1D Vehicle Scheduling with Conflicts

Torsten J. Gellert Felix G. König

Technische Universität Berlin, Institut für Mathematik, MA 5-1
Straße des 17. Juni 136, 10623 Berlin, Germany

{gellert,fkoenig}@math.tu-berlin.de

Preprint 2010/22

October 5, 2010

Abstract

Systems of rail-mounted vehicles play a key role in many logistics applications, and the efficiency of their operation frequently has a significant impact on the overall performance of the surrounding production environment. In theory, assigning transport requests to the vehicles of such systems and scheduling their execution amounts to finding k tours on a common line, where tours may never cross each other in time—dynamic collision constraints need to be respected. The goal is to minimize the makespan for a given set of transport requests.

We establish a model capturing the core challenges in transport planning problems of this type and relate it to other models in literature. After proving *NP*-hardness for a basic version of the problem, the large part of the paper is dedicated to devising various fast heuristic algorithms suitable for practice. We present computational results regarding the performance of the algorithms proposed for several classes of problem instances.

1 Introduction

In many logistics applications, transport requests are conducted in parallel by several vehicles moving along a fixed shared pathway. Examples include cranes mounted on a common rail, like gantry cranes loading and unloading containers in intermodal transportation, or forklifts moving along a narrow passageway in large warehouses. In such transportation systems, simultaneous execution of certain transport requests may lead to *conflicts*, i.e., may be impossible since vehicles cannot pass each other and/or have to keep certain safety distances.

Assigning transports to vehicles and sequencing transports on each vehicle constitutes an interesting case of a multi-vehicle routing problem. While the one-dimensional character of the transportation system simplifies routing for a single vehicle, scheduling conflicts complicate the problem. After proving that minimizing the makespan for the execution of transports in such transport systems is *NP*-hard, we develop a number of distinct fast heuristic algorithms suitable for practice. We evaluate and compare their performance for different classes of instances representing use cases from different application areas.

While one-dimensional transport systems with conflicts occur frequently in case studies in the operations research literature, our work is the first to propose a problem formulation general enough to be useful for a wide range of applications. Instances in different application areas may differ significantly in flavor, and our experimental results provide valuable insight regarding the fitness of different solution approaches for them.

2 Problem Formulation

In an instance of 1D VEHICLE SCHEDULING (1D-VS), we are given a set of n transport requests (or jobs) J, and k vehicles to conduct them. Each $j \in J$ is specified by its start and end points on the line, $\alpha_j, \beta_j \in \mathbb{R}$, and each vehicle i has an initial position $p_i \in \mathbb{R}$, $i = 1, \ldots, k$. We assume that all vehicles travel along the line at uniform, w.l.o.g. unit speed. The length of a job is $\ell_j := |\beta_j - \alpha_j|$.

A solution is a partition of J into k subsets $J_1, \ldots, J_k$, together with a start time t_j for each $j \in J$. The execution of a transport may not be delayed or preempted, i.e., for each $j \in J_i$, vehicle i needs to be at position α_j at time t_j and arrive at position β_j at time $t_j + \ell_j$. Our objective is to minimize the makespan, i.e., $\max_{j \in J} t_j + \ell_j$.

As all vehicles reside on the line in a fixed order and may never pass each other, detours may be necessary for vehicles at times in order to make room for others. Hence *evasion jobs* with corresponding start times are included in a solution. A solution is feasible if all jobs and evasions can be executed on schedule without vehicles passing each other. For the sake of unambiguousness, we assume w.l.o.g. that vehicles spend all possible waiting time in their schedules as late as possible (i.e., at the start point of their next job, respectively).

Related Work When $k = 1$, 1D-VS becomes a special case of the TRAVELING SALESMAN PROBLEM (TSP) and fits the characterization of a polynomially solvable special case by Gilmore and Gomory [5]. The generalization of 1D-VS for $k = 1$ where the vehicle travels on an undirected graph instead of a line, and transport requests are ordered pairs of nodes, is the well-studied and *NP*-hard STACKER CRANE PROBLEM (SCP) [3, 8]. In [3], also the k-SCP for k vehicles is studied, but conflicts between vehicles are not part of the model.

