29 and has to wait until $l_n = 26$ to serve location n. He departs location n at time $l_n = 26$ and drives for 4 hours before reaching his maximum driving time limit. He starts his rest immediately at time 22. At the end of his rest, at time 12, the driver continues to location $n-1$ which is reached at time 9. Here again the driver has to wait until $l_{n-1} = 5$ before being able to serve location $n-1$. At the time of departure from location $n-1$, the path has one rest, 7 hours of waiting time, and the remaining driving time and the remaining duty time are 7 hours. By exploiting slack, we can do much better. This is shown in the bottom part of the figure. By departing from location $n+1$ at time 30, we reach location n at time $e_n = 23$. After departing from location n at time $e_n = 23$, we reach the maximum drive time limit at time 19, but instead of going to rest immediately, we postpone the start of the rest until time 18. After ending the rest at time 8, we drive for three hours and reach location $n-1$ at time $l_{n-1} = 5$. At the time of departure from location $n-1$, the path has one rest, no waiting time, the

remaining driving time is 8 hours, and the remaining duty time is 11 hours. The backward search obtains this path by making the necessary adjustments to slack at each location. The middle part of the figure shows the adjustments that are made by the backward search at location n, where 3 hours of slack time were used to avoid waiting time. Table 1 shows the slack at the arrival time and at the departure time for the three locations in the above example.

Table 1: Slack computations for the example

	arrival	departure
$n+1$	-	16
n	16	3
$n-1$	6	2

The details of the backward search are given in Algorithm 2. Even though it is not necessary to explicitly keep track of the waiting time between two consecutive locations to establish feasibility of the path during the backward search, this information is necessary in Algorithm 3 when we try to restore feasibility. Therefore, we do compute and store $wait_i$, the waiting time on the constructed path between $i+1$ and i. (Waiting is handled properly in the backward search either by reducing the slack (avoid waiting) or by reducing the remaining duty time (account for waiting)).

Before proving an invariant property of the backward search, we describe its main steps. Given the initializations in SMARTRIP (lines 1-5 of Algorithm 1), a fresh driver starts at location $n+1$ at time $t = l_{n+1}$, i.e., the remaining duty time and the remaining driving time of the driver are equal to τ_{duty} and τ_{drive}, respectively. The slack of the path is initialized at $l_{n+1} - e_{n+1}$, since the driver can depart location $n+1$ at any time between l_{n+1} and e_{n+1} without affecting the remaining driving and duty time at $n+1$. The backward search is called with parameters $j = n$ and $k = 1$. Parameter j is the next location on the path to be built and parameter k is the location where the construction stops (if no infeasibility is detected along the way). In each iteration of the backward search, we start by computing the arrival time at (the next) location i. Once the arrival time t at i has been determined, we analyze it with respect to the dispatch window at i. If the arrival time is before the opening of the dispatch window ($t < e_i$), then the algorithm stops; we have failed to establish a time-feasible dispatch schedule from i to $n+1$. If, on the other hand, the arrival time is greater than or equal to the opening of the dispatch window ($t \geq e_i$), then a time-feasible dispatch schedule from i to $n+1$ exists and the algorithm attempts to extend it to a time-feasible dispatch schedule from $i-1$ to $n+1$. Two situations have to be considered. If the arrival time falls within the dispatch window at i, then the algorithm simply proceeds to the next iteration. Otherwise, waiting time *may* be incurred. As we want to rest as

Algorithm 1 SMARTRIP

1: ***** *Initialization* *****
2: $t \leftarrow l_{n+1}$, $remDuty \leftarrow \tau_{duty}$, $remDrive \leftarrow \tau_{drive}$, $slack \leftarrow (l_{n+1} - e_{n+1})$
3: ***** *Start backward search* *****
4: $Done \leftarrow False$
5: $j \leftarrow n$
6: **repeat**
7: $i \leftarrow BackwardSearch(j, 1)$
8: **if** $i = 1$ **then**
9: $Done \leftarrow True$ // A feasible trip has been found
10: **else**
11: $Restored \leftarrow Restore(i)$
12: **if** $Restored = Success$ **then**
13: $j \leftarrow i - 1$
14: **else**
15: $Done \leftarrow True$ // No feasible trip exists
16: **end if**
17: **end if**
18: **until** $Done$

late as possible, we want to avoid waiting time along the path because waiting time, which counts towards duty time, may cause us to rest earlier. The algorithm exploits any slack in the path from i to $n+1$ to eliminate waiting time (exploiting slack time does not affect remaining duty and drive time). While exploiting slack, two cases can occur. First, there is enough slack in the path to avoid waiting time altogether. This happens if the slack to move the path earlier in time is greater than the difference between the arrival time and the closing of the dispatch window ($slack \geq t - l_i$). In that case, necessary updates are made and the algorithm proceeds to the next iteration. Second, waiting time is unavoidable. This situation requires more complex updating before the algorithm can proceed to the next iteration, as the waiting time may be larger than the remaining duty time, which forces the driver to rest (maybe more than once).

A more elaborate discussion of the steps of the backward search is presented below. Given the input parameters j and k, for each location $i = j, j-1, ..., k$ the backward search performs the following steps:

1. *Calculate the arrival time at location i*: lines 3-11.

   Given the driving time from location $i+1$ to i ($t_{i,i+1}$) and the departure time t from location $i+1$, the arrival time at i is calculated. Two cases have to be distinguished:

   - The remaining driving time at $i+1$ is sufficient to cover the entire driving time

from $i+1$ to i. In this case, the arrival time at i is simply given by $t - t_{i,i+1}$ and the remaining driving and duty time are also updated as $remDrive - t_{i,i+1}$ and $remDuty - t_{i,i+1}$.

- The remaining driving time at $i+1$ is less than $t_{i,i+1}$. In this case, at least one rest has to be taken going from $i+1$ to i. The number of rests $nRests$ needed is $1 + \lfloor \frac{t_{i,i+1} - remDrive}{\tau_{drive}} \rfloor$; the first when $remDrive = 0$ and the other $\lfloor \frac{t_{i,i+1} - remDrive}{\tau_{drive}} \rfloor$ to cover the remainder of the driving time. To calculate the arrival time at i, we have to consider both the driving time $t_{i,i+1}$ and the time spent resting. Thus, the arrival time is $t - (t_{i,i+1} + nRests \times \tau_{rest})$. The remaining driving and duty time have to be calculated with respect to the last rest (lines 9-10). Also, each time a rest is taken (with possibly the exception of the first), the start can be delayed by any remaining duty time, i.e., $\tau_{duty} - \tau_{drive}$. The rest is taken because the driving time limit is reached; not because the remaining duty time limit has been reached. Any remaining duty time introduces additional *slack* (line 8).

2. *Evaluate the arrival time t at location i and calculate the departure time from i*: lines 13-53.

