



Article

Allocating Small Transporters to Large Jobs

Neil Jami , Neele Leithäuser *  and Christian Weiß 

Fraunhofer Institute for Industrial Mathematics, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany; neil.jami@itwm.fraunhofer.de (N.J.); christian.weiss@itwm.fraunhofer.de (C.W.)

* Correspondence: neele.leithaeuser@itwm.fraunhofer.de; Tel.: +49-631-31600-4621

Abstract: We optimize the assignment of transporters to several jobs. Each job consists of processing a large, decomposable volume. A fleet of transporters is given, each of which can only process a limited volume at a time. After processing its share, a transporter must rest for a short time before being able to process another part. This time is only dependent on the assigned job, not on the transporter. Other transporters can take over the processing while a transporter rests. Transporters assigned to the same job wait for their turn in a queue. A transporter can only be assigned to one job. Our goal is to simultaneously minimize the maximum job completion time and the number of assigned transporters by computing the frontier of Pareto optimal solutions. In general, we show that it is NP-hard in the strong sense to compute even a single point on the Pareto frontier. We provide exact methods and heuristics to compute the Pareto frontier for the general problem and compare them computationally.

Keywords: transporter assignment; bi-objective optimization; heuristics; mixed-integer linear programming; logistics; heterogeneous fleet



Citation: Jami, N.; Leithäuser, N.; Weiß, C. Allocating Small Transporters to Large Jobs. *Algorithms* **2022**, *15*, 60. <https://doi.org/10.3390/a15020060>

Academic Editor: Pieter Smet

Received: 29 October 2021

Accepted: 7 February 2022

Published: 12 February 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In many farming or construction processes, transporter resources are used to carry material to or from the operation site in a cyclic fashion. Examples for such applications are tractors collecting silage from a forage harvester, dump trucks getting filled with excavated earth, or cement or asphalt being fed from trucks to construction sites. Transporters are available on site for a limited amount of processing time, dependent on the loading capacity of the transporter and the processing speed of the main resource.

After that, they have to unload/reload material in some central hub (e.g., a harvest silo, asphalt producer, etc.) before returning to the processing site. The time of absence depends on the distance between the operation site and the hub, but it is the same for all transporters assigned to the same site. Multiple transporters may be assigned to the same working site, so that processing can continue even when some transporters are on-road toward/from the central hub. If no other transporter is available on site, the main work has to pause until the next transporter arrives back on site. Two transporters may not be used for on-site processing at the same time.

This kind of transporter setting is referred to by Jensen et al. [1] as *capacitated field operations*. Many instances of these types of processes have been studied in the literature from different angles, with different goals, and for different applications, see [1–8]. Recent reviews are provided in [9,10]. The important common feature of all these use cases is that the total volume that needs to be processed on site is usually much larger than the capacity of a single transporter. Figure 1 illustrates such a process for the example of farming.

Since, usually, this type of problem is NP-hard (not solvable in polynomial time unless $P = NP$, see [11]) when viewed as a whole (see [5]), authors turn to mathematical programming formulations (see [2,5,12]) or (meta-)heuristics (see [4,5]).

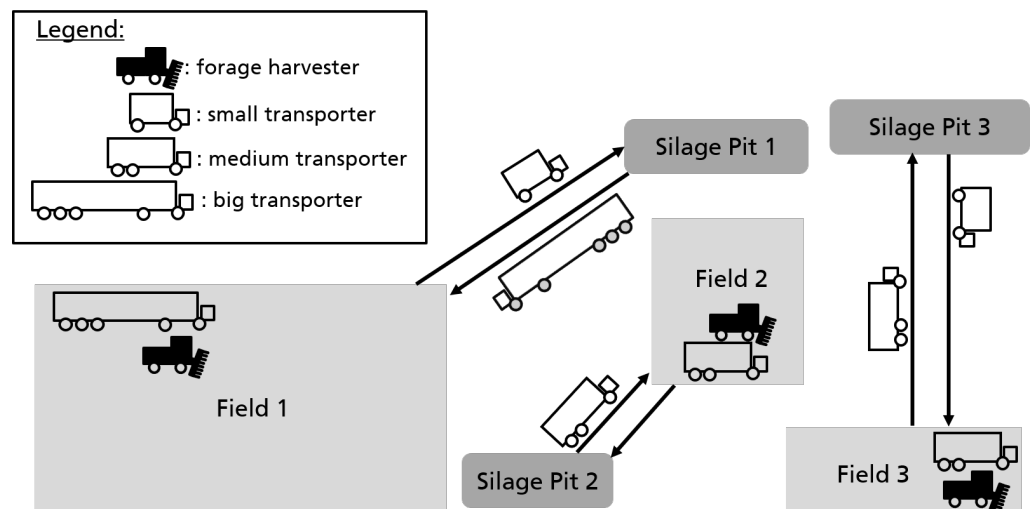


Figure 1. Example of three fields being harvested by forage harvesters and eight transporters of heterogeneous sizes. Field 1 is assigned an extra transporter due to its large size, making the avoidance of idle times even more important. Field 3 is assigned an extra transporter because it is far away from its silage pit, causing assigned transporters to waste a lot of time on the road.

In this paper, instead, we do not consider the problem as a whole but instead investigate one specific, important sub-problem, namely the assignment of transporters to operation sites. This problem gives rise to an interesting, new scheduling problem, which, to the best of our knowledge, has not been studied in the literature before.

1.1. Formal Problem Definition

For the purpose of this study, we assume that each operation site is assigned exactly one main resource. Otherwise, we split an operation site into multiple neighboring sites, one for each resource assigned. Furthermore, we assume that the processing speed of the main resources is fixed. This is reasonable for tactical planning if the tasks conducted at each site as well as the resources used to do so are similar.

Formally, a set of jobs $\mathcal{J} = \{j_1, \dots, j_{N_j}\}$ is given, each representing one operation site together with a main resource. Furthermore, let $\mathcal{R} = \{r_1, \dots, r_{N_R}\}$ be the set of transporters. In normal scheduling terminology, the transporters would be called resources or machines; however, we stay with the term transporters throughout this paper in order to stay closer to the terminology in our field of application.

Each job has a certain volume that needs to be processed before the job is complete, and each transporter has a certain capacity that can be used up before the transporter has to rest (travel to the central hub). Since we assume the processing speed of the main resources to be fixed, we can translate all volumes into time values for ease of notation. Thus, each job $j \in \mathcal{J}$ has a *saturated processing time* $C^{\text{sat}}(j) > 0$, which is the total time it needs to be processed by transporters before it is complete. Similarly, each transporter $r \in \mathcal{R}$ has a *filling time* $\kappa(r) > 0$, which is the time it can process a job before it needs to rest. Finally, we denote by $\delta(j) > 0$ the return time of every transporter assigned to job j . In this paper, we do not allow a transporter to be assigned to multiple jobs. Figure 2 shows how two transporters may process the same job.

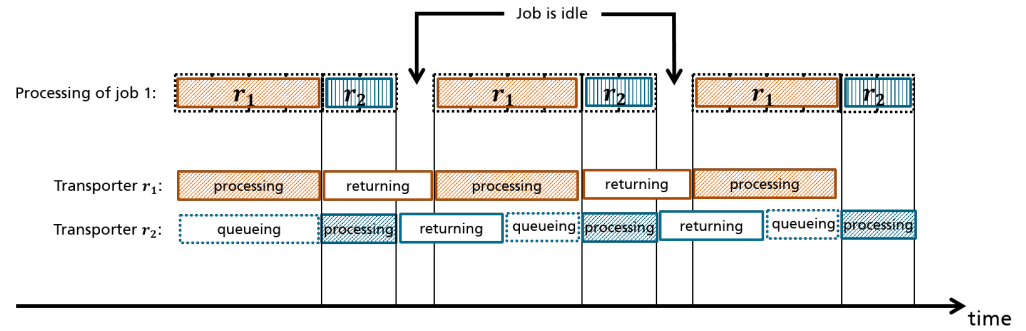


Figure 2. Example of two transporters relaying each other to process a job. The job processing stops while both transporters are in their return time.

We assume all jobs start processing at the same time. The completion time $C(j)$ of a job $j \in \mathcal{J}$ is given as the sum of $C^{\text{sat}}(j)$ and all idle times happening during the processing of job j due to no transporter being available. In particular, $C^{\text{sat}}(j)$ is the completion time of job j if job j is never idle (always has a transporter available for processing). Our goal is to minimize the maximum completion time of any job, which is also called the makespan $C^* = \max_{j \in \mathcal{J}} C(j)$. A short makespan is desirable, because it ties in directly with several crucial performance indicators, such as the necessary opening times of the hub, potential overtime pay for workers, as well as minimizing the time frame in which problems (e.g., due to weather) may happen. In addition, a large makespan may indicate unnecessary idle times for the main resources, which are to be avoided in order to operate cost efficiently.

However, it may not be possible to hire enough transporters to avoid all idle times (due to transporter availability), or, if it is possible, it may still not be economically desirable (due to drivers' wages) in cases where an additional transporter would only decrease the idle time by a marginal value. Thus, the total number of assigned transporters is another important measure for the quality of a solution. Therefore, our goal in this paper is to jointly minimize the makespan of multiple simultaneously operated sites and minimize the number of used transporters. The problem of assigning transporters to jobs in order to minimize simultaneously the makespan and the number of assigned transporters is called the *transporter allocation problem* or (TAP) for short.

As the two objectives of (TAP) are conflicting, our goal is to find all *Pareto optimal* solutions. A feasible solution to (TAP) is Pareto optimal if every other feasible solution with smaller makespan uses a larger number of transporters, and every other feasible solution that uses a smaller number of transporters has a larger makespan. The set of all Pareto optimal solutions is called the *Pareto frontier*. We refer to [13] for a general introduction to multicriteria optimization.

In the Appendix A, we prove that given a set $R \subset \mathcal{R}$ of transporters assigned to some job $j \in \mathcal{J}$, the completion time of j can be estimated by

$$C(j) = C^{\text{sat}}(j) \frac{L(j)}{L(j) - I(j)}, \tag{1}$$

where

$$L(j) := \max \left\{ \sum_{r \in R} \kappa(r), \delta(j) + \max_{r \in R} \kappa(r) \right\}, \tag{2}$$

and

$$I(j) := L(j) - \sum_{r \in R} \kappa(r). \tag{3}$$

The estimate depends on the practical assumption, mentioned above, that jobs are very large compared to transporter capacities and return times. See Theorem A1 for details. We also call $L(j)$ the *period length* and $I(j)$ the *idle time per period* of job j , according to the statements of Lemma A2 in the Appendix A. For the remainder of this paper, we do not deal

with the exact scheduling and sequencing of transporters when processing jobs and instead only focus on the assignment of transporters to jobs using Equation (1) to estimate the completion times of the jobs. Therefore, a solution to (TAP) is fully given by an assignment of transporters to jobs, which are encoded by variables $\bar{X} = (\bar{X}_{r,j})$ in the following way:

$$\bar{X}_{r,j} = \begin{cases} 1 & \text{if transporter } r \text{ is assigned to job } j \\ 0 & \text{otherwise.} \end{cases}$$

For a solution \bar{X} , we denote by $\bar{Y}_j(\bar{X})$ the number of transporters assigned to j under assignment \bar{X} :

$$\bar{Y}_j(\bar{X}) = \sum_{r \in \mathcal{R}} \bar{X}_{r,j}.$$

In addition, by $\bar{Y}(\bar{X})$, we denote the total number of transporters assigned by assignment \bar{X} , i.e.,

$$\bar{Y}(\bar{X}) = \sum_{j \in \mathcal{J}} \bar{Y}_j(\bar{X}).$$

Furthermore, we denote by $C^*(\bar{X}), C(j, \bar{X}), L(j, \bar{X})$, and $I(j, \bar{X})$ the makespan, completion time of job $j \in \mathcal{J}$, its period length, and its idle time per period, under solution \bar{X} , respectively. If there is no confusion about the solution referred, we may instead leave out \bar{X} as before.

Note that all mathematical notation introduced in this section and throughout the remainder of the paper is also given again in a concise list in the Abbreviations section at the end of this paper.

Finally, a solution \bar{X} *saturates* a job $j \in \mathcal{J}$ if job j has no idle time under solution \bar{X} , that is if $L(j, \bar{X}) = \sum_{r \in \mathcal{R}} \bar{X}_{r,j} \kappa(r_i)$ and $I(j, \bar{X}) = 0$. If solution \bar{X} saturates all jobs $j \in \mathcal{J}$, then \bar{X} is called *saturating*. Recall that in this case, the completion time of every job j is at its minimum value $C^{\text{sat}}(j)$. Note that a saturating solution minimizes both the maximum and total completion time of all jobs. An instance of problem (TAP) is *saturated* if there exists at least one saturating assignment.