In [6], also Heinrichs and Moll study one-dimensional transport systems with conflicts. In their model, however, start times for the jobs are part of the input, and they ask if all jobs can be feasibly assigned to k vehicles. They relate the problem to covering line segments by angle-restricted curves and show that this question can be decided in polynomial time by shortest path computations. They mention galvanization lines in steel production as an application.

Moreover, one-dimensional transport systems occur frequently as a subproblem in logistics case studies in the operations research literature, predominantly in container logistics. In quay crane scheduling [10, 9], cranes loading and unloading a ship move on a common rail, and sets of transport requests with precedence constraints need to be served at certain points. In other works, the focus lies on higher level planning problems like the deployment of cranes to storage blocks [2, 1] or the integrated optimization of loading and storage operations [4]. When it comes to the actual scheduling of cranes on a rail, the approach commonly used assigns cranes to disjoint sections and solves single crane problems [4, 9]. We include an evaluation of such approach in our results in Sect. 4.

Hardness In order to prove *NP*-hardness of 1D-VS for an fixed number of vehicles $k \geq 2$, we give a reduction from *NP*-complete PARTITION [7]. In an instance X of the latter, we are given a set of n numbers $\mathcal{A} = \{a_1, \ldots, a_n\}$ with $\sum_{i=1,\ldots,n} a_i = A$, and the question is whether there is a subset $\mathcal{A}' \subseteq \mathcal{A}$ with $\sum_{a_i \in \mathcal{A}'} a_i = A/2$. Given X, we will construct an instance Y of 1D-VS such that the optimum makespan for Y is $3A$ if and only if X is a yes-instance.

Our instance Y contains four special jobs $(0,A)$, $(A,0)$, $(0,-A)$, $(-A,0)$, and $2n$ jobs $(0,a_i)$, $(a_i,0)$, $i = 1, \ldots, n$, see Fig. 1 for an illustration. We initially position two vehicles at 0, and may moreover assume w.l.o.g. that $k = 2$: If $k > 2$, we can keep the $k-2$ additional vehicles occupied for time $3A$ by initially positioning them at A and adding $k-2$ identical jobs $(A, 4A)$. The hardness result now follows immediately from the following lemma.

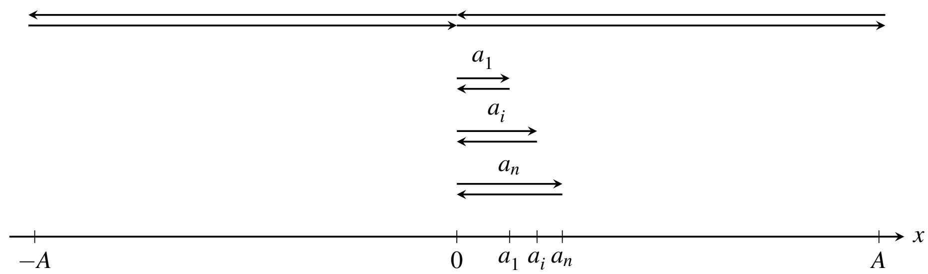

Figure 1: 1D-VS instance constructed from an instance of PARTITION.

Lemma 1 *X is a yes-instance if and only if Y has a solution with makespan $3A$.*

Proof. First assume that X is a yes-instance, and let $\mathcal{A}'$ denote a subset of $\mathcal{A}$ with $\sum_{a_i \in \mathcal{A}'} a_i = A/2$. Then in Y, we assign jobs $(0,-A),(-A,0)$ to one vehicle before jobs $(0,a_i),(a_i,0)$ for all $a_i \in \mathcal{A}'$. The other vehicle executes jobs $(0,a_i),(a_i,0)$ for all $a_i \notin \mathcal{A}'$ first, and then jobs $(0,A),(A,0)$. It is easy to verify that this is a feasible solution with makespan $3A$.