   Three cases have to be distinguished:

   - The arrival time t is before the opening of the dispatch window of location i, i.e., $t < e_i$ (lines 13-14). In this case, a dispatch window violation is detected: the backward search stops returning the index i.
   - The arrival time is inside the dispatch window of location i, i.e., $e_i \leq t \leq l_i$ (lines 16-17). In this case, the driver can immediately continue to the next location. The slack may have to be updated, since the slack can be no more than $t - e_i$ at location i.
   - The arrival time t is after the closing of the dispatch window of location i, i.e., $t > l_i$ (lines 19-52). Two situations have to be distinguished: no rest is needed to reach the closing of the dispatch window l_i and at least one rest is needed to reach the closing of the dispatch window l_i. A rest is needed if there is not enough slack and remaining duty time to reach l_i, i.e., if $t - slack - remDuty > l_i$ (line 19).
     (a) In the first situation, i.e., no rest is needed, slack is exploited first before using up any remaining duty time (lines 22-29). That is, the remaining duty time is only affected if $t - slack > l_i$ (lines 23-25). If necessary, the remaining duty time (and eventually the remaining driving time) is decreased by $(t - slack) - l_i$ (line 25).
     (b) In the second situation, i.e., at least one rest is needed, the driver exploits all slack and consumes all remaining duty time before taking a rest. If another

rest is needed, this additional rest can be taken after consuming an entire duty period τ_{duty}. Thus, the (minimum) number of rests needed to reach l_i is equal to $\lceil \frac{(t-slack-remDuty)-l_i}{\tau_{rest}+\tau_{duty}} \rceil$ (line 32). If the last rest ends after l_i, the driver has to wait to reach l_i. This waiting time will affect the remaining driving and duty time (lines 48-49). The departure time will be l_i (line 47). If instead the last rest ends before l_i (lines 34-45), in all but one case the driver can depart from location i at $t = l_i$ completely fresh (line 45). By adjusting the duty time between the second to last and the last rest, we can ensure that the driver comes off rest at exactly l_i. Of course, if there is only one rest, we have to be more careful (lines 35-40). If the arrival time t at location i minus the rest time τ_{rest} is less than e_i, then a dispatch window violation is detected (lines 36-37). (A rest is needed, since $t - slack - remDuty > l_i$, but even taking the rest immediately upon arrival causes a dispatch window violation.) If the single rest can be taken feasibly and $t - \tau_{rest} < l_i$, then the driver cannot depart at l_i, but can only leave at $t - \tau_{rest}$ (line 39). All this has to be accounted for in the slack as well, which, thus, is equal to the minimum between t and the earliest departure time, on one hand, and $t - e_i$ on the other hand (line 44).

If the index i returned by the backward search is equal to k then the backward search succeeded in finding a feasible schedule from j to k. If instead i is greater than k this means that a dispatch window violation has been encountered at location i.

The theorem below establishes an important invariant property of the path constructed by Algorithm 2.

Theorem 1 *The path from $n+1$ to i constructed by Algorithm 2 has the minimum possible number of rests. Among the paths with a minimum number of rests, the remaining duty time and the remaining driving time at the departure from i on the constructed path are as large as possible. Moreover, among the paths with the minimum number of rests and maximum remaining duty and driving time, the slack on the path constructed is as large as possible.*

Proof: By induction. The claim is trivially true for $i = n + 1$. Suppose that the claim is true for $i = k + 1$. We will prove that it is also true for $i = k$.

In the first step, i.e., *Calculate the arrival time at location i*, Algoritm 2 calculates the time needed to reach k. If no rest is needed to reach k, the number of rests does not change and thus the path still has the minimum number of rests. Moreover, since we started with maximum values for $remDuty$ and $remDrive$ and we decrease these values only by the driving time between $k + 1$ and k, they are still at their maximum values. Since, all time

between $k+1$ and k was taken up by driving, no additional slack can be introduced in the path. If at least one rest is needed, the number of rests taken is the minimum possible to reach k, because each rest is taken only when $remDrive = 0$. At the same time, this ensures that k is reached with maximum values $remDrive$ and $remDuty$. Because at least one rest occurs between $k+1$ and k, there may be additional slack. Any difference between the remaining duty and the remaining driving time at the departure from $k+1$ can be converted to slack and, for every rest but the first, the difference between the duty time and driving time limits can be converted to slack. As all these conversions are carried out, the slack is still as large as possible.

In the second step, i.e., *Evaluate the arrival time at location i and calculate the departure time from i*, the arrival time t at k is analyzed and three cases have to be considered. In the first case, i.e. $t < e_k$, the backward search stops because it has failed to construct a time-feasible path between k and $n+1$. In the second case, i.e. $e_k \leq t \leq l_k$, as the path has the minimum number of rests, $remDuty$ and $remDrive$ have maximum possible values, and $slack$ is as large as possible, the backward search proceeds to the next iteration. In the third case, i.e., $t > l_k$, as we need to depart at a time that falls inside the dispatch window, waiting time and even rest time may occur. To avoid rest time and waiting time as much as possible, we exploit any slack on the path. Two cases have to be considered: additional rests can be avoided and additional rests are unavoidable. When no additional rest is needed, i.e., when $t - (slack + remDuty) \leq l_k$, the number of rests on the path does not change and thus is still minimum. Furthermore, if it is necessary to decrease $remDuty$ (and thus maybe $remDrive$) to reach l_k, i.e., when $slack < t - l_k$, then $remDuty$ and $remDrive$ are decreased by the minimum possible quantity (line 25). Therefore, they are still at their maximum values. When at least one rest is needed, i.e., when $t - (slack + remDuty) > l_k$, rests need to be (and are) taken as late as possible. The first rest can start at time $tBest = t - (slack + remDuty)$. After a rest, it is possible to wait for a maximum of τ_{duty} before making another rest. Thus, the minimum number of additional rests $nRests$ needed to reach l_k is given by $\lceil \frac{tBest - l_k}{\tau_{rest} + \tau_{duty}} \rceil$ (line 32). The total time covered by the minimum number of rests is $nRests \times \tau_{rest} + (nRests - 1) \times \tau_{duty}$. Let t' be equal to $tBest - nRests \times \tau_{rest} - (nRests - 1) \times \tau duty$. Since we want to depart from k as early as possible in order to maximize the remaining duty and remaining driving time as well as the slack, i.e., preferably at l_k, we need to distinguish two cases: $t' \leq l_k$ and $t' > l_k$. In the former case, with the exception of a single rest that causes a dispatch window violation, we depart from k at the end of a rest and thus $remDrive$ and $remDuty$ have maximum values. In the latter case, i.e., $t' > l_k$, we have to wait until l_k, i.e., consume duty time. Regardless, we have added only the minimum number of necessary rests on the path and thus the total number of rests is still minimum. Furthermore, we only decrease $remDuty$ (and thus maybe $remDrive$) when absolutely necessary, because we always exploit slack in the path first, so $remDuty$ and $remDrive$ have maximum values at departure from k. Finally, the minimum

amount of slack necessary to maintain maximum remaining duty and maximum remaining driving time is consumed, so the slack is still as large as possible. Therefore, the claim is true. □

The complexity of Algorithm 2 is $O(n)$ since it simply goes backward through the n locations.