The remainder of this paper is structured as follows. To finish Section 1 below, we briefly summarize our results and contributions. Then, in Section 2, we present a review of the literature related to our problem. In Section 3, we state and prove some structural properties of (TAP), in particular that (TAP) is NP-hard. In Section 4, we consider the special case where transporters are identical. In Section 5, we discuss the restricted problem of finding saturated solutions, where adding a transporter cannot further decrease any completion time. Our single objective is to minimize the the number of transporters. In Section 6, we develop algorithms to compute a Pareto frontier for the general problem, using results from the saturated problem. We present simulation results in Section 7. Finally, in Section 8, we conclude our paper.

1.2. Our Results

As mentioned before, to the best of our knowledge, this is the first paper to study the problem (TAP) as presented here explicitly. We show that computing even a single point on the Pareto front for an instance of problem (TAP) is, in general, NP-hard in the strong sense. It remains NP-complete in the strong sense even if all jobs have the same return time and/or saturated processing time.

In terms of positive results, we show that the minimum number of transporters needed to saturate a job can be computed in $\mathcal{O}((N_R + N_J) \log[N_R])$ time for all jobs. Furthermore, if all transporters are identical, then the whole Pareto front to solve problem (TAP) can be computed in $\mathcal{O}((N_R + N_J) \log[N_J])$ time. In that case, a single point on the Pareto front can also be computed in $\mathcal{O}(N_J \cdot \log[\max_j \{C(j, 1)\}])$ time, where $C(j, 1)$ denotes the processing time of a job when exactly one transporter is assigned to it. This time complexity may in some cases be better as N_R is no longer part of the estimate. See also Table 1 for an overview of our complexity results.

Table 1. Overview of complexity results for problem (TAP). Note that $\delta(j) = \delta^*$ denotes the case where all return times are identical and $\kappa(r) = \kappa^*$ denotes the case where all transporter capacities are identical. Finally, $C(j, 1)$ denotes the completion time of job j when exactly one transporter is assigned to it, in the case where all transporters have identical capacity.

Version	Whole Pareto Front	Single Pareto Point	Reference
General	str. NP-hard	str. NP-hard	Section 3
$\delta(j) = \delta^*$	str. NP-hard	str. NP-hard	Section 3
$\kappa(r) = \kappa^*$	$\mathcal{O}((N_R + N_J) \log[N_J])$	$\mathcal{O}(N_J \cdot \log[\max_j\{C(j, 1)\}])$	Section 4

In addition to these complexity results for sub-problems and special versions, we also consider how to solve the general problem. We suggest several approaches using mathematical programming, heuristics, and combinations of both. We evaluate all suggested algorithms experimentally and discuss their advantages and disadvantages when compared to each other.

2. Literature Review

In this section, we outline how the transporter allocation problem studied in this paper ties in with the wider literature. As mentioned before, to the best of our knowledge, (TAP) has not been studied in this form before. However, there are several optimization problems that bear a resemblance to (TAP). Below, we focus on the two most closely related problems, namely divisible load scheduling, which can be seen as a version of (TAP) where transporters may process the job in parallel rather than in sequence, and multiple knapsack problems, which relate to (TAP) in the way that for each job, there is an upper limit of transporter sizes before adding more transporters no longer improves anything. We compare the problems to (TAP) and explain which, if any, results can be transferred to our setting.

We will not consider papers discussing capacitated field operations in general again, as in the beginning of Section 1, since other than providing the wider practical motivation, they have little relevance for (TAP). Furthermore, note that even though both problems stem from logistic tasks, (TAP) is not comparable to vehicle routing problems. In vehicle routing problems [14], each transporter is usually assigned several delivery jobs, while for (TAP), each job is assigned several transporters. Due to this fundamental difference, even though the problems may look related in practice, they are not close from a mathematical perspective.

2.1. Divisible Load Scheduling

In divisible load scheduling [15], a processing load needs to be divided onto several processors, just like the total volume of a job in (TAP) needs to be divided onto several transporters. The main difference between the two is that in divisible load scheduling, the processors are assumed to be parallel, such that they are allowed to process different parts of the main load at the same time. In addition, usually, in divisible load scheduling problems, only a single load needs to be split across processors, while for (TAP), multiple loads need to be assigned to transporters. Note that without any further assumptions, splitting a load over parallel processors is usually easy. Therefore, the literature considers many additional parameters, such as setup times, communication delays between processors, and more complex processor networks (e.g., where sub-processors on the first level have again several second-level sub-processors to which they can divide their sub-task). We refer to [16] for a review. In comparison, as we show in this paper, (TAP) is much harder, even without any additional assumptions.

Interestingly, many papers consider bi-criteria versions of divisible load scheduling, where the goal is to minimize the makespan and cost of a schedule simultaneously; see [17–19]. Here, cost is usually defined as a function of the load assigned to each processor, together with a fixed cost to activate each used processor in the first place [19].

Special versions have been studied without any additional restrictions and where only the fixed cost for each processor is used [17,19]. Note that this version is very close to (TAP) with only three differences: first, of course, for the divisible load scheduling problem, the processors work in parallel, whereas for (TAP), they in sequence; second, in the divisible load scheduling problem, processors may have different speeds but have no maximum load, whereas in (TAP), all transporters have the same speed but a maximum capacity after which they must rest; third, for (TAP), we assume additionally that the fixed cost for each transporter is the same. This special version of divisible load scheduling is shown to be NP-hard in the usual sense [17]. We prove later that (TAP) is NP-hard in the strong sense. Furthermore, note that if in the divisible load scheduling problem the fixed cost of all processors is the same, then the NP-hardness proof from [17] is no longer valid, while this assumption holds by definition for (TAP).

Most other results for divisible load scheduling are not comparable to (TAP) either due to additional restrictions for the divisible load scheduling problem or because of the difference in sharing load in parallel or sequential. Thus, techniques used to solve divisible load scheduling problems cannot be easily transferred to (TAP).

2.2. Multiple Knapsack Problems

Even though it is not immediately obvious from a practical point of view, (TAP) is rather strongly related to multiple knapsack problems. Recall that the knapsack problem [20] is a popular optimization problem, with the objective to select items with maximum total value without exceeding a certain capacity. The multiple knapsack problem considers several knapsacks with potentially different capacities. The objective is to assign items to knapsacks such that the total items value is maximized without exceeding any knapsack capacity. In comparison, for (TAP), consider each job as a knapsack with a capacity given by the total size of transporters needed before all idle times vanish (note that this capacity is not fixed, as it depends on the largest transporter assigned, cf. Equation (2)). Then, a good allocation of transporters is one where each “knapsack” is as close to exactly filled as possible.

Many knapsack-related problems are known to be NP-hard [11]. Most algorithms for the multiple knapsack problem are either greedy or meta-heuristics such as in [21,22], polynomial approximation schemes such as in [23,24], or exact methods using mathematical programming as in [25,26].

For our paper, a special kind of knapsack problem studied in [27] is of particular interest. In this version, so-called assignment restrictions are given, such that each item is only allowed to be assigned to a subset of the knapsacks. This is interesting, since in (TAP), jobs may become ineligible for further transporters once they have reached their minimum possible completion time. In [27], the authors propose several approximation algorithms, some with a simple, greedy structure. We will later (cf. Section 5) see that two of these algorithms can be adjusted to work well as heuristics for (TAP).

3. Structural Results

In this section, we consider the structural properties of (TAP) before turning to solution methods for the general problem and special cases in the remainder of the paper. In the first part of this section, we show that (TAP) is NP-hard in the strong sense. In the second part of this section, we introduce lower bounds on the number of transporters needed for a job to be saturated, i.e., have no idle time. We also provide an easy algorithm to compute all such lower bounds for a general instance of (TAP).

3.1. NP-Hardness of (TAP)

Unfortunately, (TAP) is NP-hard in the strong sense, i.e., not solvable in time polynomial in the size of the input, even if the input is encoded in unary encoding, unless $P = NP$, cf. [11]. In fact, we prove below that for an instance of (TAP) it is, in general, strongly NP-hard to compute even a single point on the Pareto frontier. Let $(TAP_{C^*}(n_r))$

be the decision version of (TAP) with target makespan C^* . That is, given an instance of (TAP) and a target makespan C^* , the task for $(TAP_{C^*}(n_r))$ is to decide if there exists an assignment of transporters to jobs such that the makespan is at most C^* , and at most, n_r transporters are used.

Theorem 1. $(TAP_{C^*}(n_r))$ is NP-complete in the strong sense, even if C^* is chosen as the minimum possible makespan, i.e., $C^* = \max_j\{C^{\text{sat}}(j)\}$. Moreover, it remains NP-complete in the strong sense, even if all jobs have the same return time and/or saturated processing time. Finally, it remains NP-complete in the strong sense even if it reduces to the decision if a saturating assignment exists, i.e., with parameters $C^* = C^{\text{sat}}(1) = C^{\text{sat}}(2) = \dots = C^{\text{sat}}(N_J)$ and $n_r = N_R$.

Proof. Reduction from three-partition problem, which is NP-complete in the strong sense [11]: Given a multiset S of $n = 3 \cdot m$ positive integers, can S be partitioned into m triplets S_1, S_2, \dots, S_m such that the sum of the numbers in each subset is equal?

The subsets S_1, S_2, \dots, S_m must form a partition of S in the sense that they are disjoint and they cover S . Let B denote the (desired) sum of each subset S_i , or equivalently, let the total sum of the numbers in S be $m \cdot B$. The three-partition problem remains strongly NP-complete when every integer in S is strictly between $\frac{B}{4}$ and $\frac{B}{2}$.

Let (I) be an arbitrary instance of three-partition with $S = s_1, \dots, s_{3 \cdot m}$ and $\sum_i s_i = m \cdot B$ and $B/4 < s_i < B/2$.

Let M be an arbitrary large positive integer with $M > m \cdot B$. Let S' be the multiset defined by $S + K$ with K being the multiset $\{M, \dots, M\}$ with $|K| = 2 \cdot m$. Hence, $|S'| = 5 \cdot m$. In the following, we refer to the elements from K as the *large* elements. Define now an instance of $(TAP_{C^*}(n_r))$ with the set of resources \mathcal{R} with filling times $\kappa(t_i) = s'_i$. The job set \mathcal{J} has cardinality m and the return times $\delta(j) = B + M$ and saturated processing times $C^{\text{sat}}(j) = B + M + M$ for all jobs $j \in \mathcal{J} := \{j_1, \dots, j_m\}$. Note that, by construction, all jobs have the same return time and the same saturated processing time.

We need to prove the following:

$$(I) \text{ has a solution} \Leftrightarrow (TAP_{B+M+M}(5 \cdot m)) \text{ has answer yes.}$$

\Rightarrow : Given a solution S_1, \dots, S_m (i.e., a set of triplets with the desired properties) to the three-partition instance (I), we can easily construct a solution to (TAP) by first assigning the transporters with the filling times corresponding to S_i to the job j_i . Then, each job is additionally assigned to two of the transporters of size M . Since by definition $\sum_{s \in S_i} s = B$, the return time of $B + M$ is covered exactly by the four smallest elements, meaning each job is saturated and no idle time occurs. By definition of the saturated processing times, this means the makespan of the constructed solution is exactly $B + M + M$.

\Leftarrow : Let $\bar{X}_{r,j}$ be a solution to (TAP) with makespan $B + M + M$ and $5 \cdot m$ resources used. By definition of our instance, all m jobs must be saturated in order to reach makespan $B + M + M$. Therefore, each job must have assigned transporters with an accumulated filling time of at least $B + M$ when ignoring the largest transporter. Since M by definition cannot be reached by using items of S , the largest and second largest transporter assigned to each job must each have filling time M . This uses up all $2m$ transporters with filling time M , and no job can have three transporters with filling time M assigned. Hence, the *large* items of size M are distributed equally to the jobs, leaving the items from S to cover the remaining return times of B for each job. Ignoring the *large* items, a solution for the three-partition instance (I) remains. \square

Note that the same arguments hold if the saturated processing time of each job is instead given by $k \cdot (B + M + M)$ for some $0 < k \in \mathbb{N}$. So, the same proof also works in a more practical instance, where jobs are very large compared to transporter capacities and return times.

Note also that nearly the same reduction can be made from a usual partition problem (cf. [11]) instead of three-partition. In that case only two jobs would be constructed and

four transporters of size M would be needed. Then, using the same arguments as above, it can be shown that a solution to the partition instance exists if and only if a solution to $(TAP_{C^*}(n_r))$ with $C^* = B + M + M$ exists, where B is the target value of the partition instance. This shows that problem (TAP) would remain NP-hard (in the usual sense) even if there are only two jobs.