Conversely, assume there is a feasible solution to Y with makespan $3A$. A trivial lower bound on the makespan of Y is the sum of job length divided by the number of vehicles, i.e.,

$$\frac{1}{k}(4A + 2 \sum_{i=1,\ldots,n} a_i + (k-2)3A) = 3A.$$

So in the solution above, no vehicle can cover any distance without executing a job. Consequently, one vehicle must execute jobs $(0,A),(A,0)$, and the other jobs $(0,-A),(-A,0)$, as in any other case the makespan of $3A$ would be exceeded. Moreover, jobs $(0,a_i)$ and $(a_i,0)$ must be assigned to the same vehicle for all $i = 1,\ldots,n$. Hence, a makespan of $3A$ can only be realized if one vehicle is assigned a subset of the jobs $(0,a_i)$ with total length $A/2$, so X must be a yes-instance. □

Theorem 2 1D VEHICLE SCHEDULING *is NP-hard for any fixed number of vehicles $k \geq 2$.*

3 Algorithms

We describe three types of fast heuristic algorithms for 1D-VS. A major objective in our design is fitness for a production environment where fast runtimes are imperative. Every approach follows a distinct algorithmic idea and thus leads to different results on the various sets of test instances as we report in Sect. 4.

A major challenge in 1D-VS is avoiding conflicts, i.e., ensuring that different vehicles' routes never cross each other. To this end we pursue two different techniques. In the algorithm described first, we schedule transports on all vehicles simultaneously and explicitly prohibit conflicting routes. In the latter two approaches, we decompose the problem into subproblems which we solve for a single vehicle, which can be done exactly in time $\mathcal{O}(n \log n)$ [5]. When assembling the final solution from these different parts, conflicts are avoided implicitly.

3.1 Insertion with Antichain Constraint

Our first algorithm for 1D-VS is based on a characterization of feasible solutions which follows from results in [6]. A job j with an associated start time t_j can be conveniently represented as a line segment in the plane. In one dimension, the line segment spans the job's spatial dimension $[\alpha_j;\beta_j]$, in the other its duration $[t_j;t_j+\ell_j]$. When vehicles travel at unit speed, these line segments have a slope of ± 1. See Fig. 2 for an illustration if a job in a space-time-diagram.

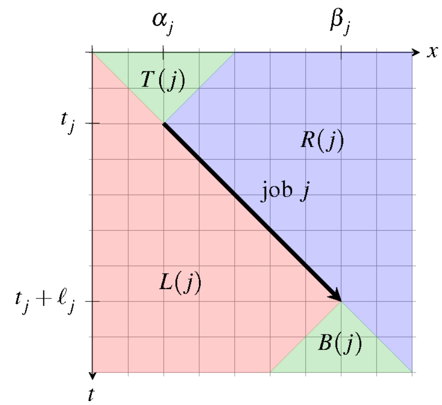

Figure 2: Line segment representation of a job j with associated start time t_j in a space-time-diagram. Job j partitions the plane into four regions. Any vehicle executing j at time t_j has to move through region $T(j)$ before, and through region $B(j)$ afterwards. On the other hand, any vehicle which ever enters region $R(j)$ or $L(j)$ can never execute j at time t_j.

Quite obviously, in a feasible solution, any subset of $k+1$ jobs has start times such that at least two of these jobs can be performed one after the other by the same vehicle—a necessary condition for a solution to be feasible. In [6], the authors essentially prove that this condition is also sufficient.

More formally, let $\prec$ denote a binary relation on jobs with assigned start times $(i,t_i) \in J \times \mathbb{R}$, and let $(i,t_i) \prec (j,t_j)$ denote that i has to be performed by a different vehicle than j, and this latter vehicle has to reside to the left of the vehicle performing j. In other words, the line segment for (i,t_i) has non-empty intersection with $L(j)$.

