Acta Cybernetica **21** (2013) 53–73.

Application Oriented Variable Fixing Methods for the Multiple Depot Vehicle Scheduling Problem

Balázs Dávid* and Miklós Krész†

Abstract

In this article, we present heuristic methods for the vehicle scheduling problem that solve it by reducing the problem size using different variable fixing approaches. These methods are constructed in a way that takes some basic driver requirements into consideration as well. We show the efficiency of the methods on real-life and random data instances too. We also give an improved way of generating random input for the vehicle scheduling problem.

Keywords: vehicle scheduling, variable fixing

1 Introduction

The multitude of problems arising in public transportation forms a complex system. These problems can be categorized into two main groups: vehicle scheduling problems and driver scheduling problems. However, these two sets can not be considered totally independently of each other, as vehicle schedules are needed as the basis of constructing driver schedules. Literature addresses the hierarchy between these tasks either by using an integrated or a sequential approach.

Integrated models for the combined vehicle- and driver scheduling problem have become more and more efficient lately [7, 11], and based on them, useful methods exist for the long-term planning of a transportation company's schedule. However, these solutions are still not fast enough for being considered as part of an interactive decision support system.

A sequential approach of the optimization problem breaks it down into a series of different tasks, which are solved one after the other. Such a system can have many different sub-problems, as the one seen in [4], but usually 3 phases have to be carried out: vehicle scheduling, driver scheduling and driver rostering. Fast methods that give efficient solutions exist for all these sub-problems. The main drawback is that the majority of these methods consider solving a stand-alone problem only, and does not deal with its integration into a larger system. However, breaking down the main problem into different smaller tasks results in a flexible

*University of Szeged, E-mail: davidb@inf.u-szeged.hu
†University of Szeged, E-mail: kresz@jgypk.u-szeged.hu

DOI: 10.14232/actacyb.21.1.2013.5

system, so this approach can be an ideal choice to be used for interactive decision support.

In this paper, we will deal with the vehicle scheduling problem as part of such a system. We give a brief overview of the problem itself, and analyze the requirements for real-life applications. We examine a heuristic by Gintner et al. [12], and develop new algorithms using their idea as a basis. We provide test results for all examined methods on real life and random data as well. We also introduce a new way of creating random data based on a method by Carpaneto et al [9]. This new algorithm is needed because the structure of the instances generated by the method in [9] is different from real-life instances of Hungarian transportation companies. We wanted to test our methods on random data that more closely resembles the structure of real-life Hungarian instances.

2 The Vehicle Scheduling Problem

In this section, we present the vehicle scheduling problem, and give an overview of its important models and methods from literature.

Let V be the set of vehicles, and T the set of trips. For each $t \in T$ trip, let $dt(t)$ and $at(t)$ be its departure and arrival time, and let $sl(t)$ and $el(t)$ denote its starting and ending location respectively. Two trips $i, j \in T$ are compatible if there is enough time between $at(i)$ and $dt(j)$ to cover the distance between $el(i)$ and $sl(j)$. A vehicle schedule can contain compatible trips only. A vehicle task is called a deadhead trip if the vehicle changes its location without executing a trip.

The aim of the vehicle scheduling problem is to assign vehicles to execute the trips of a given timetable. These assignments must satisfy certain conditions:

- Every trip must be executed exactly once.
- Trips assigned to a vehicle must be compatible with one another.
- The cost of the assignment must be minimal.

We examine the possible components of the cost function in the following subsection.

2.1 Costs of a Vehicle Schedule

One of the most important steps is to determine the costs arising in our problem. There are several operational costs that can be taken into consideration for a vehicle:

- **Trip distance cost:** the cost of a vehicle to cover a unit distance (usually 1 km) while executing a trip.
- **Deadhead distance cost:** the cost of a vehicle to cover a unit distance while executing a deadhead trip.

- **Daily cost of a vehicle**: the cost of making a vehicle available for use on the given day.

Trip and deadhead distance costs can easily be determined, as the gas consumption of the vehicles is a known parameter. Daily costs are more difficult to give, as they incorporate a number of different smaller costs (eg. maintenance, upkeep, etc.), but the cost of renting a vehicle of the same type for one day can be a good estimate. Using these operational costs only, the problem can be solved in different ways:

- If only the daily costs are used in the objective, the number of vehicles will be minimized.
- If only the deadhead costs are included, the amount of deadhead trips will be minimized, thus minimizing the location changes.
- To get an overall operational cost, all of the above costs have to be included in the objective function.