3.2. Saturating Transporter Sets

Given a set of transporters \mathcal{R} and a job j , recall that a set of transporters $R \subset \mathcal{R}$ saturates job j if

$$\sum_{r \in R} \kappa(r) \geq \delta(j) + \max_{r \in \mathcal{R}} \kappa(j).$$

Any (cardinality) smallest subset of \mathcal{R} that can saturate job j is called a *saturating transporter set* for j . The common cardinality of all saturating transporter sets for job j is called the *saturating transporter cardinality*, which is denoted by $N_{\mathcal{R}}^*(j)$. If no saturating transporter set exists, that is, if even \mathcal{R} itself cannot saturate job j , then we define $N_{\mathcal{R}}^*(j) = \infty$. Note that, by definition, the $N_{\mathcal{R}}^*(j)$ transporters in \mathcal{R} with biggest filling times are a saturating transporter set for job j , if one exists.

The saturating transporter cardinality of every job is a lower bound of the number of assigned transporters to this job in any saturating assignment. Our algorithms use this concept to compute assignments.

Lemma 1. *The time complexity to compute the saturating transporter cardinality of all jobs is:*

$$\mathcal{O}((N_R + N_J) \cdot \log[N_R]).$$

Proof. We sort the transporters by decreasing the filling time in $\mathcal{O}(N_R \cdot \log[N_R])$. Then, we remove the biggest transporter, which will be the transporter whose return time has to be covered.

For a single job, we can build a saturating transporter set by adding to the biggest transporter as many transporters as necessary from a sorted list such that their filling time exceeds the job’s return time. Since this set contains the biggest transporters, there cannot be any smaller transporter set saturating the job.

We can improve this process to compute the saturating transporter cardinality for every job. We first compute the cumulative filling times of the transporters in $\mathcal{O}(N_R)$. Then, for each, we find the minimum cumulative filling time bigger than the return time, using a binary search in $\mathcal{O}(\log[N_R])$ time. Hence, for all jobs, it takes $\mathcal{O}(N_J \cdot \log[N_R])$.

Thus, the total complexity is $\mathcal{O}((N_R + N_J) \cdot \log[N_R])$. \square

4. The Homogeneous Transporter Allocation Problem

In this section, we consider the special case of (TAP) in which all transporters have the same filling time κ . We call instances of (TAP) that fulfill this condition *homogeneous*. Note that in this case, the saturating transporter cardinality $N_{\mathcal{R}}^*(j)$ of job j can be computed in $\mathcal{O}(1)$ by the equation:

$$N_{\mathcal{R}}^*(j) := \left\lceil \frac{\delta(j)}{\kappa} \right\rceil + 1. \tag{4}$$

That means that the saturating transporter cardinality of all jobs can be computed in $\mathcal{O}(N_J)$ instead of $\mathcal{O}((N_R + N_J) \cdot \log[N_R])$ as in the general case.

By definition, if we assign $\bar{Y}_j(\bar{X}) = \sum_r \bar{X}_{r,j} \geq N_{\mathcal{R}}^*(j)$ transporters to a job j , then the job j is saturated and its completion time is $C^{\text{sat}}(j)$. Otherwise, every period takes duration $\delta(j) + \kappa$ while the effective completion time is only $\bar{Y}_j \cdot \kappa$, and the job j is idle for the remaining $\delta(j) + \kappa - \bar{Y}_j \cdot \kappa$ time units per period.

Then, the job completion time is:

$$C(j, \bar{X}) = \begin{cases} C^{\text{sat}}(j) \cdot (\delta(j) + \kappa) / (\kappa \cdot \bar{Y}_j(\bar{X})) & \text{if } \bar{Y}_j < N_{\mathcal{R}}^*(j), \\ C^{\text{sat}}(j) & \text{if } \bar{Y}_j \geq N_{\mathcal{R}}^*(j). \end{cases} \quad (5)$$

In what follows, we show that for a homogeneous instance of (TAP), the full frontier of Pareto optimal solutions can be computed in $\mathcal{O}((N_R + N_J) \cdot \log[N_J])$ time.

Consider algorithm (HOM), which computes the Pareto frontier in the following steps:

1. The first Pareto solution is achieved by assigning one transporter to every job;
2. Add one more transporter to any job with maximum completion time according to Equation (5);
3. If step 2 decreased the makespan, then the new solution is a Pareto solution;
4. If any job with maximum completion time is saturated, or if all transporters are used, stop; otherwise, return to step 2.

The following theorem shows that algorithm (HOM) computes the frontier of Pareto optimal solutions in full.

Theorem 2. *If every transporter has the same filling time, Algorithm (HOM) computes a Pareto solution to (TAP) at every step in which the makespan decreases. Moreover, each Pareto solution is found by algorithm (HOM).*

Proof. We prove, equivalently, that at the k -th step (including the step where the first solution is computed), Algorithm (HOM) computes a solution of minimum makespan amongst all solutions in which exactly $N_J + k - 1$ transporters are used. That means that if the makespan decreases in the k -th step, the achieved solution is Pareto optimal, since any fewer number of used transporters can only achieve a strictly larger makespan. In addition, since all possible numbers of used transporters are tested once, no Pareto optimal solution is missed.

It remains to be shown that indeed, the solution computed in the k -th step has a minimum makespan amongst all solutions in which exactly $N_J + k - 1$ transporters are used. Suppose that at the k -th step, the assignment is not minimizing the makespan over every solution using k transporters. Let $(\bar{Y}_{j_1}, \dots, \bar{Y}_{j_{N_J}})$ be the number of transporters assigned to the jobs at this step. Consider an optimal assignment of k transporters $(\bar{Y}_{j_1}^*, \dots, \bar{Y}_{j_{N_J}}^*)$, minimizing the makespan. Pick the first iteration where the algorithm assigns a transporter to a job j_i such that the number $\bar{Y}_{j_i}^*$ of the optimal solution is being exceeded. By definition, the completion time of the optimal solution is greater or equal to the completion time of j_i when using $\bar{Y}_{j_i}^*$ transporters. However, by construction, at this point in time, the algorithm assignment is such that the completion time of every job is smaller than or equal to this duration. This contradicts the assumption that the maximum completion time with \bar{Y}^* is strictly lower than with \bar{X} , hence that \bar{Y}^* is optimal. \square

Additionally, Algorithm (HOM) has the runtime as claimed above.

Theorem 3. *Algorithm (HOM) runs in time $\mathcal{O}((N_R + N_J) \cdot \log[N_J])$.*

Proof. We first compute the partial assignment of one transporter per job in $\mathcal{O}(N_J)$. Then, for at most N_R iterations, we adjust the longest job and reorder the sorted list. Computing the completion time of a job takes $\mathcal{O}(1)$ time by Equation (5). If we store job completion times in a heap data structure in $\mathcal{O}(N_J \cdot \log[N_J])$, then every iteration is performed in $\mathcal{O}(\log[N_J])$. Thus, the total time complexity is $\mathcal{O}((N_R + N_J) \cdot \log[N_J])$. \square

Together, Theorems 2 and 3 prove our initial proposition.

Corollary 1. For a homogeneous instance of (TAP), the full frontier of Pareto optimal solutions can be computed in $\mathcal{O}((N_R + N_J) \cdot \log[N_J])$ time.

Note that in the homogeneous (TAP), the set of transporters can be encoded in $\log[N_R]$ space by simply encoding the number of transporters (all transporters are of the same type). Thus, technically, a runtime of $\mathcal{O}((N_R + N_J) \cdot \log[N_J])$ is not polynomial in the input length. On the other hand, for a generic instance, a Pareto frontier has $\mathcal{O}(N_R)$ points, so it cannot be put out in less than $\mathcal{O}(N_R)$ steps. In scheduling, this is usually referred to as a high-multiplicity problem, see [28].

One way to resolve this issue is to show that any single Pareto point can be computed in polynomial time (with an algorithm that is at most logarithmic in N_R). For our algorithm, note that any single iteration of step 2 runs in $\mathcal{O}(\log[N_J])$ time and that a new Pareto point is found at least every N_J such iterations (after each job is assigned one more transporter). Thus, our algorithm moves from one Pareto point to the next in polynomial time. Still, in order to compute a particular point, using at most m transporters, with Algorithm (HOM), we need $\mathcal{O}((m + N_J) \cdot \log[N_J])$ steps, which is not polynomial when $m = \mathcal{O}(N_R)$.

Instead, one can compute for a given makespan exactly the number of transporters needed to be assigned to each job via Equation (5) and then check if the total number of transporters does not exceed the given upper bound. For any given makespan, this check can be run in $\mathcal{O}(N_J)$ time, running one computation for each job. Then, via binary search using this computation at each step, one obtains a polynomial algorithm to compute a particular point on the Pareto frontier. To be precise, the obtained algorithm runs in $\mathcal{O}(N_J \cdot \log[\max_j \{C(j, 1)\}])$, where $C(j, 1)$ denotes the processing time of a job when exactly one transporter is assigned to it.

However, our primary goal is to compute the Pareto frontier as a whole, for which Algorithm (HOM) is better suited than iteratively using the suggested binary search algorithm, which would take $\mathcal{O}(N_R \cdot N_J \cdot \log[\max_j \{C(j, 1)\}])$ time.

5. The Saturated Transporter Allocation Problem

In this section, we consider a simplification of (TAP), which we call the *saturated transporter allocation problem* (STAP). The objective of (STAP) is to compute an optimal saturating assignment, that is with a minimum number of transporters and where jobs are never idle. If the transporter fleet is too small to allow such an assignment, then (STAP) is infeasible.

Note that this simplification removes the bi-criteria nature of the problem, as the only objective is to assign as few transporters as possible, under the constraint of saturating every job. In addition, note that the solution to problem (STAP) is not necessarily a point on the Pareto frontier of (TAP). Indeed, from a solution of (STAP), it may be possible to remove some transporters from shorter jobs without increasing the makespan of the solution (but leaving shorter jobs unsaturated as a consequence). Finally, note that (STAP) is NP-hard in the strong sense, by Theorem 1, where we showed that computing a point on the Pareto frontier is strongly NP-hard, even if it reduces to finding a saturating assignment.

We first provide two mixed integer programs to solve (STAP): one for the general case and one optimized for the case where the transporter fleet consists of several types of similar transporters, which is more prevalent in practical settings. Then, we introduce several greedy heuristics, which we will later extend also to the solution of general (TAP).

5.1. General MIP Formulation

The following mixed integer program (sMIP) describes the saturated problem (sTAP), depending on the assignment $\bar{X}_{r,j}$, the transporter filling times $\kappa(r)$, and the return times $\delta(j)$:

(sMIP) :

$$\min_{\bar{X}} \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{J}} \bar{X}_{r,j} \tag{6}$$

$$\text{s.t.} \sum_{j \in \mathcal{J}} \bar{X}_{r,j} \leq 1, \quad \forall r \in \mathcal{R} \tag{7}$$

$$\sum_{r' \in \mathcal{R}} [\bar{X}_{r',j} \cdot \kappa(r')] - \bar{X}_{r,j} \cdot \kappa(r) \geq \delta(j), \quad \forall r \in \mathcal{R}, \forall j \in \mathcal{J} \tag{8}$$

$$\bar{X}_{r,j} \in \{0, 1\}, \quad \forall r \in \mathcal{R}, \forall j \in \mathcal{J}. \tag{9}$$

Equation (8) represents the saturation constraints: the return time of each assigned transporter is covered by the total filling time of the remaining assigned transporters, hence preventing any idle time. Recall that, actually, it is sufficient to test this for the transporter with maximum capacity, leading to

$$\sum_{r \in \mathcal{R}} [\bar{X}_{r,j} \cdot \kappa(r)] - \max_{t \in \mathcal{R}} \bar{X}_{t,j} \cdot \kappa(t) \geq \delta(j), \quad \forall j \in \mathcal{J}.$$

However, since this is not a valid constraint for a MIP, Equation (8) is used instead.

5.2. Aggregated MIP Formulation

In the mentioned real-life applications, it is rarely the case that transporter sizes vary arbitrarily. Rather, some small number of different models or variants exist that come with different capacities. In that case, it is reasonable to exploit this by grouping transporters with the same filling time. In the following, we identify each filling time with a transporter model and rewrite (sMIP) after grouping the transporters by their model.

Suppose that the set of transporter models is denoted by \mathcal{M} , and for each $m \in \mathcal{M}$, there are $M(m)$ many transporters of model m . Furthermore, assume the filling time of transporter model m is denoted by $\kappa(m)$. Introduce variables $\bar{Y}_{m,j}$, denoting the number of transporters of model m assigned to job j . Additionally, the binary variable $\bar{X}_{m,j}$ is equal to 1 if and only if at least one transporter of model m is assigned to job j .