Definition 1 (antichain graph) *Let $T = (t_1, \dots, t_n)$ be an assignment of start times to jobs $i \in J$. The directed graph $\dot{G} = (J,E)$ is called the* antichain graph *for (J,T) if $(i,j) \in E \Leftrightarrow i \prec j$.*

Lemma 3 ([6]) *For a set of jobs with assigned start times (J,T), we have:*

- *J can be scheduled with start times T on k vehicles $\Leftrightarrow$ $\dot{G}$ is cycle-free and* $\mathrm{diam}(\dot{G}) \leq k$
- *In time $\mathcal{O}(|J|^2)$, such solution can be constructed from T, or it can be proven that it does not exist.*

So essentially, an algorithm for 1D-VS merely needs to compute feasible start times T, then an assignment of jobs to vehicles and the needed evasion moves can be computed accordingly. This leads to the following idea for an algorithm: Take all jobs in J, and in some fixed order set their start time and insert the resulting node into the antichain graph $\dot{G}$. We start with $\dot{G} = \emptyset$ and always choose the earliest start times which avoids a cycle and a path of length greater than k in $\dot{G}$. See Alg. 1 for a formal description of this procedure.

As insertion order, we use the order given by the start times of an optimal solution for a single vehicle (which can be computed easily). To boost the performance of this approach at the cost of a higher runtime, meta-heuristics or local search algorithms can be used to modify the insertion order. In a variant of our algorithm, we perform local search in the neighborhood defined by simple swaps of positions of any possible pair of jobs, immediately moving on to a neighboring solution if it has a better makespan. We present results in Sect. 4.

Mixed Integer Program The antichain graph also motivates a mixed integer programming formulation of 1D-VS. Its fractional variables are job start times, but integer variables are needed in order to formulate feasibility constraints in terms of the resulting antichain graph. We omit a detailed description of this model here, as only instances up to twenty jobs and two vehicles could be solved employing state-of-the-art solvers, and its linear relaxation does not seem to yield worthy lower bounds.

Algorithm 1 `insert`

Require: instance (J,k) of 1D-VS, insertion order π
Ensure: solution to 1D-VS

$T_{\text{start}} :=$ empty list of jobs with start times, $\dot{G}$:=empty antichain graph
for $i := \pi(1)$ **to** $\pi(n)$ **do**
  $j := i$-th job in J $\{$ select the next job via the permutation$\}$
  $t := \alpha_j$
  $V(\dot{G}) := V(\dot{G}) \cup \{j\}$ $\{$insert new node to the graph$\}$
  while (j,t) not added to T_{start} **do**
    $\{$add all new edges to the anti chaingraph$\}$
    $V_1 := \{i \in J | i \prec j\}$, $V_2 := \{i \in J | j \prec i\}$
    $E(\dot{G}) := E(\dot{G}) \cup \{(i,j) | i \in V_1\} \cup \{(j,i) | i \in V_2\}$
    if $\operatorname{diam}(\dot{G}) > k$ **or** $\dot{G}$ has a cycle **then** $\{$remove all edges and increment the time variable $t\}$
      $E(\dot{G}) := E(\dot{G}) \setminus (\{(i,j) | i \in V_1\} \cup \{(j,i) | i \in V_2\})$
      increment t such that at least one of the current relations between j and $V_1 \cup V_2$ does not hold any longer
    else $\{$adopt start time if it is feasible$\}$
      add (j,t) to T_{start}
    end if
  end while
end for
return solution to 1D-VS given by T_{start}

3.2 Clustering

In our second approach, the key idea is to treat all k vehicles like one vehicle with capacity k. We cluster jobs going in the same direction to fill this capacity as well as possible and solve the scheduling problem for clusters on one vehicle exactly. Finally, all k vehicles execute this solution in parallel, while each of the (potentially more than k) jobs in a cluster is scheduled on one of the k vehicles. The following definition is illustrated in Fig. 3.