When the algorithm stops prematurely, it has identified a location i for which the constructed path does not provide a time-feasible dispatch schedule from i to $n+1$. The constructed path is made up of driving time, waiting time, and rest time. Since the path has a minimum number of possible rests, a feasible path from i to $n+1$ has to have less waiting time; the driving time and the rest time cannot be reduced. The first idea that may come to mind is that to reduce waiting time along the path, we should simply depart later as this will result in arriving later at locations that have waiting time and thus reduce this waiting time. Unfortunately, this "adjusting the departure time" is already embedded in the algorithm in the form of exploiting the slack of the path. Slack precisely captures our ability to adjust the departure time to reduce waiting time along the path. The only way to reduce waiting time is to convert waiting time into rest time by taking rests earlier in time ("by pushing up rests"). An example of reducing waiting time by taking rests earlier is depicted in Figure 5. The top part of Figure 5 shows a portion of a path that might be constructed by the backward search. The bottom part of Figure 5 shows how that portion of the path can be converted to a time-feasible dispatch schedule by pushing up the rest. Instead of taking the rest between the first and second location, the rest is taken later at

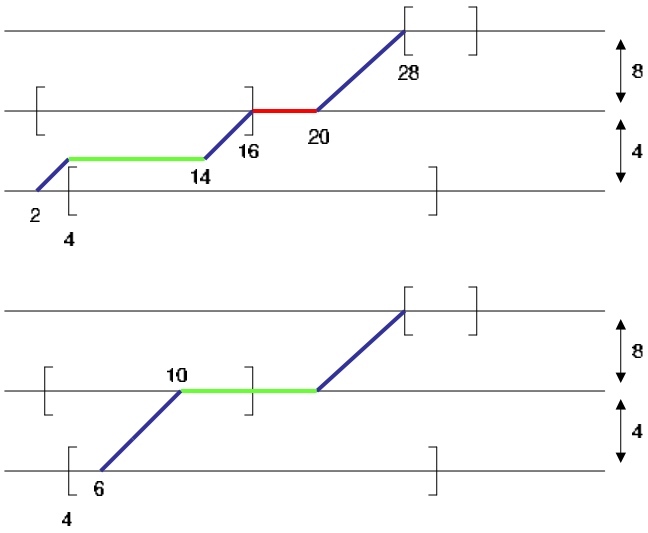

Figure 5: Converting waiting time to rest time ("pushing up" the rest).

the second location.

This suggests that if the backward search stops prematurely at i, an attempt should be made to restore feasibility by pushing up the nearest possible rests (when traversing the constructed path forward in time). That is, when a location j with waiting time is encountered and when there is at least one rest between i and j, then an attempt is made to push up a rest to j. The rationale behind this scheme is provided by the fact that if the rest(s) between i and j are not pushed up, then the arrival time at i does not change and thus feasibility will not be restored. In fact, the driver already leaves from location j (on the backward path) as early as possible, i.e., at $t = l_j$. Thus, the arrival time at i will not change unless at least one rest is pushed up. On the other hand, if there is no rest between i and j (the first location with waiting time), then there exists no time-feasible dispatch schedule. The (backward) path arrives at i as early as possible (i.e., the latest possible feasible dispatch time at i has been identified).

Thus, to restore feasibility, it is necessary to push up the rest(s) between i and j. Once the rest(s) between i and j is (are) pushed up to j, we revert to the backward search to see if i can now be reached feasibly.

These ideas are more formally captured in Algorithm 3, which determines a time-feasible dispatch schedule between i and $n+1$ or establishes that no such schedule exists. The input parameter i represents the location where a dispatch window violation occurred. Before describing the algorithm in detail, we need to introduce some new notation. Consider the path constructed by the backward search. Let

- t_i be the time of arrival at location i,
- $remDuty_i$ be the remaining duty time upon arrival at location i,
- $slack_i$ be the *slack* upon arrival at location i,
- $nRests_i$ be the number of rests between the time of arrival at i and the time of departure from i,
- $nRests_{ij}$ be the number of rests between the time of departure at i and the time of arrival at j.

This information can be easily captured in Algorithm 2. We did not include the relevant statements in the description of Algorithm 2 in order to keep it as clear and readable as possible.

When pushing up a rest to location j, we need to determine the proper values to initiate the backward search, i.e., t, $remDrive$, $remDuty$, and $slack$. We want to depart from j as early as possible (in order for the slack to be as large as possible). To determine the departure time, we consider the time of arrival at j (t_j) and the number of rests at j ($nRests_j + 1$; the original number of rests at j plus the rest being pushed up to j). If

$t_j - (nRests_j + 1) \times \tau_{rest} < e_j$, then $wait_j + (l_j - e_j) < \tau_{rest}$. As a consequence, performing a "wait-to-rest" conversion at location j would lead to a dispatch window violation at j. This does not imply that no time-feasible path exists, as it may be possible to push up the rest further and convert waiting time at locations j and k ($k > j$) to rest time. An example is depicted in Figure 6. The top part of the figure shows the dispatch window violation that causes a forward search to restore feasibility. When the rest is pushed-up to the first location with waiting time, shown in the middle part of the figure, the wait-to-rest conversion causes a dispatch window violation at the location where the waiting time is converted into rest time. However, when pushing the rest up even further, shown in the bottom part of the figure, then the wait-to-rest conversion does lead to a feasible path. Thus, in case a wait-to-rest conversion at location j leads to a dispatch window violation at j, Algorithm 3 simply pushes the rest up to the next location with waiting time and performs a wait-to-rest conversion there. If instead $t_j - (nRests_j + 1) \times \tau_{rest} \geq e_j$, then two

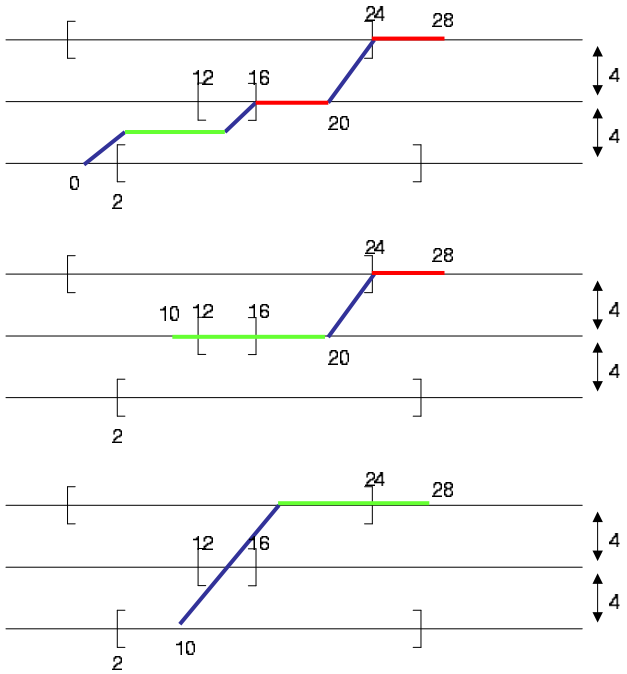

Figure 6: Conversion of two waiting times into rest time.