We denote by \mathbb{N}^+ the set of natural numbers, including 0. Then, the aggregated mixed integer program (sMIPg) is as follows:

(sMIPg) :

$$\min_{\bar{X}, \bar{Y}} \sum_{m \in \mathcal{M}} \sum_{j \in \mathcal{J}} \bar{Y}_{m,j} \tag{10}$$

$$\text{s.t.} \sum_{j \in \mathcal{J}} \bar{Y}_{m,j} \leq M(m), \quad \forall m \in \mathcal{M} \tag{11}$$

$$\bar{X}_{m,j} \leq \bar{Y}_{m,j}, \quad \forall m \in \mathcal{M}, \forall j \in \mathcal{J} \tag{12}$$

$$\bar{Y}_{m,j} \leq M(m) \cdot \bar{X}_{m,j}, \quad \forall m \in \mathcal{M}, \forall j \in \mathcal{J} \tag{13}$$

$$\sum_{m' \in \mathcal{M}} [\bar{Y}_{m',j} \cdot \kappa(m')] - \bar{X}_{m,j} \cdot \kappa(m) \geq \delta(j), \quad \forall m \in \mathcal{M}, \forall j \in \mathcal{J} \tag{14}$$

$$\bar{X}_{m,j} \in \{0, 1\}, \quad \forall m \in \mathcal{M}, \forall j \in \mathcal{J} \tag{15}$$

$$\bar{Y}_{m,j} \in \mathbb{N}^+, \quad \forall m \in \mathcal{M}, \forall j \in \mathcal{J}. \tag{16}$$

Note that the variables $\bar{X}_{m,j}$ are only used in the constraints from Equations (14). Since these constraints are more restrictive when $\bar{X}_{m,j} = 1$, the solver will by default set $\bar{X}_{m,j}$ to 0.

Therefore, we do not need to fix an upper bound for variables $\bar{X}_{m,j} = 1$. Hence, we could omit Equation (12), but we leave it for the sake of clarity.

This new program (sMIP g) has $2 \cdot N_M \cdot N_J$ variables and $N_M + 3 \cdot N_M \cdot N_J$ constraints, whereas (sMIP) has $N_R \cdot N_J$ variables and $N_R + N_R \cdot N_J$. Therefore, the size of the new program (sMIP g) is smaller than the size of (sMIP) as soon as there are at least three transporters per model.

After solving (sMIP g), we still need to arbitrarily assign the transporters of each model to the jobs, which can be done in $\mathcal{O}(N_R + N_J)$ time.

5.3. Greedy Algorithms

In this section, we develop three greedy algorithms for (sTAP), which are inspired by algorithms used for the multiple knapsack problem.

In order to illustrate the connection between (sTAP) and multiple knapsack problems, below, we use the wording of knapsack instances to formulate the goal of (sTAP). When solving a multiple knapsack problem, the goal is to pack items with given weight and value in knapsacks to maximize the total packed value without exceeding the weight capacity of any knapsack. Using the same wording, when solving the saturated transporter allocation problem, the goal is to pack items with given weight and constant value of 1 in knapsacks to minimize the total packed value while ensuring the following minimum capacity conditions: in every knapsack, the total packed item weights excluding the biggest one exceeds a minimum total weight.

In [27], the authors propose, among others, two heuristics to solve the multiple knapsack problem under additional assignment restrictions. Their first greedy heuristic consists of taking the items in non-increasing order and assigning them to the first eligible knapsack. Their second algorithm takes knapsacks in an arbitrary order and successively solves single knapsack problems using the still unassigned items.

Following these ideas, our first algorithm assigns transporters to the jobs one job after the other, whereas the second and third algorithms assign transporters one by one to the best suited job. These heuristics are later also extended to the general version of (TAP) in Section 6 and are computationally evaluated along with other algorithms and MIPs in Section 7.

5.3.1. Greedy Algorithm by Job

Our first greedy algorithm (sGJ) iterates through the jobs in order of non-increasing return times. It assigns exactly as many transporters as needed to one job at a time so that the job is saturated. If at any time during the algorithm the current job cannot be saturated, the algorithm fails.

We assume that transporters are indexed in order of non-increasing filling times. We start with transporter set $\mathcal{R}' := \mathcal{R}$. We iteratively assign to a job a saturating transporter set with respect to the current set \mathcal{R}' and update \mathcal{R}' thereafter by deleting the assigned transporters. Since the ideal transporter sets of different jobs may overlap, it cannot be guaranteed that every job j will be assigned to $N_{\mathcal{R}}^*(j)$ transporters.

At any iteration, for a current job j , we first compute the minimum number of transporters $N_{\mathcal{R}'}^*(j)$ needed to saturate j using the current remaining set of transporters \mathcal{R}' . Recall that by Theorem 3.2, this can be done efficiently. Then, we enumerate all subsets of \mathcal{R}' of size $N_{\mathcal{R}'}^*(j)$ and assign one to j such that j is saturated. If there are multiple subsets of \mathcal{R}' of size $N_{\mathcal{R}'}^*(j)$ which saturate j , then as a tie breaker, we choose the set that uses the smallest maximum indexed transporter, which is not used by all other sets. If there is still a tie, we repeat the above rule with only the still tied sets until only one candidate set is left.

For example, suppose that the current transporter set is $\mathcal{R}' = \{r_1, r_2, r_3, r_4, r_5\}$, and that our current job is j with $N_{\mathcal{R}'}^*(j) = 3$. Suppose there are four subsets of \mathcal{R}' of size 3 that saturate job j , namely $\{r_1, r_2, r_3\}$, $\{r_1, r_2, r_4\}$, $\{r_1, r_2, r_5\}$, and $\{r_1, r_3, r_4\}$. Recall that transporters are numbered in order of non-increasing filling times. All sets use transporter r_1 . After removing r_1 , the transporter with maximum filling time in the first three candidates

is r_2 , while the transporter with maximum filling time in the fourth candidate is r_3 . We pick the fourth candidate, since r_3 has a smaller or equal filling time compared to r_2 .

If in the above example, $\{r_1, r_3, r_5\}$ would have been a candidate as well, then in a second round of tie breaking, our algorithm would only consider sets $\{r_1, r_3, r_4\}$ and $\{r_1, r_3, r_5\}$. After removing common transporters, only r_4 remains in the first set, while only r_5 remains in the second, leading to us choosing the later set $\{r_1, r_3, r_5\}$.

Note that the runtime of algorithm (SGJ) is exponential in the number of transporters, since we enumerate over a set of subsets of the transporter set. However, since all subsets have the same cardinality and this cardinality is usually relatively small, in practice, (SGJ) still has a reasonable runtime, as is shown in the computational experiments in Section 7.

5.3.2. Greedy Algorithms by Transporter

Algorithms (SGRc) and (SGRi) assign transporters one by one in order of non-increasing filling times.

Given currently assigned transporters, Algorithm (SGRc) assigns the next transporter to the non-saturated job with maximum completion time, whereas Algorithm (SGRi) assigns it to the non-saturated job with maximum idle time per period.

If every transporter has been assigned but the solution is still not saturated, these algorithms do not find any saturating solution. However, as opposed to algorithm (SGJ), the found solution is still feasible for the unsaturated problem as long as the number of transporters is at least as large as the number of jobs, i.e., if the instance itself is feasible. Indeed, both algorithms (SGRc) and (SGRi) assign the first N_J transporters to different jobs.

Recall that by Equation (1):

$$C(j, \bar{X}) := C^{\text{sat}}(j) \cdot \frac{L(j, \bar{X})}{L(j, \bar{X}) - I(j, \bar{X})},$$

and by Equation (2):

$$L(j, \bar{X}) := \max \left\{ \sum_{r \in \mathcal{R}} \bar{X}_{r,j} \cdot \kappa(r), \delta(j) + \max_{r \in \mathcal{R}} [\bar{X}_{r,j} \cdot \kappa(r)] \right\}.$$

Therefore, at every step of Algorithms (SGRc) and (SGRi), we only need to save the current values of the total and of the maximum assigned transporter capacities in order to decide where to assign the next transporter. We deduce from Equation (2) whether a job is saturated or not.

Theorem 4. Algorithms (SGRc) and (SGRi) run in $O(N_R \cdot \log[N_R])$.

Proof. In both algorithms, we first sort the transporters by decreasing filling time in $O(N_R \cdot \log[N_R])$. Then, the two algorithms use a different strategy to select the best job to which the next transporter is assigned. For Algorithm (SGRi), the best job maximizes the idle time per period. For Algorithm (SGRc), the best job maximizes the completion time.

In either case, we sort the jobs in $O(N_J \cdot \log[N_J])$ using e.g., a heap map. Then, the algorithm extracts the best job in $O(\log[N_J])$ time, update its value along with its total and its maximum transporter filling times in $O(1)$ time, and re-insert it into the heap in $O(\log[N_J])$.

Since our problem requires $N_J \leq N_R$, the time complexity of the algorithm is $O(N_R \cdot \log[N_R])$. \square

6. The General Transporter Allocation Problem

In this section, we consider the general *transporter allocation problem* (TAP), where we allow jobs to have idle time.

In the first part of this section, we show how to exactly compute the frontier of Pareto optimal solutions for an instance of (TAP). In order to do this, we first consider how to

minimize the makespan for a given set of transporters. Then, the Pareto frontier can be computed by iteratively increasing the number of transporters we are allowed to assign.

Then, in the second part of this section, we introduce heuristics to approximate the Pareto frontier instead.

All algorithms developed in this section are evaluated in Section 7.

6.1. Computing the Pareto Frontier Exactly

Now, we consider the problem of finding a solution of minimum makespan for a given instance of (TAP). Note that if only a subset of the given transporter set of size k should be used instead, it is the same as computing the minimum makespan for a new instance of (TAP) using only the k transporters with largest filling time. Observe also that a solution of minimum makespan does not necessarily use every transporter, as short jobs do not require to be saturated.

We first introduce a mixed integer quadratic program for which it is easy to see that it indeed finds a solution with minimum makespan for a feasible instance of (TAP). Then, we show how the quadratic program can be solved by instead iteratively solving a mixed integer linear program.

6.1.1. A Mixed Integer Quadratic Program

We denote by \mathbb{R}^+ the set of real positive numbers, including 0. The following mixed integer non-linear program computes a solution to (TAP) minimizing the makespan, using the return time $\delta(j)$, the saturated job processing time $C^{\text{sat}}(j)$, the period length \bar{L}_j , and the idle time per period \bar{I}_j . Note that \bar{L} and \bar{I} in this case are used as variables, instead of the predefined values $L(j, \bar{X})$ and $I(j, \bar{X})$, as introduced in Section 1.

(QTAP) :

$$\min_{\bar{X}, C^*, \bar{L}, \bar{I}} C^* \tag{17}$$

$$\text{s.t. } \sum_{j \in \mathcal{J}} \bar{X}_{r,j} \leq 1, \quad \forall r \in \mathcal{R} \tag{18}$$

$$\sum_{r \in \mathcal{R}} \bar{X}_{r,j} \geq 1, \quad \forall j \in \mathcal{J} \tag{19}$$

$$C^* \geq C^{\text{sat}}(j) \cdot \frac{\bar{L}_j}{\bar{L}_j - \bar{I}_j} \quad \forall j \in \mathcal{J} \tag{20}$$

$$\bar{L}_j = \bar{I}_j + \sum_{r \in \mathcal{R}} \bar{X}_{r,j} \cdot \kappa(r) \quad \forall j \in \mathcal{J} \tag{21}$$

$$\bar{L}_j \geq \bar{X}_{r,j} \cdot \kappa(r) + \delta(j) \quad \forall j \in \mathcal{J}, \forall r \in \mathcal{R} \tag{22}$$

$$\bar{X}_{r,j} \in \{0, 1\}, \quad \forall j \in \mathcal{J}, \forall r \in \mathcal{R} \tag{23}$$

$$\bar{L}_j, \bar{I}_j \in \mathbb{R}^+, \quad \forall j \in \mathcal{J} \tag{24}$$

Note that because of the terms $\frac{\bar{L}_j}{\bar{L}_j - \bar{I}_j}$ in Equation (20), the above program is indeed not linear. Namely, we get a quadratic program by multiplying both sides of the equation by $\bar{L}_j - \bar{I}_j$. In the next subsection, we will propose an alternative algorithm to solve (TAP) with the means of a slightly modified linear formulation.

Equation (19) ensures that $\bar{L}_j - \bar{I}_j > 0$ (i.e., the idle time is finite); hence, Equation (20) does not induce a division by zero. Equation (20) computes the job completion time.