Definition 2 (cluster) *A subset $\mathcal{C} \subseteq J$ of jobs is called a* positive (negative) cluster of width k *if all jobs in $\mathcal{C}$ have positive (negative) direction and no point of the line is contained in more than k of them, i.e.,*

$$\alpha_j < \beta_j \quad \forall j \in \mathcal{C} \quad \left(\alpha_j > \beta_j \quad \forall j \in \mathcal{C}\right) \quad \textit{and} \quad |\{j \in \mathcal{C} : \; p \in [\alpha_j; \beta_j]\}| \leq k \quad \forall p \in \mathbb{R}.$$

Moreover, the start and end points of a positive (negative) cluster $\mathcal{C}$ are defined as

$$\alpha(J_{\mathcal{C}}) := \min_{j \in \mathcal{C}} \; \alpha_j \; \left(\max_{j \in \mathcal{C}} \; \alpha_j\right) \quad \textit{and} \quad \beta(J_{\mathcal{C}}) := \max_{j \in \mathcal{C}} \; \beta_j \; \left(\min_{j \in \mathcal{C}} \; \beta_j\right).$$

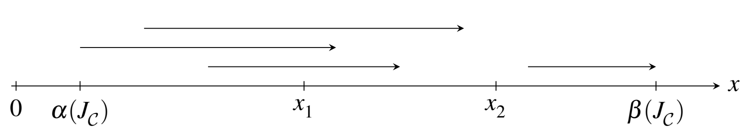

Figure 3: A positive cluster of width three; point x_1 is contained in three jobs of the cluster, x_2 in none.

Definition 3 (clustering) *A partition $\mathcal{C} = (\mathcal{C}_1, \ldots, \mathcal{C}_m)$ of a set of jobs J into clusters of width k is called a k-*clustering *of J.*

Our algorithm first finds a k-clustering, then defines a job $j_{\mathcal{C}_i} := (\alpha(\mathcal{C}_i), \beta(\mathcal{C}_i))$ for each resulting cluster $\mathcal{C}_i$, and finally solves the resulting smaller instance of 1D-VS on one vehicle exactly. When executing the resulting solution with all k vehicles in parallel, all jobs of each cluster can be handled simultaneously. A formal description of the algorithm is given in Alg. 2.

Algorithm 2 `cluster`

Require: instance (J,k) of 1D-VS
Ensure: solution to 1D-VS

$\mathcal{C}^+ :=$ k-clustering of all jobs with $\alpha_j < \beta_j$
$\mathcal{C}^- :=$ k-clustering of all jobs with $\beta_j < \alpha_j$
$Sol_1 :=$ optimal solution for the 1D-VS instance $(J', 1)$ with $J' = \{j_{\mathcal{C}_i} | \mathcal{C}_i \in \mathcal{C}^+ \cup \mathcal{C}^-\}$
for $j' \in J'$ **do**
  find the cluster $\mathcal{C}_{j'}$ which belongs to j'
  set the start times of all jobs j in $\mathcal{C}_{j'}$ to the start time of j in Sol_1 plus $|\alpha_j - \alpha(\mathcal{C}_{j'})|$
end for
return start times and partition $J_1, \ldots, J_k$ defined by $\mathcal{C}^+, \mathcal{C}^-$

Clearly, clusters should cover each point by as many jobs as possible (up to k) to utilize all available vehicles well. We compute positive (negative) k-clusters by first sorting all positive (negative) jobs in increasing (decreasing) order of their α_js. Then, starting with an empty clustering, we consider all jobs in this order and insert each job j into the first cluster for which j does not increase its width beyond k.

3.3 Disjoint Domains

At its core, this last heuristic mimics a rule of thumb used frequently when scheduling e.g., gantry cranes on a common rail in practice [4, 9]. The k available vehicles are assigned to k disjoint sections of the line, and all transport requests which lie completely inside one section are assigned to the corresponding vehicle. Again, the resulting subproblems on one vehicle can be solved exactly. However, transports across section borders demand special care, and section borders need to be chosen wisely as to balance all vehicles' work loads as well as possible.