cases need to be considered: $t_j - (nRests_j + 1) \times \tau_{rest} \leq l_j$ and $t_j - (nRests_j + 1) \times \tau_{rest} > l_j$. In the first case (lines 7-11), the waiting time at j is less than τ_{rest} and the driver departs at $t_j - (nRest_j + 1) \times \tau_{rest}$. To depart as early as possible, the driver takes all rests consecutively. The driver starts fresh, i.e., $remDrive = \tau_{drive}$ and $remDuty = \tau_{duty}$. The

slack of the path is the minimum between $t - e_j$ and $slack_j + remDuty_j + nRest_j \times \tau_{duty}$, where the first two terms represent the slack already present in the original path and the last term captures the duty time between consecutive rests, if any. This situation is depicted in Figure 7. In the second case (lines 12-16), the waiting time at j is greater

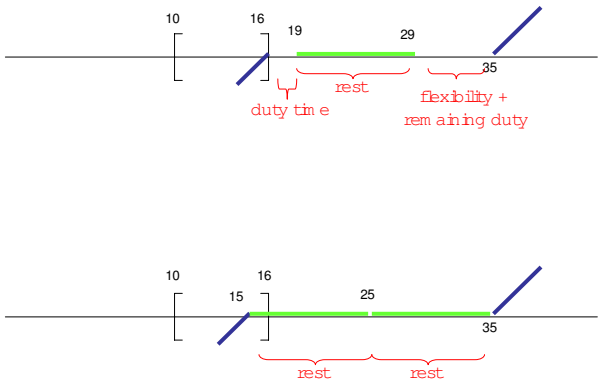

Figure 7: Pushing up a rest when the waiting time is less than the rest time.

than τ_{rest} and the driver can depart at l_j. The driver starts fresh, i.e., $remDrive = \tau_{drive}$ and $remDuty = \tau_{duty}$. The slack of the path is the minimum between $l_j - e_j$ and $slack_j + remDuty_j + nRest_j \times \tau_{duty} - (t_j - (nRest_j + 1) \times \tau_{rest} - l_j)$, where the last term captures the fact that some of the existing slack of the path has to be "consumed." This situation is depicted in Figure 8.

The main steps of Algorithm 3 are the following:

- *Find a location j to which we can push up a rest*: line 3. The algorithm searches forward from $i + 1$ and stops as soon as a location j is found with waiting time on the backward path and at least one rest on the backward path between i and j (the location where a dispatch window violation was detected).
- *Push up a rest to location j*: lines 5-17. This step has already been described.

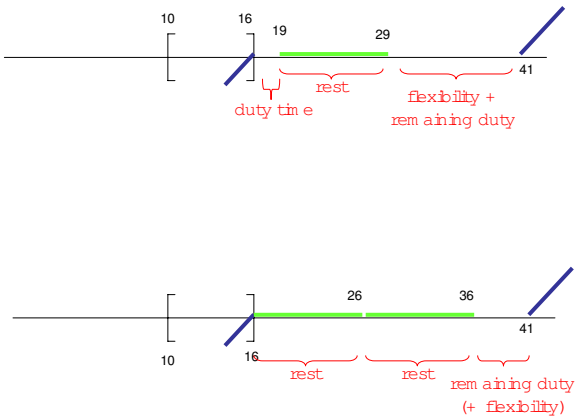

Figure 8: Pushing up a rest when the waiting time is larger than the rest time.

- *Initiate a backward search from j to i*: line 19. Once a rest has been pushed up, the backward search parameters are initiated, and backward search is used to constructed a path from i to j.
- *Check feasibility*: lines 20-21. If the path constructed by the backward search is feasible, i.e., the path reaches i at time $t \geq e_i$, the algorithm stops: a time feasible schedule to location i has been found. Otherwise, the forward search is resumed to find the next location to which we can push up a rest (line 25). If location $n+1$ is reached without restoring feasibility, then the algorithm stops: no time-feasible dispatch schedule exists (line 27).

It is important to realize that after a rest has been pushed up, it can never be pushed down again, because that would reintroduce infeasibility.

Theorem 2 *Algorithm 3 determines a time-feasible (backward) path from $n+1$ and i or establishes that no such path exists. The path has the minimum possible number of rests.*

Among the paths with a minimum number of rests, the remaining duty time and the remaining driving time at the departure from i on the constructed path are as large as possible. Moreover, among the paths with the minimum number of rests and maximum remaining duty and driving time, the slack on the path constructed is as large as possible.

Proof: Suppose Algorithm 3 does not return a time-feasible (backward) path from $n+1$ to i, i.e., it terminates without restoring the dispatch window violation at i. If the constructed path between $n+1$ and i has no waiting time, then clearly no time-feasible path exists, because the path has a minimum number of possible rests.

Thus, we consider the case where there is waiting time at at least one location along the constructed path.

First, consider the case where there is a single location with waiting time, say j. The fact that there is waiting time at j means that Algorithm 3 is not able to push up rests between i and j and convert waiting time at j into rest time. Either there were no rests between i and j, or $\tau_{rest} > wait_j + (l_j - e_j)$ in which case converting waiting time to rest time at j results in a dispatch window violation at j. In both cases, we conclude that there does not exist a time-feasible path between j and i (the path starts at l_j, has the minimum number of possible rests, has no waiting time, and arrives at i before e_i) and thus that there exists no time-feasible path between $n+1$ and i.

The case where there are multiple locations with waiting time is dealt with similarly. Consider the first location with waiting time encountered when traversing the path from i to $n+1$, say location j. Using the same arguments as for the case with a single location with waiting time, we conclude that there does not exist a time-feasible path between j and i and thus that there exists no time-feasible path between $n+1$ and i.

On the other hand, if a rest can be pushed up feasibly to location j, then the backward search is re-started with appropriately set parameters (lines 7-17) in an attempt to construct a feasible path from j to i. If successful, i.e., the path arrives at i at or after e_i, then feasibility has been restored. Otherwise, Algorithm 3 attempts to push up rests beyond j along the path to $n+1$, to explore whether that results in a different arrival time at j and thus a new chance to restore the infeasibility. Iteratively, either a time-feasible path from $n+1$ to i is found, or it is demonstrated that wait-to-rest conversions cannot lead to a time-feasible path from $n+1$ to i.

As Algorithm 3 relies on the backward search to determine a time-feasible path from $n+1$ and i the path has the desired properties. □

The complexity of Algorithm 3 is $O(n^2)$ since each wait-to-rest conversion attempt takes linear time (as it involves a backward search from j to i) and no more than n wait-to-rest conversions are performed during a feasibility recovery action (as there are at most n locations on the path from i to $n+1$).

We are now in a position to revisit the main algorithm, SMARTRIP, and present the main

theorem of the paper.