Equations (21) and (22) model the period length along with the idle time per period. For a given assignment \bar{X} , the value of $\bar{L}_j - \bar{I}_j$ is fixed by Equation (21). Since we minimize the makespan, the program will tend to minimize \bar{L}_j by Equation (20) while keeping it above $\delta(j) + \max \{ \kappa(r) : \bar{X}_{r,j} = 1 \}$ by Equation (22). However, in this mathematical program, \bar{L}_j and \bar{I}_j do not exactly model the period length and idle time per period, as defined in Equations (2) and (3). Indeed, a solution to (QTAP) may contain additional, unnecessary

idle times for some jobs, and thus, the period length of those jobs may be artificially increased as long as the resulting completion time stays shorter than the completion time of the longest job. This problem could be fixed, though, by minimizing, for example, the average completion time as an additional objective.

6.1.2. Solving the Mixed Integer Quadratic Program

Instead of directly computing the optimal completion time of C^* , we iteratively look for a solution with a given completion time. We denote the maximum saturated job processing time by:

$$C_{\max}^{\text{sat}} := \max_j \{C^{\text{sat}}(j)\}. \tag{25}$$

The makespan C^* of any solution to (TAP) takes at least this duration:

$$C^* \geq C_{\max}^{\text{sat}}. \tag{26}$$

We define the *rate of idle time per period* $\rho(j, C)$ as the rate of idle time to have at every period for job j so that its completion time is C :

$$\rho(j, C) := 1 - \frac{C^{\text{sat}}(j)}{C} < 1. \tag{27}$$

Hence, we fix the idle time per period as:

$$\bar{I}_j = \rho(j, C) \cdot \bar{L}_j. \tag{28}$$

The following mixed integer program (ρ MIP) minimizes the number of assigned transporters under the same constraints as in the above mathematical program after replacing the idle time variables and the job completion time variables:

(ρ MIP) :

$$\min_{\bar{X}} \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{J}} \bar{X}_{r,j} \tag{29}$$

$$\text{s.t. } \sum_{j \in \mathcal{J}} \bar{X}_{r,j} \leq 1, \quad \forall r \in \mathcal{R} \tag{30}$$

$$\frac{\sum_{r'} [\bar{X}_{j,r'} \cdot \kappa(r')]}{1 - \rho(j, C)} - \bar{X}_{r,j} \cdot \kappa(r) \geq \delta(j) \quad \forall j \in \mathcal{J}, \forall r \in \mathcal{R} \tag{31}$$

$$\bar{X}_{r,j} \in \{0, 1\}, \quad \forall j \in \mathcal{J}, \forall r \in \mathcal{R}. \tag{32}$$

Since $\rho(j, C) < 1$, it follows that $\bar{I}_j < \bar{L}_j$ and hence Equation (19) is not needed anymore. Equation (31) is the combination of Equations (21) and (22) from the previous mathematical program along with the idle time per period formula of Equation (28). Note that it differs from the saturation condition of Equation (8) only by the factor $1 / (1 - \rho(j, C))$.

Lemma 2. *For any solution of (ρ MIP), the completion time of every job is C^* .*

Proof. Consider a solution \bar{X} of (ρ MIP). For every job j , the idle time per period is:

$$\begin{aligned} \bar{I}_j &= \rho(j, C) \cdot \bar{L}_j \\ &= \left(1 - \frac{C^{\text{sat}}(j)}{C^*}\right) \cdot \bar{L}_j \\ &= \bar{L}_j - \bar{L}_j \cdot \frac{C^{\text{sat}}(j)}{C^*}. \end{aligned}$$

Using Equation (1), the job completion time $C(j)$ is then:

$$\begin{aligned} C(j) &= C^{\text{sat}}(j) \cdot \frac{\bar{L}_j}{\bar{L}_j - \bar{I}_j} \\ &= C^{\text{sat}}(j) \cdot \bar{L}_j \cdot \frac{C^*}{\bar{L}_j \cdot C^{\text{sat}}(j)} \\ &= C^*. \end{aligned}$$

□

Now, we show that for a given target makespan C^* , a solution to (ρMIP) is a solution to (TAP) , i.e., an element of the Pareto frontier. Furthermore, ignoring artificial idle times, a solution to (TAP) can be found by solving (ρMIP) with an appropriate target makespan C^* .

Theorem 5. *For a given target makespan C^* , any solution of (ρMIP) is a solution of (TAP) with makespan at most C^* . Conversely, any solution of (TAP) with makespan C^* can be transformed into a solution of (ρMIP) with target makespan C^* .*

Proof. Algorithm (ρMIP) has been constructed from (QTAP) by fixing the variables \bar{I} and \bar{L} for a given target makespan C^* . Therefore, any solution to (ρMIP) is a solution to (QTAP) and therefore to (TAP) . Moreover, by definition of (ρMIP) , the makespan of the any solution is at most C^* .

On the other hand, given a solution \bar{X} of (TAP) , we can increase the completion time of every job up to the makespan C^* . After adding such artificial idle times, note that the new solution \bar{X}' fulfills Equation (28) for all jobs. Therefore, it is also a solution of (ρMIP) with target makespan C^* . □

Theorem 5 implies that in order to find a solution minimizing the makespan C_{\min}^* for (QTAP) and therefore (TAP) , we can instead run (ρMIP) for different values of C^* using a binary search. The following lemma limits the search interval for C_{\min}^* .

Lemma 3. *The minimum makespan C_{\min}^* is between C_{\max}^{sat} and the minimum makespan of any solution to the homogeneous transporter fleet problem where every transporter has filling time $\min_r \{\kappa(r)\}$.*

Proof. By definition, the completion time of a job $j \in \mathcal{J}$ cannot be smaller than its saturated processing time $C^{\text{sat}}(j)$. On the other hand, given an assignment for the homogeneous transporter fleet problem, then increasing the values for $\kappa(j)$ while keeping the assignment the same does not increase the completion time of any job. □

Hence, we can use a binary search between these values to approximate an optimal solution. The following Lemma is straightforward.

Lemma 4. *Consider $\epsilon > 0$, C_1 and $C_2 \in [C_1, C_1 + \epsilon]$ such that (ρMIP) has a solution for $C^* = C_2$ and no solution for $C^* = C_1$. Then, the solution for $C^* = C_2$ has a maximum job completion time at most ϵ away from the optimal solution of (TAP) . Moreover, it minimizes the number of transporters used among the solutions with a makespan of C_2 .*

We denote by (BS) the above described binary search algorithm framework, which requires an inner algorithm to solve (ρMIP) . Note that we can aggregate (ρMIP) by a transporter model similarly to (SMIPg) .

6.1.3. Pareto Frontier

Now, instead of computing a single solution, we compute the Pareto frontier of a (TAP) instance. By construction, the assignment computed by Algorithm (BS) has

minimum makespan C_{min}^* and uses as few transporters as possible. The Pareto frontier of the assignments is as depicted in Figure 3.

The transporter count in a Pareto optimal solution is at least equal to the number of jobs and at most equal to the transporter count computed by Algorithm (BS).

Our main algorithm (BS.MIP g) in this paper computes, for each transporter count, a solution with the minimum makespan using this number of transporters. This can be done with one binary search (BS) per transporter count. At every iteration, we solve an aggregated version of (ρ MIP) using an MIP solver. Note that when computing a Pareto solution for a given count of every transporter, we choose the transporters with the biggest capacity.

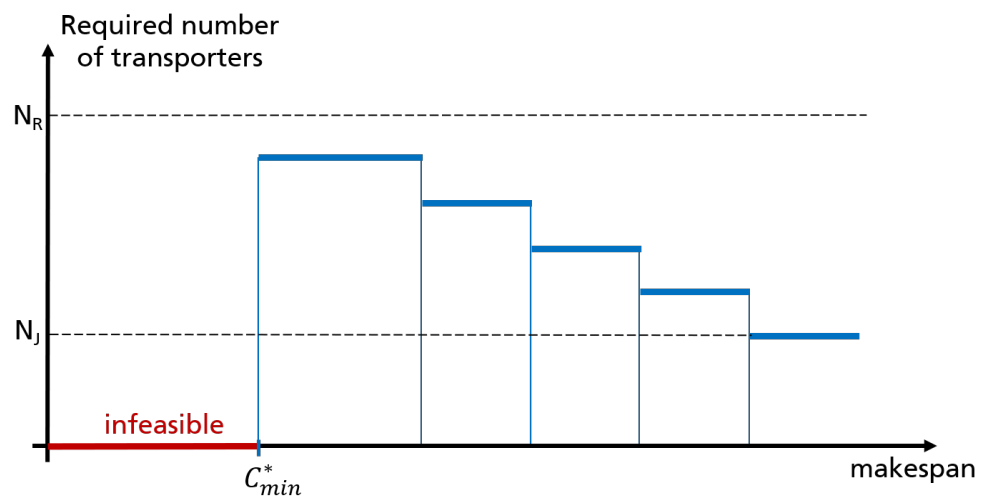


Figure 3. Minimum number of transporters in an assignment with a given makespan. A feasible solution needs at least one transporter per job. The solution minimizing the makespan does not necessarily require all N_R transporters.

6.2. Heuristics to Compute the Pareto Frontier

Now, we introduce several different heuristic algorithms to approximate the Pareto frontier of a (TAP) instance.

First, we use simple extensions of the heuristics for the saturated problem to compute the Pareto frontier directly. Then, we show how to achieve alternative heuristics by combining the heuristics for the saturated problem directly with ideas from the first part to approximate solutions of minimum makespan for the general (TAP), given a fixed set of transporters. Then, as in the first part, an approximation of the Pareto frontier can be gained by iteratively increasing the number of transporters we are allowed to assign.

6.2.1. Extensions of the Saturated Algorithms

Now, we extend the saturated greedy algorithms (sGJ), (sGRi), and (sGRc) to compute a Pareto frontier for (TAP). Note that every solution of the Pareto frontier must assign at least one transporter to each job, as otherwise, the makespan is infinite.

It is straightforward to extend the *greedy algorithms by transporter* (sGRi) and (sGRc) to compute a Pareto frontier, since every iteration already provides a partial assignment using one more transporter. At every iteration, we generate a completed assignment by temporarily providing one transporter to each job that had not received any transporter yet. If we do not have enough transporters to complete the assignment, the algorithm stops and returns the computed Pareto frontier. We denote by (GRi) and (GRc) the extension of the greedy heuristics to the general case.

For the *greedy algorithm by job* (sGJ), there is no simple way of creating a Pareto frontier. Our extension (GJ) of algorithm (sGJ) iteratively runs with different transporter counts, always choosing the ones with biggest capacity. For a fixed transporter count, when assign-

ing transporters to a job, (GJ) must restrict the maximum number of transporters assigned to this job such that the next jobs can still receive at least one transporter. Since Algorithm (GJ) allows jobs to not be saturated, it becomes more important to order the jobs in a meaningful sequence before assigning the transporters. We sort the jobs by decreasing values of completion time they would have if they were only assigned the smallest transporter. If we denote by κ_{\min} the minimum transporter filling time, by Equation (1), that means jobs are sorted by non-increasing value of $C^{\text{sat}}(j) \cdot \frac{\delta(j) + \kappa_{\min}}{\kappa_{\min}}$, depending on the job return time $\delta(j)$. In our simulations, we will only consider jobs with the same saturated processing time. In this case, our job order is equivalent to ordering the jobs by decreasing return times.

6.2.2. Binary Search Heuristics

In the following, we develop new heuristics by replacing the resolution of program (ρ MIP) with greedy heuristics in the binary search framework from (BS.MIPg).

The program (ρ MIP) consists of minimizing the number of used transporters for a fixed makespan. Hence, Algorithm (BS.MIPg) computes a solution, minimizing the number of used transporters for different makespan values. This is exactly what Algorithm (GRc) does. Therefore, combining this algorithm with a binary search will not yield any better result.

On the other hand, Algorithms (SGJ) and (GRi) focus on the idle time per period. The former looks for a combination of transporters saturating a job, while the latter aims at minimizing the maximum idle time per period. In the following, we slightly modify program (ρ MIP) to look like an instance to (sTAP) and use our greedy algorithms.

The program (ρ MIP) only differs from a saturated transporter allocation program by the factor $1/(1 - \rho(j, C))$ in Equation (31). Namely, for every job j :

$$\frac{1}{1 - \rho(j, C)} \cdot \sum_r [\bar{X}_{r,j} \cdot \kappa(r)] - \max_r [\bar{X}_{r,j} \cdot \kappa(r)] \geq \delta(j).$$

We rewrite Equations (31) as:

$$\begin{aligned} \sum_r [\bar{X}_{r,j} \cdot \kappa(r)] - [1 - \rho(j, C)] \cdot \max_r [\bar{X}_{r,j} \cdot \kappa(r)] &\geq [1 - \rho(j, C)] \cdot \delta(j) \\ \sum_r [\bar{X}_{r,j} \cdot \kappa(r)] - \max_r [\bar{X}_{r,j} \cdot \kappa(r)] &\geq [1 - \rho(j, C)] \cdot \delta(j) \\ &\quad - \rho(j, C) \cdot \max_r [\bar{X}_{r,j} \cdot \kappa(r)]. \end{aligned}$$

Since the maximum capacity assigned to a job is bounded by the maximum capacity over all transporters κ_{\max} , the following constraint is more restrictive:

$$\sum_r [\bar{X}_{r,j} \cdot \kappa(r)] - \max_r [\bar{X}_{r,j} \cdot \kappa(r)] \geq [1 - \rho(j, C)] \cdot \delta(j) - \rho(j, C) \cdot \kappa_{\max}. \tag{33}$$

This more restrictive constraint corresponds to a modified instance of (sTAP), with a return time of $[1 - \rho(j, C)] \cdot \delta(j) - \rho(j, C) \cdot \kappa_{\max}$ instead of $\delta(j)$ for job j . We call (r MIP) the mixed integer program obtained by replacing Equation (31) with Equations (33), modeling a saturated transporter assignment problem.