We restrict our choice of initial section borders to start and end points of jobs and aim to balance the amount of work, i.e., the sum of the lengths of (partial) jobs which lie in each section. More precisely, we minimize the maximum difference between two section w.r.t. work to be performed. This can be done by simply evaluating all of the quadratically many possibilities.

Starting from this initial partitioning of the line into k sections, our algorithm repeatedly schedules all jobs inside each domain on a single vehicle before merging adjacent domains to obtain new subproblems. The procedure is described formally in Alg. 3.

4 Computational Study

We now evaluate the performance of our algorithms in different application scenarios. Before stating numerical results, we give a brief description of the instances used.

4.1 Instances

We generated five distinct sets of instances containing random transport requests, designed to model use cases from different applications. In each set of instances, job lengths are chosen uniformly at random from one of the following ranges measured in percentage of storage length: 0–5%, 0–15%, 0–30%, 0–50%, and 0–100%. Each set comprises twenty instances containing 100 jobs each, and we ran our algorithms on each instance for $k \in \{2, 3, 4, 5\}$ vehicles.

Algorithm 3 `disjoint`

Require: instance (J,k) of 1D-VS
Ensure: solution to 1D-VS
$D := \{d_i | 1 \leq i \leq k\}$ { find a partition of the storage into disjoint domains}
create empty sets $J_1, \ldots, J_k$
while $J \neq \emptyset$ **do**
  for $i := 1$ **to** $|D|$ **do** { treat each domain on its own}
    $Sol_{\min} := \emptyset$ {saves the solution with the smallest makespan}
    for $c \in d_i$ **do** {for all vehicles currently in d_i}
      $Sol' :=$ optimal solution for the jobs in d_i with vehicle c
      if makespan$(Sol') <$ makespan$(Sol_{\min})$ **then**
        $Sol_{\min} := Sol'$
        $c' = c$ {stores the vehicle used for $Sol_{\min}$}
      end if
    end for
    $J_{c'} := J_{c'} \cup \{j | j \in d_i\}$
    set start times for $\{j | j \in d_i\}$ to values in $Sol_{\min}$
    add evasion jobs to all J_c for $c' \neq c \in d_i$, s.t. they wait until c' finished its last job
    $J := J \backslash \{j | j \in d_i\}$ {remove all processed jobs}
  end for
  $D := \{d_{2i} \cup d_{2i+1} | 1 \leq i \leq \left\lfloor \frac{|D|}{2} \right\rfloor\}$ { merge pairs of neighboring domains }
end while
return start times and partition $J_1, \ldots, J_k$

4.2 Results

Tab. 1–5 compares the makespans obtained by our algorithms for the different sets of instances, respectively. We state numbers for Alg. 1–3 and an extension of Alg. 1 where we perform local search on the insertion sequence as described in Sect.3.1. Our basic algorithms' runtimes are extremely fast (well below one second for all instances), while the extended local search algorithm naturally has an increased runtime of some minutes per instance.

Column ∅ contains the average makespan over all twenty instances of a set for each algorithm, normalized to the best basic algorithm's result. The next two columns state the number of instances for which each algorithm performed best, where the slower local search algorithm is only taken into account in the latter column. For the sake of brevity, we only quote results for $k = 2$ and $k = 5$ vehicles, as those for $k \in \{3,4\}$ fall in between as expected.

For very short jobs (Tab.1), the disjoint domains algorithm (`disjoint`) consistently outperforms all other approaches for few as well as many vehicles. Only for $k = 5$, the insertion algorithm with local search (`insert l/s`) comes close. For slightly longer jobs (Tab.2), this begins to change: For $k = 2$, the advantage of `disjoint` over the other basic algorithms narrows, and `insert l/s` already outperforms it on twelve of the twenty instances. For more vehicles, the solution quality of `disjoint` deteriorates, with the clustering algorithm (`cluster`) performing best out of all basic approaches.