First, SMARTRIP initializes the parameters assuming that a fresh driver departs from location $n+1$ at time l_{n+1} (line 2). Then a backward search is initiated attempting to reach location 1 (line 7). If the location i returned is 1, then a feasible path from $n+1$ to 1 has been found and the algorithm stops (lines 8-9). Otherwise (the location i returned is greater than 1), a dispatch window violation has been detected at location i. Thus, SMARTRIP attempts to restore feasibility, i.e., find a time feasible path from $n+1$ to i (line 11). If successful, then a backward search is initiated to extend the path to 1, i.e., from $i-1$ to 1 (lines 12-13). Otherwise, no time feasible dispatch schedule exists and the algorithm stops (line 15).

Theorem 3 *Given a sequence of transportation requests $S = \{1, 2, ..., n\}$, SMARTRIP determines in $O(n^3)$ time whether a time-feasible dispatch schedule exists.*

Proof: SMARTRIP runs in polynomial time because at most n feasibility recovery actions are needed (there are only n locations on the path), thus Algorithm 3 is called at most n times (each call taking at most $O(n^2)$ time). □

Example. We illustrate the algorithms presented above on a small instance with $n = 6$. The instance is depicted in Figure 9. Algorithm 2 constructs the path depicted in Figure 10. Table 2 shows the values of the variables at the beginning and end of each iteration. The

Table 2: Values of Variables

	arrival				departure					
i	t	$slack$	$remDrive$	$remDuty$	$tBest$	t	$slack$	$remDrive$	$remDuty$	$wait_i$
7	-	-	-	-	-	100	25	11	14	-
6	75	28	7	10	-	75	0	7	10	-
5	55	3	8	11	41	45	0	4	4	10
4	25	0	5	8	-	25	0	5	8	-
3	21	0	1	4	17	18	0	1	1	3
2	6	0	10	13	-	6	0	10	13	-
1	0	0	4	7	-	-	-	-	-	-

driver departs fresh from location 7 at $t = l_7 = 100$. The slack is initialized at $l_7 - e_7 = 25$. He has to drive 15 hours to reach location 6 and therefore rests after 11 hours. At the start of the rest the difference between the remaining duty time and remaining driving time is 3, so the slack increases from 25 to 28. Location 6 is reached within the dispatch window at time $t = e_6 = 75$ with 7 hours of remaining driving time and 10 hours of remaining duty time. Because location 6 is reached at $t = e_6$, the slack of the path has to be set to zero. Next, the driver continues on to location 5 making a rest after 7 hours of driving. At the start of the rest the difference between the remaining duty time and remaining driving

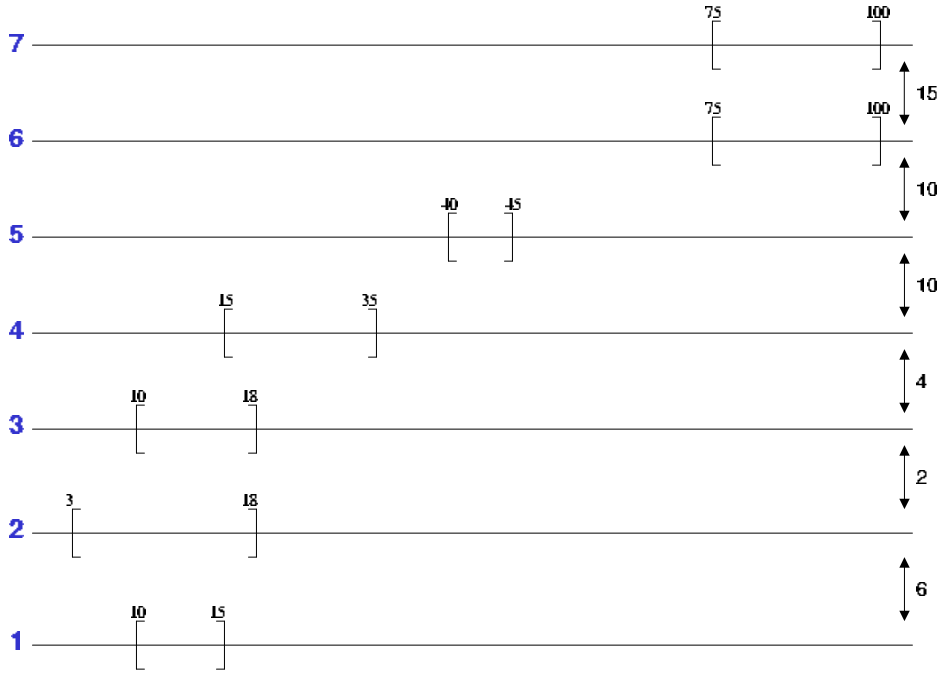

Figure 9: An instance of the TSP problem

time is 3, so the slack increases from 0 to 3. Location 5 is reached after the closing of the dispatch window at $t = 55$ ($l_5 = 45$) with 8 hours of remaining drive time and 11 hours of remaining duty time. Consequently, the driver has to wait. Fortunately, we can exploit the 3 hours of slack on the path by departing 3 hours later at location 6 (at time 72). Thus, the "real" waiting time at location 5 is 7 hours (which count towards duty time). Therefore, at the departure at location 5 (at time $l_5 = 45$), there are only 4 hours of remaining duty time (and thus also only 4 hours of remaining drive time). On his way to location 5, the driver has to make a rest after 4 hours of driving at time 41. At the start of the rest the difference between the remaining duty time and remaining driving time is 0, so the slack remains unchanged at 0. Location 4 is reached within the dispatch window at time $t = 25$ with 5 hours of remaining driving time and 8 hours of remaining duty time. The driver immediately continues to location 3, which is reached after the closing of the dispatch window at time $t = 21$ ($l_3 = 18$) with 1 hour of remaining driving time and 4 hours of remaining duty time. The driver has to wait 3 hours. There is no slack on the path, so these hours have to come out of the remaining duty time. Thus, the driver departs location 3 at time $t = l_3 = 18$ with 1 hour of remaining drive time and 1 hour of remaining duty time. On his way to