We can use this modified framework with any of our greedy heuristics to approximate the Pareto frontier. We denote by (BS.GJ) and (BS.GRi) the binary search-based heuristics using Algorithms (SGJ) and (SGRi) respectively to solve (r MIP).

7. Experiments

In this section, we provide two series of experiments. Firstly, we analyze the computation time and the efficiency of the saturated algorithms, which are the core of our algorithms. Afterward, we take a look at the Pareto frontiers computed for the general problem.

7.1. Simulation Environment

The simulations are done on a ThinkPad T490s with a processor Intel Core i7-8665U with 1.90GHz. Algorithms are implemented under Windows 10 in C#, and we use the open-source library *Google-OR-Tools* to solve mixed integer programs.

For the sake of simplicity, we restrict our simulations to a limited case study. We evaluate the performance of the algorithms on 100 feasible instances with a limited transporter fleet, using the following scenarios:

- 4 jobs, 20 transporters;
- 5 jobs, 25 transporters;
- 6 jobs, 30 transporters;
- 7 jobs, 35 transporters.

We randomly generate 100 problem instances for each scenario. The number of jobs, number of transporters, as well as the job return times and the transporter filling times are all integers following an uniform distribution. The return time of every job is between 5 and 15, and the filling time of every transporter is between 1 and 5. Therefore, every job will need on average $1 + \lceil \frac{5+15}{1+5} \rceil = 5$ transporters. Every job has the same saturated processing time of 180.

We are interested in three criteria:

1. The average computation time of the algorithm.
2. The number of infeasible instances, where the algorithm did not find an assignment.
3. The number of assigned transporters. Whenever the model is infeasible, we set the number of assigned transporters equal to the total number of transporters. Thus, we choose for this criterion to not penalize infeasible instances compared to a feasible assignment effectively using every transporter at our disposal.

Each of these criteria is sought to be minimized: a good performing algorithm must have low values for these criteria. Note that for saturated algorithms, the completion time of every job is fixed; hence, it does not need to be evaluated.

7.2. Performance of the Saturated Algorithms

We consider the following saturating algorithms:

- Algorithm (SGJ), assigning transporters to jobs one by one.
- Algorithms (SGR_i) and (SGR_c), iteratively assigning transporters to the job with maximum idle time per period and maximum completion time, respectively.
- Algorithm (SMIP_g), the mixed integer program grouping transporters with equal filling time.

We excluded the simple mixed integer program (SMIP), as it turned out that it is already too slow for our problem instances (see Figure 4).

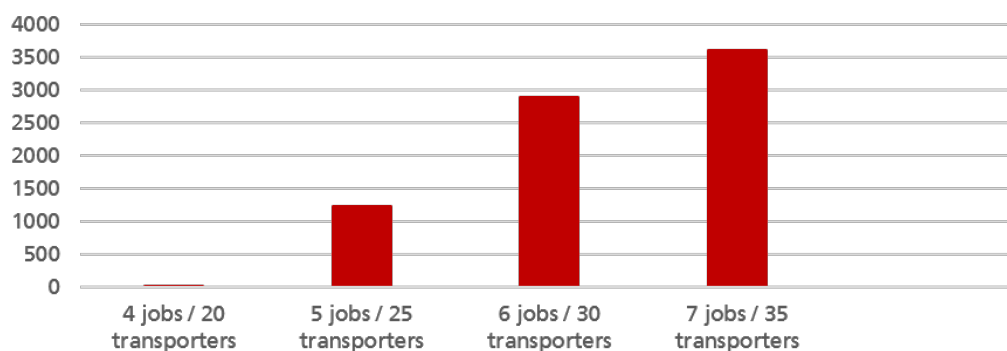


Figure 4. Average running time of Algorithm (SMIP) over five randomly generated instances, in seconds. We set up the solver with a one hour time-out limit, such that an instance takes a maximum of 3600 s duration.

Figure 5 presents the running times of the algorithms. Algorithm (sMIPg) is reasonably fast for our small problem instances of up to 10 jobs. All greedy heuristics always run in a few milliseconds. We remind here that the complexity of Algorithm (sGJ) is exponential in the saturating transporter cardinality, which should not exceed six in our test sets.

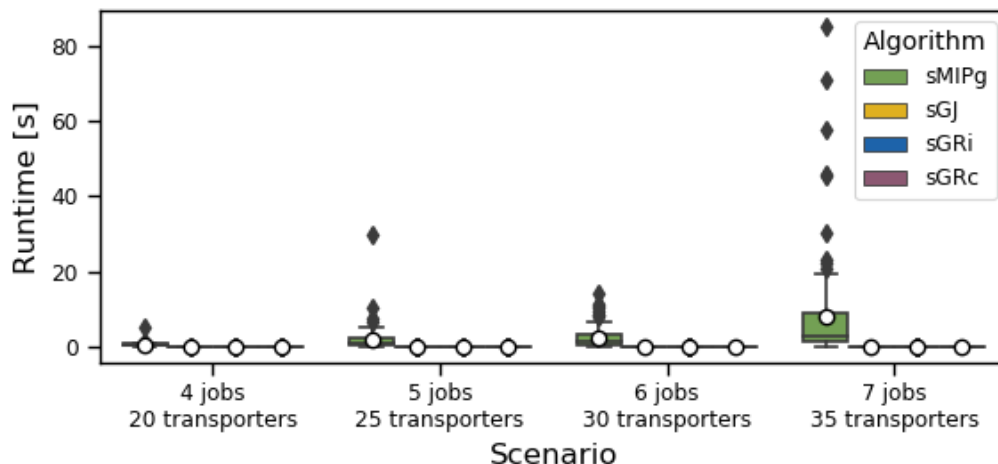


Figure 5. Distribution of running times of the algorithms, in seconds. The distributions are shown as box plots. The dot shows the average value and the line the median. The box (if it exists) shows the 25% to 75% quantile. If there is no box, then the values in those quantiles coincide with the average. Outliers are shown as diamonds.

Figures 6 and 7 show for every algorithm the total number of infeasible instances and the average number of used transporters, respectively. Since our test sets only contain feasible instances, the optimal algorithm (sMIPg) has no infeasible instance. Among the greedy heuristics, the heuristic by job (sGJ) performs better than the heuristics by transporters. This is an intuitive result in saturated scenarios, as (sGJ) looks for the tightest assignment for every job, whereas Algorithms (sGRi) and (sGRc) assign transporters on the fly, without ensuring that the transporters assigned to a same job fit well together. Algorithm (sGRc), which minimizes the completion time at every step, performs slightly worse than Algorithm (sGRi), which minimizes the idle time at every step, for a similar reason: since the objective is to remove the idle time, it is more beneficial to focus on the idle time than the completion time, as the latter also depends on the superfluous total size of the job.

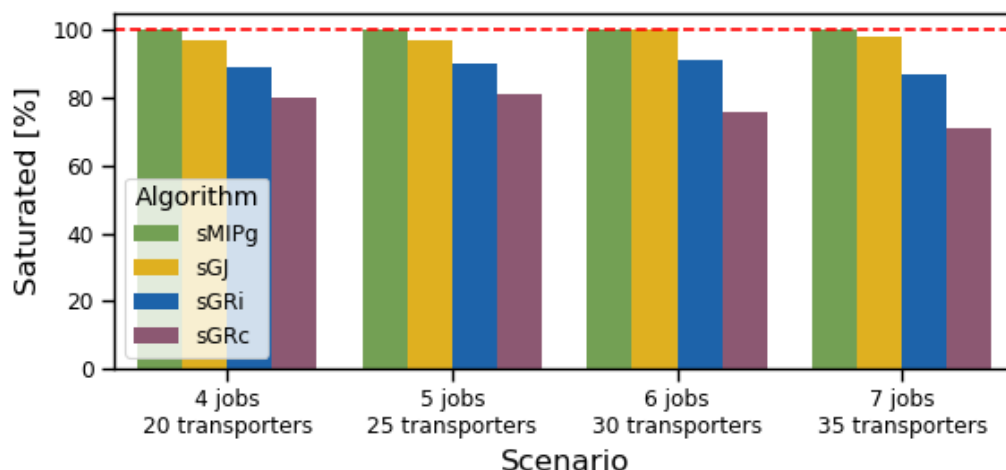


Figure 6. Share of the 100 instances where the respective algorithms do return a feasible, saturated solution.

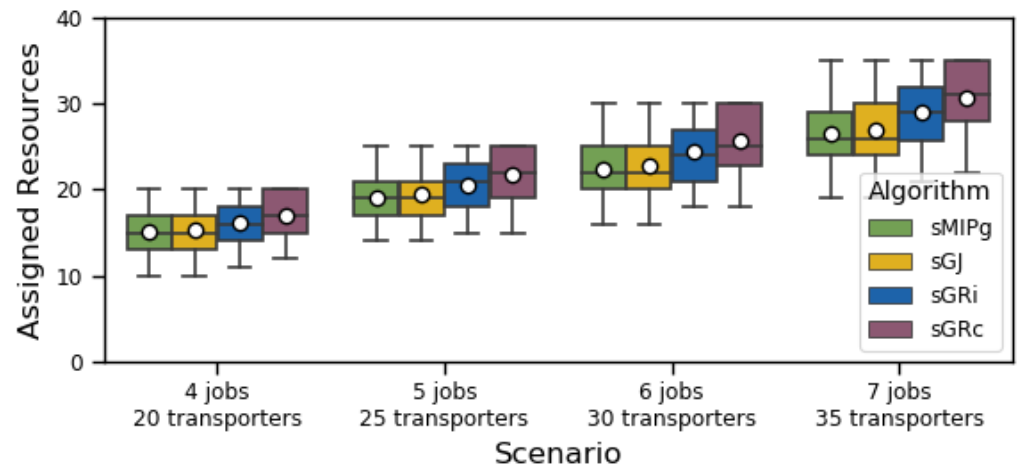


Figure 7. Distribution of assigned transporters (using the same boxplot settings as in Figure 5). Whenever an instance is infeasible, this number is set to the total number of transporters.

We conclude that when looking for a saturated solution, a good recommendation is to use the greedy heuristic by jobs (sGJ).

7.3. Performance of the General Algorithms

We now evaluate the Pareto frontier generated by the following algorithms:

- The three greedy algorithms (GJ), (GRi), and (GRc) generalizing their saturated counterparts (sGJ), (sGRi), and (sGRc).
- The binary search algorithm (BS.MIPg) solving an aggregated version of (ρ MIP) by grouping transporters with equal filling time.
- The binary search heuristics (BS.GJ) and (BS.GRi), which are iteratively called respectively (sGJ) and (sGRi) to solve the saturated mixed integer program (r MIP) with altered job return times.

Figure 8 shows the computed Pareto frontiers for our first simulation scenario, using *four* jobs and 20 transporters. Contrary to the saturated case, Algorithm (GJ) performs poorly for limited transporter fleets, as it over-invests transporters for the first jobs while letting future jobs have a single transporter. Since the last job typically gets assigned the smallest transporter in the fleet, adding an even smaller transporter to the fleet will further increase its completion time, hence potentially increasing the makespan of the assignment. In other words, Algorithm (GJ) may generate a worse, dominated solution while using a bigger transporter fleet. In order to create a non-increasing Pareto frontier, we set the makespan with N available transporters as the minimum makespan over every instance using at most N transporters.

Algorithm (GRi) is less efficient than Algorithm (GRc) for a similar reason: since we want to minimize the maximum job completion time, we should assign more transporters to jobs with the highest residual completion time instead of the job with the highest residual idle time.

However, the binary search-based heuristics generally perform very well when compared to the nearly optimal algorithm (BS.MIPg), with the exception of (BS.GJ) when using less than 10 transporters. This shows that virtually increasing the return times of the small jobs (with short saturated processing time) is a good way to balance the transporter fleet between the jobs and ensure a short makespan. Note that the Pareto frontiers generated by the binary search heuristics only start at eight transporters, because each job needs at least two transporters to be saturated.