In the following, `cluster` consistently beats all other basic algorithms (Tab.3– 5), while `disjoint` remains competitive only for medium length jobs and $k = 2$. Even if the basic insertion algorithm (`insert`) is never the best, it performs reasonably well on all sets of instances. Moreover `insert l/s`, with an added local search, computes the best average solutions for all instances but those with very short jobs (Tab.1).

	$k=2$			$k=5$		
	∅	#best	#best l/s	∅	#best	#best l/s
`disjoint`	1.00	20	20	1.00	20	13
`cluster`	1.39	0	0	1.36	0	0
`insert`	1.37	0	0	1.34	0	0
`insert l/s`	1.21	-	0	1.02	-	7

Table 1: Makespan comparison for instances with job lengths 0–5%.

	$k=2$			$k=5$		
	∅	#best	#best l/s	∅	#best	#best l/s
`disjoint`	1.00	20	8	1.28	0	0
`cluster`	1.15	0	0	1.00	16	0
`insert`	1.20	0	0	1.07	4	0
`insert l/s`	0.98	-	12	0.74	-	20

Table 2: Makespan comparison for instances with job lengths 0–15%.

	$k=2$			$k=5$		
	∅	#best	#best l/s	∅	#best	#best l/s
`disjoint`	1.05	5	0	2.18	0	0
`cluster`	1.00	15	0	1.00	20	0
`insert`	1.14	0	0	1.19	0	0
`insert l/s`	0.89	-	20	0.86	-	20

Table 3: Makespan comparison for instances with job lengths 0–30%.

	$k=2$			$k=5$		
	∅	#best	#best l/s	∅	#best	#best l/s
`disjoint`	1.33	0	0	2.89	0	0
`cluster`	1.00	20	0	1.00	20	0
`insert`	1.16	0	0	1.24	0	0
`insert l/s`	0.91	-	20	0.91	-	20

Table 4: Makespan comparison for instances with job lengths 0–50%.

	$k=2$			$k=5$		
	∅	#best	#best l/s	∅	#best	#best l/s
`disjoint`	1.59	0	0	3.50	0	0
`cluster`	1.00	20	4	1.00	20	6
`insert`	1.21	0	0	1.33	0	0
`insert l/s`	0.97	-	16	0.96	-	14

Table 5: Makespan comparison for instances with job lengths 0–100%.

Finally, in Tab. 6– 8, we compare the reduction in makespan achieved by increasing the number of vehicles. For each algorithm and different sets of instances, we state the average speedup factors achieved by multiple vehicles, i.e., 2.00 means the average makespan could be halved as compared to the optimal tour with one vehicle. Obviously, k is an upper bound on the speedup by k vehicles.

Now, the inherently different natures of our algorithms become visible: While `disjoint` achieves

job length (%)	$k=2$	$k=3$	$k=4$	$k=5$
0–5	1.81	2.05	2.24	2.27
0–15	1.62	1.84	1.94	1.88
0–30	1.44	1.32	1.47	1.39
0–50	1.17	1.06	1.18	1.10
0–100	1.03	0.99	1.02	0.99

Table 6: Reduction in makespan over the optimal tour for one vehicle when using algorithm `disjoint`.

job length (%)	$k=2$	$k=3$	$k=4$	$k=5$
0–5	1.30	1.50	1.61	1.66
0–15	1.41	1.83	2.15	2.41
0–30	1.51	2.04	2.55	3.02
0–50	1.55	2.12	2.65	3.19
0–100	1.63	2.24	2.85	3.47

Table 7: Reduction in makespan over the optimal tour for one vehicle when using algorithm `cluster`.

job length (%)	$k=2$	$k=3$	$k=4$	$k=5$
0–5	1.32	1.48	1.75	2.14
0–15	1.36	1.72	2.06	2.26
0–30	1.33	1.76	2.16	2.55
0–50	1.33	1.78	2.24	2.57
0–100	1.35	1.81	2.19	2.62

Table 8: Reduction in makespan over the optimal tour for one vehicle when using algorithm `insert`.

the best speedup for very short jobs on any number of vehicles, it performs very poorly for average and longer jobs (Tab.6). The more jobs cross domain borders, the less efficient this approach becomes.