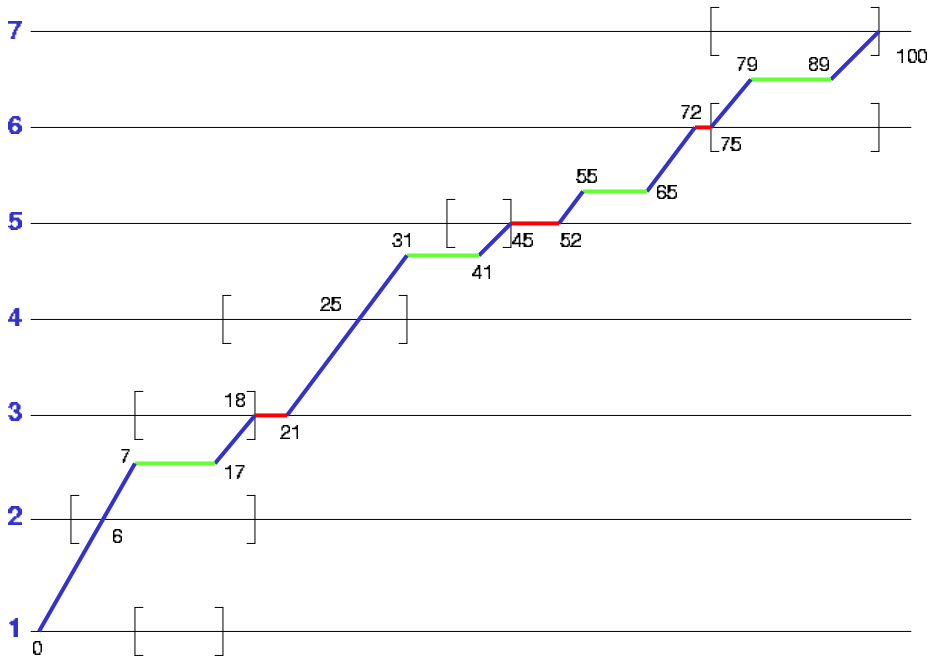

Figure 10: Path constructed by Algorithm 1

location 2, the driver has to take a rest after 1 hour of driving at time 17. At the start of the rest the difference between the remaining duty time and remaining driving time is 0, so the slack remain unchanged at 0. Location 2 is reached within the dispatch window at time $t = 6$ with 10 hours of remaining driving time and 13 hours of remaining duty time. The driver immediately continues to location 1, which is reached before the opening of the dispatch window at time $t = 0$ ($e_1 = 10$); a dispatch window violation is detected.

At this point, Algorithm 3 is called which pushes up the rest between location 2 and 3 to location 3, where there is a waiting time of 3 hours. Thus, these 3 hours of waiting time are converted into rest hours. Because of the wait-to-rest conversion, the driver now leaves location 3 at time $t = 11$ and, after 8 hours of driving, reaches location 1 at time $t = 3$. Since that still implies a dispatch window violation, Algorithm 3 continues and it pushes up the rest between location 4 and 5 to location 5, where there is a waiting time of 10 hours. Thus, these 10 hours of waiting are converted into rest hours. Because of the wait-to-rest conversion, the driver leaves location 5 at time $t = 45$. He arrives at location 4 at time $t = 35$, continues to location 3, making a rest after 1 hour of driving. (Note that this rest was previously pushed up to location 3 and is now pushed up even further between location 3 and location 4). Location 3 is reached at time $t = 21$, where the driver has to wait 3 hours. The driver leaves location 3 at time $t = 18$. After 8 hours of driving, the

driver reaches location 1 at time $t = 10$. Thus, a time-feasible dispatch schedule has been found (see Figure 11).

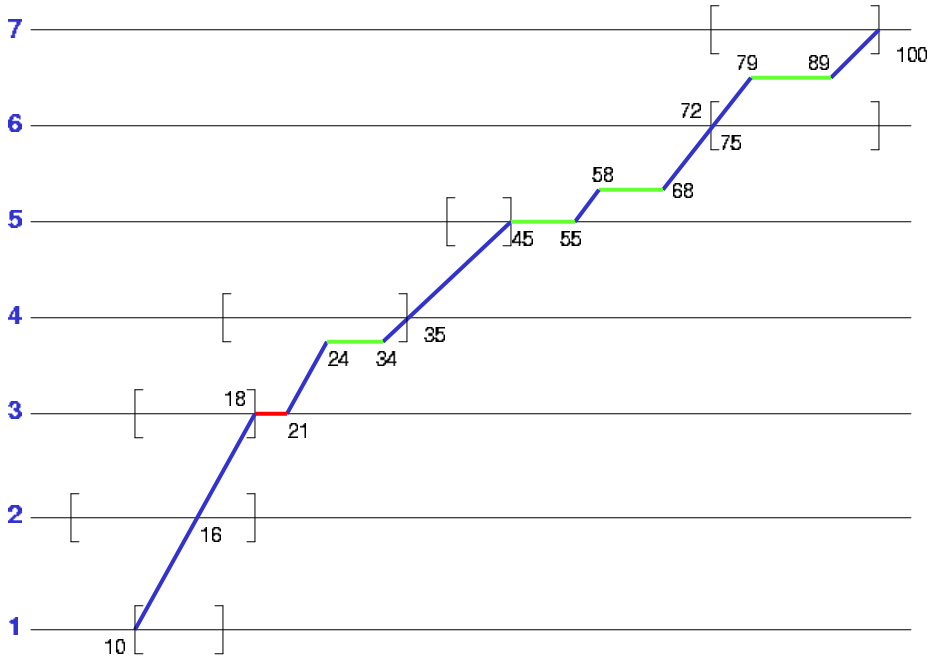

Figure 11: Final solution

4 The General Case

Up to now, we have assumed that the rest time is exactly τ_{rest} hours and that a rest can start at any time during a duty. As a consequence of the latter assumption, it is possible for a driver to take two consecutive rests, i.e., without any duty time in between.

Next, we show that the algorithms presented above can be easily modified to handle more realistic situations in which we have a minimum and maximum rest time, τ_{rest}^{min} and τ_{rest}^{max}, as well as a minimum and maximum duty time, τ_{duty}^{min} and τ_{duty}^{max}. The situation considered previously corresponds to $\tau_{rest}^{min} = \tau_{rest}^{max}$ and $\tau_{duty}^{min} = 0$.

First, we consider the case of variable rest time. To handle this situation, Algorithm 2 and Algorithm 3 need to be modified as follows:

- Whenever a rest is taken, the minimum possible rest time is assumed (τ_{rest} is replaced by τ_{rest}^{min}).

- Whenever the flexibility of the path is updated, the difference between τ_{rest}^{max} and τ_{rest}^{min} is properly accounted for ($\tau_{rest}^{max} - \tau_{rest}^{min}$ is included in *slack*).

The reason for these modifications is as follows: the option to choose the length of a rest adds flexibility to the construction of a time-feasible path. The two suggested modifications exploit his flexibility: each time a rest is made its duration is set to the minimum possible, i.e., τ_{rest}^{min}. To account for the fact that the rest can take up to τ_{rest}^{max}, we adjust *slack* accordingly. That is, instead of committing to a particular rest time, we set the rest time to the minimum required and capture the opportunity to rest longer in the slack in the path. Thus, to handle variable rest time only requires a change in the calculation of *slack*; nothing else changes in Algorithm 2 and Algorithm 3. Thus, Theorem 1, Theorem 2, and Theorem 3 still hold.