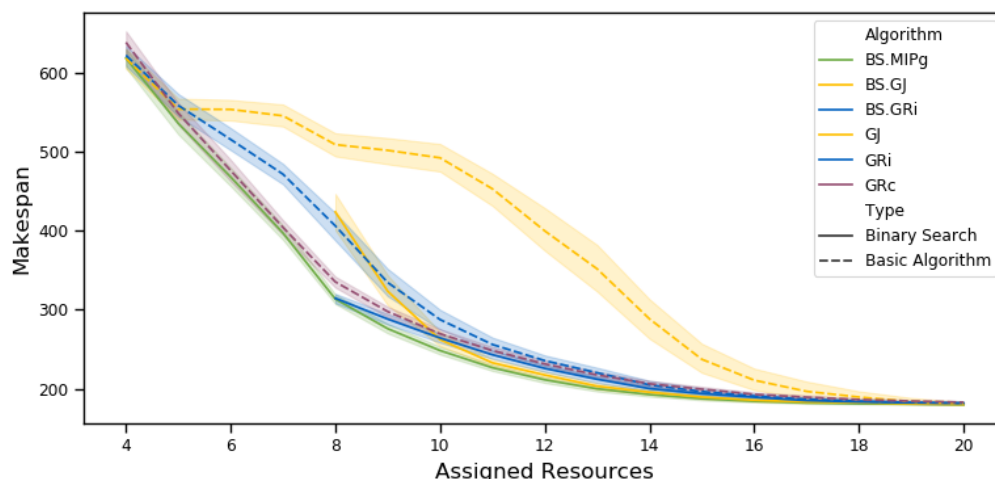


Figure 8. Median Pareto frontier over 100 instances with 95% confidence intervals. Algorithm (BS.MIPg) computes Pareto optimal solutions, whereas the other algorithms have no optimality guarantee. By design, Algorithms (BS.GJ) and (BS.GRi) cannot compute any solution using less than eight transporters.

Finally, Table 2 shows that the binary search-based heuristics are extremely fast, despite being 10 to 100 times slower than their greedy counterpart.

Table 2. Average running times to compute the Pareto frontiers over 100 instances, in seconds.

Algorithm	(BS.MIPg)	(GJ)	(GRi)	(GRc)	(BS.GJ)	(BS.GRi)
Runtimes [s]	33	0.00075	0.00011	0.00015	0.013	0.010

We conclude that to solve an instance of the transporter assignment problem, an efficient method consists of iteratively increasing the return time of the jobs until the problem has a saturated solution. Then, the computed saturated solution is a good solution to the initial assignment problem. An advantage of this method is that the saturated problem instances do not have to be solved to optimality, and most decent algorithms should induce a good solution to the transporter assignment problem.

8. Conclusions

The objective of the transporter allocation problem is to assign transporters to jobs to jointly minimize the makespan of the jobs and the number of assigned transporters. Crucially, transporters have a limited capacity, and usually, multiple transporters are assigned to a single job. After processing a share of the job load, a transporter must wait for a given time before it can process the job again. In particular, the goal is to compute the frontier of Pareto optimal solutions.

To the best of our knowledge, this is the first study on the assignment of an heterogeneous transporter fleet where transporters are regarded as a dynamic resource impacting the completion times and not as a fixed requirement.

We show that in general, it is NP-hard in the strong sense to compute even a single point on the Pareto frontier. In particular, it is NP-complete to decide whether a solution exists where every job has the same minimum completion time. Furthermore, the problem remains NP-hard in the usual sense, even if there are only two jobs.

On the other hand, when every transporter is identical, the whole Pareto frontier can be computed in a runtime polynomial in the number of jobs and transporters.

We provide several exact algorithms and heuristics to compute the minimum number of transporters needed such that all jobs are contracted to their minimum completion time.

Then, we use these results to compute Pareto frontiers for general (TAP) instances, both exactly and heuristically. Computational experiments show that our heuristics compute quickly decent approximations of the optimal solution, where the mixed integer program is already slow.

The results presented in this paper are part of a wider, practical study, in which we used a multi-level approach to solve applied problems in farming logistics together with an industrial partner. Assigning transporters to jobs was one of several levels of decision making in this regard, and our algorithms (adjusted for some additional practical needs) have proven to be useful also in early applied experiments.

In this paper, we assumed that every team of transporters only completes a single job. However, depending on the job sizes, there are also a variety of practical applications, where transporters are grouped into teams to fulfill several jobs per day in a sequence. Removing this limitation in a future study would significantly increase the number of use cases to which our results can be applied. Note that moving from one job to another may induce location-dependent traveling times, which we did not need to take into account in this paper. Even though we expect a single move from one job (or from the hub) to the next job to only have a minor impact on the solution, it will be important to find the best time to switch a transporter from one job to the next. Another challenge would be how to decide which jobs should be served by the same transporter team and which should receive a different team. Such a clustering, if not mandated by practical circumstances such as large travel times between the jobs, would be an additional degree of freedom, which can have a large impact on the optimality of a solution.

As a further area of extension, note that in our model, it is implicitly assumed that all teams of transporters are disjointed (since each transporter can only be assigned to one job). This is common practice to simplify the planning task, but it can lead to less efficient solutions. In some applications, such as [2,4], transporters are allowed to alternate between different jobs, optimizing the usage of each machine and hence providing better solutions. Allowing this for our problem would add an additional layer of difficulty. In particular, period lengths would no longer be constant but would differ, depending on when some transporters may serve another job. The scheduling decision, when to switch transporters to another job, would be much more impactful, and the scheduling of transporters to process each job would most likely have to be dealt with explicitly in that case.

Author Contributions: Conceptualization, N.L. and N.J.; methodology, N.J., N.L. and C.W.; software, N.J.; validation, N.J.; formal analysis, N.J., N.L. and C.W.; investigation, N.J., N.L. and C.W.; resources, N.J.; data curation, N.J. and N.L.; writing—original draft preparation, N.J. and N.L.; writing—review and editing, N.J., N.L. and C.W.; visualization, N.J. and N.L.; supervision, N.L.; project administration, N.L.; funding acquisition, N.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The input data of our numerical experiments presented in this study are openly available in Zenodo at https://zenodo.org/record/5583977#.YgYc79_MKUK (accessed 11 February 2022).

Acknowledgments: We thank Sven Krumke from TU Kaiserslautern for his helpful ideas concerning the NP-hardness proof (Theorem 1). We also thank our colleagues Sebastian Velten and Sven Jäger for helpful discussions and suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

(TAP)	Transporter Assignment Problem
(TAP _{C*} (n_r))	Decision version of the Transporter Assignment Problem
(sTAP)	Saturated Transporter Assignment Problem
\mathcal{O}	Complexity order of the algorithm
(HOM)	Our algorithm for (TAP) with homogeneous transporters
(sGRi)	Heuristic for (sTAP), greedily assigning a transporter to the job with maximum idle time per period
(sGRc)	Heuristic for (sTAP), greedily assigning a transporter to the job with maximum makespan
(sGJ)	Heuristic for (sTAP), greedily generating an saturating assignment to each job
(GRi)	Extension of (sGRi) to compute a Pareto frontier for (TAP)
(GRc)	Extension of (sGRc) to compute a Pareto frontier for (TAP)
(GJ)	Extension of (sGJ) to compute a Pareto frontier for (TAP)
MIP	Mixed Integer Program
(sMIP)	MIP formulation of (sTAP)
(sMIPg)	MIP formulation of (sTAP) aggregating transporters by model
(ρ MIP)	MIP formulation of (TAP) with fixed job idle rates I_j
(r MIP)	MIP formulation of (TAP) with fixed job idle rates and altered job return times
(BS)	Algorithm framework to compute a solution of (TAP) with minimum makespan
(BS.MIPg)	Exact algorithm for (TAP), using framework (BS) and an MIP solver for (ρ MIP)
(BS.GRi)	Heuristic for (TAP), using framework (BS) and (sGRi) to generate a solution for (r MIP)
(BS.GJ)	Heuristic for (TAP), using framework (BS) and (sGJ) to generate a solution for (r MIP)
\mathcal{J}	Set of jobs
\mathcal{R}	Set of transporters
\mathcal{M}	Set of transporter models
N_j	Number of jobs
N_R	Number of transporters
N_M	Number of transporter models
$M(m)$	Number of transporters with model m
j	Job variable
r	Transporter variable
m	Transporter model variable
$\kappa(r)$	Filling time of transporter r
$\delta(j)$	Return time of job j
$N_R^*(j)$	Saturating transporter cardinality for job j
$\bar{X}_{r,j}$	Boolean value of “transporter r is assigned to Job j ”
\bar{Y}_j	Number of transporters assigned to Job j
$\bar{Y}_{m,r}$	Number of “transporters with model m assigned to Job j ”
$\bar{X}_{m,r}$	Boolean value of “At least one transporter with model m is assigned to Job j ”
$C(j, \bar{X})$	Completion time of job j given assignment \bar{X}
$C^*(\bar{X})$	Makespan given assignment \bar{X} = maximum job completion time
C_{\min}^*	Minimum makespan over all feasible solutions
$C^{\text{sat}}(j)$	Saturated processing time of job j
C_{\max}^{sat}	Makespan in a saturated solution = maximum saturated job processing time
$I(j)$	Idle time per period for job j
$\rho(j, C)$	idle rate per period for job j to have completion time C
$L(j)$	Period length for job j

Appendix A. Proof of the Completion Time Estimate Given by Equation (1)

In this appendix, we show that the completion time estimate as given in Section 1, Equations (1)–(3) is accurate. To be precise, our goal is to show the following theorem.

Theorem A1. Given a set $R \subset \mathcal{R}$ of transporters assigned to some job $j \in \mathcal{J}$, it holds that

$$\left| C^{\min}(j) - C^{\text{sat}}(j) \frac{L(j)}{L(j) - I(j)} \right| \leq 2\delta(j)$$

where $C^{\min}(j)$ is the minimum completion time reachable when processing job j with the set of transporters R , and $L(j)$ and $I(j)$ are given by Equations (2) and (3), respectively.

Since in our practical applications, usually $C^{\text{sat}}(j)$ and therefore also $C(j)$ is much larger than the return time $\delta(j)$, Theorem A1 implies that the error incurred by using Equation (1) to estimate the completion times of jobs is negligible.

For the remainder of this appendix, we use notations R , j , $C^{\min}(j)$, L_j , and $I(j)$ as defined in Theorem A1. Furthermore, we continue to denote by $C(j)$ the estimated completion time of job j as given by Equation (1). Finally, let the resources in R be given as r_1, r_2, \dots, r_k , as numbered by the non-increasing order of filling times.

We first proof the following Lemma.

Lemma A1. If

$$L(j) = \sum_{r \in R} \kappa(r),$$

then, the transporters in R saturate job j , i.e., $C^{\min}(j) = C^{\text{sat}}(j)$.

Proof. Note that if $L(j) = \sum_{r \in R} \kappa(r)$, then $\sum_{r \in R} \kappa(r) \geq \delta(j) + \kappa(r_1)$, by definition of $L(j)$. In particular, for any subset $R' \subset R$ with exactly $k - 1$ elements, it holds that $\sum_{r \in R'} \kappa(r) \geq \delta(j)$. This means that using the transporters in R in a cyclic fashion, a transporter always returns before all other transporters finish their processing of job j . Thus, job j can be processed without idle times and the transporters in R saturate job j . \square

Consider next the schedule in which transporters in R are used in a cyclic fashion, in order of their numbering, that is, they use each transporter once, in order of the transporter numbering, then restart with transporter r_1 and continue, again in order of the transporter numbering and so on, until job j is fully finished. Denote this schedule by S^{cyc} and its completion time by C^{cyc} .

Lemma A2.

1. In schedule S^{cyc} , the time between the starts of two uses of transporter r_1 is exactly $L(j)$.
2. In schedule S^{cyc} , between any two uses of transporter r_1 , job j is idle for exactly $I(j)$ time. Furthermore, all idle time, if any, appears immediately before the second use of r_1 .
3. It holds that

$$C^{\text{cyc}} = \left(\left\lceil \frac{C^{\text{sat}}(j)}{L(j) - I(j)} \right\rceil - 1 \right) L(j) + ((C^{\text{sat}}(j) - 1) \bmod (L(j) - I(j))) + 1.$$

Proof. Proof of 1: By Lemma A1, if $L(j) = \sum_{r \in R} \kappa(r)$, then job j is saturated and clearly the time between the starts of two uses of a transporter r is given as the sum of filling times of all transporters, which is exactly $L(j)$. On the other hand, $\delta(j) + \kappa(r)$ is a lower bound for the time between the starts of two uses of any transporter, since between two starts, the transporter needs to finish its processing and then its whole return time. Since from $L(j) = \delta(j) + \kappa(r_1)$, it follows that $\delta(j) \geq \sum_{i=2}^k \kappa(r_i)$, this means by the time r_1 returns, no other transporter is processing job j so it can start immediately, leading again to a time of $L(j)$ between any two starts of uses of r_1 .