As we mentioned earlier, the solution of the vehicle scheduling problem alone is just the basis of a daily schedule of a transportation company. Using the above defined operational costs only does not necessarily give a good solution if we want to use it in an "application oriented" way. The vehicle schedules have to be assigned to drivers, who also have a set of constraints for their driver schedules. Driver rules have many different types, and the regulations also vary from country to country, and from company to company. One of the most important constraints that all of them have in common is the limitation of the maximum continuous driving time of a driver. Because of this, vehicle drivers must be assigned a break after a fixed amount of driving time at most.

If the solution of the vehicle scheduling problem produces too "dense" schedules (which do not have enough gaps between tasks to assign these breaks), then the driver scheduling algorithms have to transform the schedules. This means extra computation time for the driver algorithm, and it also means that the underlying vehicle schedule will also be changed significantly, modifying its cost retroactively. This aspect must also be considered, when solving the vehicle scheduling problem.

2.2 Single and Multiple Depot

In some cases the vehicles can also be classified into depots. A depot of a vehicle can mean its starting geographical location, but it can also represent its vehicle type. In both cases a depot-compatibility vector is given for every trip that shows the types of vehicles able to execute the trip.

Complexity of the vehicle scheduling problem depends on the number of vehicle depots. If all vehicles belong to the same depot, the problem is a single depot vehicle scheduling problems (SDVSP). An SDVSP can be modeled as a minimum-cost flow problem, thus even large-size (several thousand trips) instances are polynomially solvable to an optimum. Such a formulation is given in [5].

However, real life instances usually have 2 or more depots. These types of problems are called multiple depot vehicle scheduling problems (MDVSP). The MDVSP was first formulated by Bodin et al. [6], and its NP completeness was proven by Bertossi et al. [3].

Solving the MDVSP results in a set of vehicle schedules, each assigned to a vehicle from one of the depots. For every pair of trips i and j on any schedule, they have to be compatible, and their depot-compatibility has to match the depot of the vehicle assigned to the schedule. However, the solution itself can be constructed with regards to many different costs and constraints.

2.3 Models and Methods for the MDVSP

Literature discusses three main types of models for the MDVSP: single commodity, set partitioning, and multi-commodity. An overview of the different approaches can be found in [8]. In this paper, we will be dealing with the multi-commodity approach, which uses multi-commodity network flow models. In these models, every depot has its own commodity layer in the network, and only those trips are considered at a given layer, which can be executed from the corresponding depot.

The *connection-based network* gives every possible connection between the trips of the problem. To define the problem, the following additional notations have to be introduced: The set of depots that can execute a trip t is denoted by $g(t)$. Let $T_d \subseteq T$ be the set of trips that can be executed from depot d. Similar to the trips, let every d depot have a starting location $sl(d)$ and ending location $el(d)$. The set of nodes of our network will be the following:

$$N = \{dt(t) \cup at(t) \cup sl(d) \cup el(d) | t \in T, d \in D\}.$$

Let

$$A^d = \{(dt(t), at(t)) | t \in T_d\}$$

be the set of trips that can be served by depot d, and let

$$B^d = \{(at(t), dt(t')) | t, t' \in T_d \textit{ are compatible}\}$$

be the possible deadhead trips of depot d.
Let

$$P^d = \{(sl(d), dt(t)), (at(t), el(d)) | t \in T_d\}$$

be all the pull-in and pull-out edges of depot d.
The above sets give us the set of edges of the connection based network:

$$E = A^d \cup B^d \cup P^d \cup \{(el(d), sl(d))\} \textit{ for every } d \in D.$$

Based on the sets introduced above, a solution of the MDVSP can be determined by using the network (N, E). We define an integer vector x for every edge of the network. A vector component belonging to an edge e of depot d is denoted by x_e^d. The problem can be formalized in the following way:

$$\sum_{d \in g(t)} x^d_{dt(t),at(t)} = 1, \forall t \in T \tag{1}$$

$$\sum_{e \in n^+} x_e^d - \sum_{e \in n^-} x_e^d = 0, \forall d \in D, \forall n \in N \tag{2}$$

$$x_e^d \in 0, 1, \text{ except for } (el(d), sl(d)) \tag{3}$$

where n^+ is the set of outgoing, and n^- is the set of incoming edges from node n.