Next, we consider the case with a minimum duty time greater than zero. To handle this situation, the following changes need to be made:

- In Algorithm 2 and Algorithm 3 τ_{duty} corresponds with τ_{duty}^{max}.
- In the backward search, when we arrive at location i and find that at least one additional rest is required, two changes have to be made. First, "unused duty time" between consecutive rests is now at most $\tau_{duty}^{max} - \tau_{duty}^{min}$. Therefore, the updating of *slack* at line 44 of Algorithm 2 has to change to

  $$slack \leftarrow min(t - e_i, [t - (tBest - (nRests \times \tau_{rest}^{max} + (nRests - 1) \times (\tau_{duty}^{max} - \tau_{duty}^{min})))]).$$

  Second, we may not be able to go to rest immediately upon arrival, because we have not reached the minimum duty hour limit yet. The additional duty time that has to be expended before a driver can go to rest is $[\tau_{duty}^{min} - (\tau_{duty}^{max} - remDuty)]^+$. Therefore, at lines 35, 36 and 39 of Algorithm 2, $t - \tau_{rest}$ has to be replaced by $t - [\tau_{duty}^{min} - (\tau_{duty}^{max} - remDuty)]^+ - \tau_{rest}^{min}$.
- When trying to restore feasibility, similar issues arise. When we push up a rest, we have to ensure that this rest starts at the proper time and when there are multiple rests we have to ensure that the minimum duty time between rests is properly captured. Therefore, at lines 6 and 7 of Algorithm 3, $t_j - nRests_j \times \tau_{rest}$ has to be replaced by

  $$t_j - [\tau_{duty}^{min} - (\tau_{duty}^{max} - remDuty_j)]^+ - nRests_j \times \tau_{rest}^{min} - (nRests_j - 1) \times \tau_{duty}^{min}.$$

  Moreover, the updating of *slack* at line 8 of Algorithm 3 has to change to

  $$slack \leftarrow min(l_j - e_j, slack_j + remDuty_j - [remDuty_j - (\tau_{duty}^{max} - \tau_{duty}^{min})]^+)$$

  $$+(nRests_j - 1) \times (\tau_{duty}^{max} - \tau_{duty}^{min})$$

and at line 13 of Algorithm 3 to

$$slack \leftarrow min(l_j - e_j, slack_j + remDuty_j - [remDuty_j - (\tau^{max}_{duty} - \tau^{min}_{duty})]^+$$
$$+(nRests_j - 1) \times (\tau^{max}_{duty} - \tau^{min}_{duty}) + nRests \times (\tau^{max}_{rest} - \tau^{min}_{rest})$$
$$-(t_j - [\tau^{min}_{duty} - (\tau^{max}_{duty} - remDuty_j)]^+ - nRests_j \times \tau^{min}_{rest} - (nRests_j - 1) \times \tau^{min}_{duty} - l_j)).$$

Notice that the introduction of a minimum duty time has the opposite effect compared to the introduction of a variable rest time, i.e., it reduces the flexibility. Flexibility is reduced since a minimum amount of time (equal to the minimum duty time) has to take place between two consecutive rests. This reduction in flexibility has to be taken into account when updating *slack* and when deciding on when to take a rest (a minimum amount of time has to have elapsed since the last rest). The modifications listed above take into account this decrease in the flexibility of constructing a feasible path. No other changes are needed to Algorithm 2 and Algorithm 3 since these are the only parts where the introduction of a minimum duty time has an influence. Thus, with these changes Theorem 1, Theorem 2, and Theorem 3 still hold.

5 Service Times

In this section, we consider the simpler variant of the trip scheduling problem in which there are no dispatch windows, but there is a service time at the locations. The analysis of this simple case is justified by its practical relevance.

In this setting, a service time s_i is incurred, when the truck visits a location i. The driver has to decide whether to wait for the service to be completed and then continue, or whether to commence his rest as soon he arrives and continue only after resting. It is assumed that the service times are less than the rest time and that service can be performed while the driver is resting. We want to find a time-feasible dispatch schedule of minimum length that accounts for service time at locations and respects the HOS regulations.

We propose a forward dynamic programming algorithm. Let d_i denote the departure time at location i (after service has taken place). The aim of the algorithm is to minimize d_{n+1}.

A fresh driver starts at location 0. It is easy to see that, while driving from one location to another, it is optimal to always rest as late as possible in order to arrive as early as possible at the next location. (Recall that there are no dispatch windows.) A decision has to be made upon arrival. If the driver decides to wait and leave as soon as the service is completed, then the departure time is equal to the arrival time plus the service time, but remaining duty time at the departure time and the remaining drive time at the departure time depend on the decisions at earlier locations. If instead, the driver decides to rest as

soon as he arrives, then the departure time is equal to the arrival time plus the rest time, and the remaining duty time is τ_{duty} and the remaining drive time is τ_{drive}. Consequently, in the latter case, only the path that arrives as early as possible has to be considered.

Let p_i be the number of different paths constructed by the dynamic program to reach i. Then p_{i+1} is $p_i + 1$. Each of the existing p_i paths has to be extended for the decision to wait and leave immediately after service has been completed, and one new path has to be constructed for the decision to rest as soon as the driver arrives, choosing among the existing paths, the one that minimizes d_i. A more formal description of the steps taken at location i is presented in Algorithm 4, where d_i^j, $remDuty^j$, and $remDrive^j$ are, respectively, the departure time, the remaining duty time, and the remaining drive time for path j.

Since $p_1 = 0$, we have $p_{n+1} = n$. Among the n final paths, we choose the one that minimizes d_{n+1}. The algorithm is clearly polynomial.

Suppose now that s_i is greater than τ_{rest} for at least one location i'. Then, upon arrival at i', a rest has to be made independently of the path considered. For each path, the rest is made at the most suitable time, i.e., the time which maximizes the values of $remDuty$ and $remDrive$ at the departure from i'. If $remDuty + \tau_{rest} \leq s_i$, then the rest is made as late as possible. Otherwise, the driver waits for $s_i - \tau_{rest}$ and then goes to rest. No new paths have to be created for location i' because each path already includes a rest at i'.

6 Final Remarks

Hours of service regulations have a significant impact on the feasibility of truckload transportation trips, especially when long distances have to be traveled and trips last several days. Therefore, carriers have to be concerned with these regulation when they plan trips for their drivers. We have developed technology that can (and should) be part of this planning process.

References

[1] A. Campbell and M.W.P. Savelsbergh. "Delivery Volume Optimization." *Transportation Science 38,* 210-223, 2004.

[2] Federal Motor Carrier Safety Administration. *"Hours-Of-Service Regulations"* http://www.fmcsa.dot.gov/rules-regulations/topics/hos/hos-2005.htm

[3] B. Hunsaker and M.W.P. Savelsbergh. "Efficient Feasibility Testing for Dial-a-Ride Problems." *Operations Research Letters 30,* 169-173, 2002.

[4] H. Xu, Z.-L. Chen, S. Rajagopal, and S. Arunapuram. "Solving a Practical Pickup and Delivery Problem." *Transportation Science 37,* 347-356, 2003.

[5] W.B. Powell. "A Stochastic Model of the Dynamic Vehicle Allocation Problem," *Transportation Science 20,* 117-129, 1986.

[6] W.B. Powell. "An Operational Planning Model for the Dynamic Vehicle Allocation Problem with Uncertain Demands," *Transportation Research 21B,* 217-232, 1987.

[7] W.B. Powell, A. Marar, J. Gelfand, S. Bowers. "Implementing Real-Time Optimization Models: A Case Application From The Motor Carrier Industry." *Operations Research 50,* 571-581, 2002.