Proof of 2: Since, by definition, $I(j) = 0$ if and only if $L(j) = \sum_{r \in R} \kappa(r)$ and by Lemma A1 this means job j is saturated, in that case, the statement holds. Suppose $I(j) > 0$ and $L(j) = \delta(j) + \kappa(r_1) > \sum_{r \in R} \kappa(r)$. In that case, since during time $L(j)$, each transporter

processes job j exactly once, by construction of schedule S^{cyc} , the total time spent idle is given as

$$\delta(j) + \kappa(r_1) - \sum_{r \in \mathcal{R}} \kappa(r) = I(j),$$

by definition. Finally, note that due to the ordering of jobs in S^{cyc} , the time between the start of a use of any transporter r and the next time r is needed is given by $L(j) \geq \delta(j) + \kappa(r)$, so any transporter other than r_1 always arrives back at site before it is needed again. Therefore, any idle time can only happen before a use of transporter r_1 .

Proof of 3: During any time $L(j)$, exactly $L(j) - I(j)$ time is spent processing in schedule S^{cyc} , by Statement 1 and 2 of this Lemma. Thus, after $\left(\left\lceil \frac{C^{sat}(j)}{L(j) - I(j)} \right\rceil - 1\right)L(j)$ time,

$$\left(\left\lceil \frac{C^{sat}(j)}{L(j) - I(j)} \right\rceil - 1\right)(L(j) - I(j))$$

of job j has been processed. It remains an amount of either exactly $L(j) - I(j)$ if $L(j) - I(j)$ divides $C^{sat}(j)$ or $(C^{sat}(j) \bmod (L(j) - I(j))) < L(j) - I(j) = \sum_{r \in \mathcal{R}} \kappa(r)$, otherwise, which combined yields a remaining amount of

$$((C^{sat}(j) - 1) \bmod (L(j) - I(j))) + 1.$$

Finally, note that the remaining amount can be processed immediately and without interruption, since after time $\left(\left\lceil \frac{C^{sat}(j)}{L(j) - I(j)} \right\rceil - 1\right)L(j)$, transporter r_1 is ready to process, and by the second statement of this lemma, no further idle time happens before all the remaining amount is processed. \square

Obviously, it holds that $C^{min}(j) \leq C^{cyc}$. Furthermore, note that

$$\begin{aligned} & C^{cyc} - C(j) \\ &= \left(\left\lceil \frac{C^{sat}(j)}{L(j) - I(j)} \right\rceil - 1\right)L(j) + ((C^{sat}(j) - 1) \bmod (L(j) - I(j))) + 1 \\ &\quad - C^{sat}(j) \frac{L(j)}{L(j) - I(j)} \\ &= ((C^{sat}(j) - 1) \bmod (L(j) - I(j))) + 1 \\ &\quad - L(j) \frac{((C^{sat}(j) - 1) \bmod (L(j) - I(j))) + 1}{L(j) - I(j)} \\ &\leq 0, \end{aligned}$$

where the last inequality is due to $L(j) > I(j) \geq 0$. Therefore, it is proven that $C^{min}(j) - C(j) \leq 0 \leq 2\delta(j)$.

We are left to prove that $C(j) - C^{min}(j) \leq 2\delta(j)$. First, we show the following lemma, extending the statement of Lemma A2 (2).

Lemma A3. *In any schedule using transporters in R to process job j , between two uses of transporter r_1 , idle time of at least $I(j)$ happens.*

Proof. If $I(j) = 0$, there is nothing to prove. So assume $I(j) > 0$ and $L(j) = \delta(j) + \kappa(r_1) > \sum_{r \in R} \kappa(r)$. In the proof of Lemma A2, it was shown that if every other transporter is used exactly once between two uses of r_1 , then between the two uses of r_1 , the idle time of exactly $I(j)$ is reached. Using fewer transporters in between does obviously not reduce idle time. On the other hand, using any other transporter r^* twice without using r_1 in between leads to an idle time between the two uses of r^* of at least

$$\delta(j) - \sum_{r \in R, r \neq r_1, r \neq r^*} \kappa(r) > I(j), \tag{A1}$$

which means the idle time between the two uses of r_1 would also be larger, which finishes the proof. \square

Using Lemma A3 and Equation (A1), we can prove the following lemma.

Lemma A4. For any schedule using transporters in R to process job j , let transporter r^* be a transporter with the most uses and let r^* be used exactly ℓ times. Then, job j is idle for at least

$$(\ell - 2)I(j)$$

time.

Proof. If r_1 is used ℓ times, by Lemma A3, job j is idle for at least $(\ell - 1)I(j)$, and the statement is proven. So, assume transporter r_1 is used $\ell_1 < \ell$ times. By Lemma A3, between any two uses of transporter r_1 , idle time of at least $I(j)$ appears. However, by Equation (A1), between any two uses of job r^* where r_1 is not used in between, idle time of at least

$$I^* = \delta(j) - \sum_{r \in R, r \neq r_1, r \neq r^*} \kappa(r) > I(j)$$

appears. Thus, the total idle time of job j is at least

$$(\ell_1 - 1)I(j) + (\ell - 1 - \ell_1)I^* \geq (\ell - 2)I(j)$$

time units long. \square

Note that using schedule S^{cyc} , each transporter can be used ℓ times, causing idle time of at most $(\ell - 1)I(j)$. Thus, by Lemma A4, we have

$$C^{cyc} - C^{min}(j) \leq I(j) \leq \delta(j). \tag{A2}$$

Furthermore,

$$\begin{aligned} & C(j) - C^{cyc} \\ &= L(j) \frac{((C^{sat}(j) - 1) \bmod (L(j) - I(j))) + 1}{L(j) - I(j)} \\ &\quad - ((C^{sat}(j) - 1) \bmod (L(j) - I(j))) + 1 \\ &= \frac{L(j) - L(j) + I(j)}{L(j) - I(j)} ([(C^{sat}(j) - 1) \bmod (L(j) - I(j))] + 1) \\ &\leq I(j) \leq \delta(j). \end{aligned}$$

Therefore, and due to Equation A2, we have $C(j) - C^{min}(j) \leq 2\delta(j)$, which finishes the proof of Theorem A1. Note that in the case where the transporters in R saturate job j and $I(j) = 0$, then the above and Equation A2 even imply that $C(j) = C^{min}(j)$. In addition, note that we actually showed that $C(j) - C^{min}(j) \leq 2I(j) \leq 2\delta(j)$, and thus the worst case estimate as given in Theorem A1 can only happen when only a single transporter is assigned to a job (which is the only time when $I(j) = \delta(j)$).

References

1. Jensen, M.F.; Bochtis, D.; Sørensen, C.G. Coverage planning for capacitated field operations, part II: Optimisation. *Biosyst. Eng.* **2015**, *139*, 149–164. [CrossRef]
2. Aliano Filho, A.; Melo, T.; Pato, M.V. A bi-objective mathematical model for integrated planning of sugarcane harvesting and transport operations. *Comput. Oper. Res.* **2021**, *134*, 105419. [CrossRef]
3. Kuenzel, R.; Teizer, J.; Mueller, M.; Blickle, A. SmartSite: Intelligent and autonomous environments, machinery, and processes to realize smart road construction projects. *Autom. Constr.* **2016**, *71*, 21–33. [CrossRef]
4. Payr, F.; Schmid, V. Optimizing deliveries of ready-mixed concrete. In Proceedings of the 2009 2nd International Symposium on Logistics and Industrial Informatics, Linz, Austria, 10–12 September 2009; pp. 1–6.

5. Wörz, S.K. Entwicklung eines Planungssystems zur Optimierung von Agrarlogistik-Prozessen. Ph.D. Thesis, Technische Universität München, Munich, Germany, 2017.
6. Lopez Milan, E.; Miquel Fernandez, S.; Pla Aragones, L.M. Sugar cane transportation in Cuba, a case study. *Eur. J. Oper. Res.* **2006**, *174*, 374–386. [[CrossRef](#)]
7. Salassi, M.E.; Barker, F.G. Reducing harvest costs through coordinated sugarcane harvest and transport operations in Louisiana. *J. Am. Soc. Sugar Cane Technol.* **2008**, *28*, 32–41.
8. Ahmadi-Javid, A.; Hooshangi-Tabrizi, P. Integrating employee timetabling with scheduling of machines and transporters in a job-shop environment: A mathematical formulation and an Anarchic Society Optimization algorithm. *Comput. Oper. Res.* **2017**, *84*, 73–91. [[CrossRef](#)]
9. Bochtis, D.D.; Sørensen, C.G.; Busato, P. Advances in agricultural machinery management: A review. *Biosyst. Eng.* **2014**, *126*, 69–81. [[CrossRef](#)]
10. Kusumastuti, R.D.; van Donk, D.P.; Teunter, R. Crop-related harvesting and processing planning: A review. *Int. J. Prod. Econ.* **2016**, *174*, 76–92. [[CrossRef](#)]
11. Garey, M.R.; Johnson, D.S. *Computers and Intractability*; Freeman: San Francisco, CA, USA, 1979; Volume 174,
12. Morales-Chávez, M.M.; Soto-Mejía, J.A.; Sarache, W. A mixed-integer linear programming model for harvesting, loading and transporting sugarcane: A case study in Peru. *DYNA* **2016**, *83*, 173–179. [[CrossRef](#)]
13. Ehrgott, M. *Multicriteria Optimization*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2005; Volume 491.
14. Ralphs, T.K.; Kopman, L.; Pulleyblank, W.R.; Trotter, L.E. On the capacitated vehicle routing problem. *Math. Program.* **2003**, *94*, 343–359. [[CrossRef](#)]
15. Bharadwaj, V.; Ghose, D.; Robertazzi, T.G. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Clust. Comput.* **2003**, *6*, 7–17. [[CrossRef](#)]
16. Ghanbari, S.; Othman, M. Comprehensive Review on Divisible Load Theory: Concepts, Strategies, and Approaches. *Math. Probl. Eng.* **2014**, *2014*, 1–13. [[CrossRef](#)]
17. Drozdowski, M.; Lawenda, M. The combinatorics in divisible load scheduling. *Found. Comput. Decis. Sci.* **2005**, *30*, 297–308.
18. Shakhlevich, N.V. Scheduling Divisible Loads to Optimize the Computation Time and Cost. In *Economics of Grids, Clouds, Systems, and Services*; Altmann, J., Vanmechelen, K., Rana, O.F., Eds.; Springer International Publishing: Cham, Switzerland, 2013; pp. 138–148.
19. Drozdowski, M.; Shakhlevich, N. Scheduling divisible loads with time and cost constraints. *J. Sched.* **2019**, *24*, 507–521. [[CrossRef](#)]
20. Salkin, H.M.; De Kluyver, C.A. The knapsack problem: A survey. *Nav. Res. Logist. Q.* **1975**, *22*, 127–144. [[CrossRef](#)]
21. Hiley, A.; Julstrom, B.A. The quadratic multiple knapsack problem and three heuristic approaches to it. In Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Seattle, WA, USA, 8–12 July 2006; pp. 547–552.
22. Boyer, V.; Elkihel, M.; El Baz, D. Heuristics for the 0–1 multidimensional knapsack problem. *Eur. J. Oper. Res.* **2009**, *199*, 658–664. [[CrossRef](#)]
23. Chekuri, C.; Khanna, S. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM J. Comput.* **2005**, *35*, 713–728. [[CrossRef](#)]
24. Jansen, K. Parameterized Approximation Scheme for the Multiple Knapsack Problem. *SIAM J. Comput.* **2010**, *39*, 1392–1412. [[CrossRef](#)]
25. Martello, S.; Toth, P. A Bound and Bound algorithm for the zero-one multiple knapsack problem. *Discret. Appl. Math.* **1981**, *3*, 275–288. [[CrossRef](#)]
26. Pisinger, D. An exact algorithm for large multiple knapsack problems. *Eur. J. Oper. Res.* **1999**, *114*, 528–541. [[CrossRef](#)]
27. Dawande, M.; Kalagnanam, J.; Keskinocak, P.; Salman, F.S.; Ravi, R. Approximation algorithms for the multiple knapsack problem with assignment restrictions. *J. Comb. Optim.* **2000**, *4*, 171–186. [[CrossRef](#)]
28. Brauner, N.; Crama, Y.; Grigoriev, A.; van de Klundert, J. A Framework for the Complexity of High-Multiplicity Scheduling Problems. *J. Comb. Optim.* **2005**, *9*, 313–323. [[CrossRef](#)]