For all but the shortest jobs, `cluster` consistently achieves the best speedups (Tab.7). Especially when instances contain very long jobs, this algorithms utilizes extra transport capacity in form of additional vehicles most effectively. Algorithm `insert` again delivers acceptable speedups across all instances (Tab.8), however outperforming `cluster` only for very short jobs and $k \in \{4,5\}$.

5 Conclusions

In this paper, we proposed three very fast heuristic algorithms for 1D VEHICLE SCHEDULING. Our computational results demonstrate that the most straightforward approach of scheduling vehicles within their own disjoint domains is only suitable for instances with very short jobs and few vehicles, while its performance deteriorates quickly with increasing job length. The best results for longer jobs and especially many vehicles are realized by clustering jobs, scheduling clusters by the exact algorithm for one vehicle, and scheduling multiple vehicles to execute clusters in parallel.

Finally, the arguably most involved approach of inserting jobs into schedules in parallel while avoiding conflicts performs reasonably well in all scenarios. In situations when the runtime of the algorithm is not critical, this approach can be equipped with a local search technique, yielding better results than any of the basic algorithm in virtually all cases.

Open Questions The algorithms in this paper are geared towards very short runtimes. It would be interesting if significantly improved results can be obtained by more elaborate algorithms when allowing for more computation time. In this context, it also seems challenging to study the mixed integer programming formulation of the problem mentioned in Sect.3.1 in order to devise a specialized solution scheme for it.

Furthermore, we propose studying 1D-VS from an approximation perspective. While we have shown the problem to be *NP*-hard, we conjecture that approximation schemes for 1D-VS are possible.

References

[1] Raymond K. Cheung, Chung-Lun Li, and Wuqin Lin. Interblock crane deployment in container terminals. *Transportation Science*, 36(1):79–93, 2002.

[2] Shawn Choo, Diego Klabjan, and David Simchi-Levi. Multiship crane sequencing with yard congestion constraints. *Transportation Science*, 44(1):98–115, 2010.

[3] Greg N. Frederickson, Matthew S. Hecht, and Chul E. Kim. Approximation algorithms for some routing problems. In *SFCS '76: Proceedings of the 17th Annual Symposium on FOCS*, pages 216–227, Washington, DC, USA, 1976. IEEE Computer Society.

[4] Gary Froyland, Thorsten Koch, Nicole Megow, Emily Duane, and Howard Wren. Optimizing the landside operation of a container terminal. *OR Spectrum*, 30:53–75, 2008.

[5] Paul C. Gilmore and Ralph E. Gomory. Sequencing a one state-variable machine: A solvable case of the traveling salesman problem. *Operations Research*, 12(5):655–679, 1964.

[6] Udo Heinrichs and Christoph Moll. On the scheduling of one-dimensional transport systems. Technical Report 277, Mathematisches Institut, Universität zu Köln, 1997. `http://www.zaik.uni-koeln.de/~paper/unzip.html?file=zpr97-277.ps`.

[7] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[8] Eugene L. Lawler, Jan K. Lenstra, Alexander H. G. R. Kan, and David B. Shmoys. *The Travelling Salesman Problem*. Wiley, 1985.

[9] W. C. Ng and K. L. Mak. Quay crane scheduling in container terminals. *Engineering Optimization*, 38:723–737, 2006.

[10] Marcello Sammarra, Jean-François Cordeau, Gilbert Laporte, and Maria F. Monaco. A tabu search heuristic for the quay crane scheduling problem. *Journal of Scheduling*, 10:327–336, 2007.