[8] J. Yang, P. Jaillet, and H. Mahmassani. "Real-Time Multi-vehicle Truckload Pickup and Delivery Problems." *Transportation Science 38,* 135-148, 2004.

[9] A. Regan, H. Mahmassani, and P. Jaillet. "Evaluation of Dynamic Fleet Management Systems." *Transportation Research Record 1645,* 176-184, 1998.

[10] W.B. Powell, Y. Sheffi, K. Nickerson, K. Butterbaugh, and S. Atherton. "Maximizing Profits for North American Van Lines' Truckload Division: A New Framework for Pricing and Operations," *Interfaces 18,* 21-41, 1988.

Algorithm 2 *BackwardSearch*(j, k)

for $i = j, j-1, \ldots, k$ **do**
  ***** *Phase I: Compute arrival at customer* *****
  if $t_{i,i+1} \leq remDrive$ **then**
    $t \leftarrow t - t_{i,i+1}$, $remDrive \leftarrow remDrive - t_{i,i+1}$, $remDuty \leftarrow remDuty - t_{i,i+1}$
  else
    $nRests \leftarrow 1 + \lfloor \frac{t_{i,i+1} - remDrive}{\tau_{drive}} \rfloor$
    $t \leftarrow t - (t_{i,i+1} + nRests \times \tau_{rest})$
    $slack \leftarrow slack + (remDuty - remDrive) + (nRests - 1) \times (\tau_{duty} - \tau_{drive})$
    $remDrive \leftarrow \tau_{drive} - (t_{i,i+1} - remDrive) \mod \tau_{drive}$
    $remDuty \leftarrow \tau_{duty} - (t_{i,i+1} - remDrive) \mod \tau_{drive}$
  end if
  ***** *Phase II: Analyze arrival at customer* *****
  if $t < e_i$ **then**
    return i // Dispatch window violation at location i
  else
    if $e_i \leq t \leq l_i$ **then**
      $slack \leftarrow \min(slack, t - e_i)$
    else
      $tBest \leftarrow t - (slack + remDuty)$
      if $tBest \leq l_i$ **then**
        ***** *Case a: No additional rest required* *****
        $wait_i \leftarrow t - l_i$
        if $t - l_i > slack$ **then**
          $t \leftarrow t - slack$, $slack \leftarrow 0$
          $remDuty \leftarrow remDuty - (t - l_i)$, $remDrive \leftarrow \min(remDrive, remDuty)$
        else
          $slack \leftarrow \min(slack - (t - l_i), l_i - e_i)$
        end if
        $t = l_i$
      else
        ***** *Case b: At least one additional rest required* *****
        $nRests \leftarrow \lceil \frac{tBest - l_i}{\tau_{rest} + \tau_{duty}} \rceil$
        $wait_i \leftarrow [t - l_i - (nRest \times \tau_{rest})]^+$
        if $tBest - (nRests \times \tau_{rest} + (nRests - 1) \times \tau duty) < l_i$ **then**
          if $t - \tau_{rest} < l_i$ **then**
            if $t - \tau_{rest} < e_i$ **then**
              return i // Dispatch window violation at location i
            else
              $t \leftarrow t - \tau_{rest}$
            end if
          else
            $t \leftarrow l_i$
          end if
          $slack \leftarrow min(t - e_i, [t - (tBest - (nRests \times \tau_{rest} + (nRests - 1) \times \tau_{duty}))])$
          $remDuty \leftarrow \tau_{duty}$, $remDrive \leftarrow \tau_{drive}$
        else
          $t \leftarrow l_i$, $slack \leftarrow 0$
          $remDuty \leftarrow \tau_{duty} - [(tBest - (nRests \times \tau_{rest} + (nRests - 1) \times \tau_{duty})) - t]$
          $remDrive \leftarrow min(remDuty, \tau_{drive})$
        end if
      end if
    end if
  end if
end for
return i

Algorithm 3 *Restore*(i)

$j \leftarrow i+1$ // i is the node with a dispatch window violation
while $j \leq n+1$ **do**
  if $wait_j > 0$ and $nRests_{ji} > 0$ **then**
    ***** *Move rests up to restore feasibility* *****
    $nRests_j \leftarrow nRests_j + 1$
    if $t_j - nRests_j \times \tau_{rest} \geq e_j$ **then**
      if $t_j - nRests_j \times \tau_{rest} \leq l_j$ **then**
        $remDuty \leftarrow \tau_{duty}$, $remDrive \leftarrow \tau_{drive}$
        $wait_j \leftarrow 0$
        $t \leftarrow t_j - nRests_j \times \tau_{rest}$
        $slack \leftarrow min(t - e_j, slack_j + remDuty_j + (nRests_j - 1) \times \tau_{duty})$
      else
        $slack \leftarrow min(l_j - e_j, slack_j + remDuty_j + (nRests_j - 1) \times \tau_{duty} - (t_j - nRests_j \times \tau_{rest} - l_j))$
        $remDuty \leftarrow \tau_{duty}$, $remDrive \leftarrow \tau_{drive}$
        $wait_j \leftarrow t_j - nRests_j \times \tau_{rest} - l_j$
        $t \leftarrow l_j$
      end if
      ***** *Check feasibility* *****
      $BackwardSearch(j-1, i)$
      if $t \geq e_i$ **then**
        return *Success*
      end if
    end if
  end if
  $j \leftarrow j+1$
end while
return *Failure*

Algorithm 4 Decision at location i

for $j = 1, 2, \ldots, |p_i|$ **do**
  if $t_{i-1,i} < remDrive$ **then**
    $d_i^j \leftarrow d_{i-1}^j + t_{i-1,i}$, $remDrive \leftarrow remDrive - t_{i-1,i}$, $remDuty \leftarrow remDuty - t_{i-1,i}$
  else
    $nRests \leftarrow 1 + \lfloor \frac{t_{i-1,i} - remDrive^j}{\tau_{drive}} \rfloor$
    $d_i^j \leftarrow d_{i-1}^j + (t_{i-1,i} + nRest \times \tau_{rest})$
    $remDrive^j \leftarrow \tau_{drive} - (t_{i-1,i} - remDrive) \mod \tau_{drive}$
    $remDuty \leftarrow \tau_{duty} - (t_{i-1,i} - remDrive) \mod \tau_{drive}$
  end if
  if $remDuty^j \geq s_i$ **then**
    $d_i^j \leftarrow d_i^j + s_i$
    $remDuty \leftarrow remDuty - s_i$
    $remDrive \leftarrow \min(remDuty, remDrive)$
  else
    // If $remDuty^j < s_i$ a rest has to be made
    // The path becomes equivalent to path $p_i + 1$
    // Thus it can be eliminated
  end if
end for
// Construction of path $p_i + 1$
$j* \leftarrow j : d_i^j = \min_{1,2,\ldots,|p_i|} d_j^i$
$d_i^{|p_i+1|} \leftarrow d_i^{j^*} + \tau_{rest}$
$remDrive^{|p_i+1|} \leftarrow \tau_{drive}$
$remDuty^{|p_i+1|} \leftarrow \tau_{duty}$