According to constraint (1) each trip has to be executed exactly once, while (2) means that every vehicle arriving to a geographical location has to leave that location. Constraints for the number of vehicles in each depot can be set as an upper bound for the circulation edges of the corresponding depot. Any flow satisfying the above conditions can give a feasible solution of the problem. For an optimal solution, we have to minimize

$$\sum_e c_e x_e,$$

where c_e is the cost of edge e.

The main drawback of the connection based model comes from its size. The number of compatible trips is high, even with a small number of trips in the problem, and this results in a large number of possible deadhead trips. This makes the size of the problem so large that it can not be used effectively on real-life data, where the number of trips is a couple of thousands usually.

The time-space network has been introduced to vehicle scheduling by Kliewer et al. [13]. It eliminates the drawback that comes from the size of the connection-based network. This way, it is possible to solve larger-sized real-time MDVSP instances efficiently. As we described earlier, the number of edges connecting compatible trips in the connection based model is high, but only a few of these are actually used in a feasible solution. However, if we left any of these connections from the model, we would lose the optimality of the solution.

The time-space network efficiently reduces the number of edges. The model arranges data in two dimensions: time and space. Space represents the set of geographical locations, while the timelines at each location represent a sequence of events. The arrival and departure times of the tasks are denoted on the corresponding timelines, and give the nodes of the model.

The time-space network is constructed using the above given nodes. Apart from this difference, the set N of nodes of the network can be defined similar to the connection based approach. The definition of A^d is also similar for each depot $d \in D$

and P^d is given with the help of the time-lines associated with the corresponding depot.

The definition of deadhead trips is the other main difference between the two models. The timelines used by the time-space network can be used to aggregate deadhead trips by introducing so-called waiting edges. These edges connect adjacent nodes on the timeline. This method reduces the size of the problem significantly. Waiting edges always connect two adjacent nodes on the appropriate timeline. Denoting the set of waiting edges with W^d for every depot $d \in D$, the set of edges of the time-space network is:

$$E = A^d \cup B^d \cup P^d \cup W^d \cup \{(el(d), sl(d))\} \textit{ for every } d \in D.$$

Using these, the IP model of the time-space network can be given in a similar way to the connection based network:

$$\sum_{d \in g(t)} x^d_{dt(t),at(t)} = 1, \forall t \in T \tag{4}$$

$$\sum_{e \in n^+} x^d_e - \sum_{e \in n^-} x^d_e = 0, \forall d \in D, \forall n \in N \tag{5}$$

$$x^d_e \geq 0, \tag{6}$$

$$x^d_e \text{ integer} \tag{7}$$

Considerably bigger instances can be solved to optimality using the time-space network. However, the running time can still be an issue, especially in the case of larger real-life instances. Literature provides a variety of methods for solving large MDVSP. Selected heuristics based on both mathematical programming and combinatorial aspects can be found in [14]. However, these solve the standalone vehicle scheduling problem, and do not consider the "application oriented" structure of the vehicle schedules discussed above. We will present a solution method that takes this aspect into consideration also.

In [10], we present a collection of heuristic methods for the MDVSP found in literature, and also analyze the "application oriented" usefulness of their results. Paper [10] partially studied the idea of variable fixing too, which research later became the basic idea for this paper.

3 Reducing the MDVSP model size

In this section, we will give a heuristic to solve the MDVSP, taking into consideration both operational costs, and the "application oriented" structure of the vehicle schedules. A solution with short running time and well structured schedules is important for an interactive decision support system.

Gintner et al. [12] propose a two-phase heuristic to solve large instances of the MDVSP by decreasing its model size. A number of variables of the model are fixed,

and the resulting new problem is solved to optimality afterwards. Because of these two steps, they call this approach fix-and-optimize. The idea behind their heuristic is to solve a number of simplified models of the original problem, and decide on the variables to be fixed based on their results. This is done by finding series of trips that are common in all solutions. If such trips are found, it is presumed that they are likely to appear in the global optimum in the same way. These trips are called stable chains, and are used as single trips in the model of the MDVSP.

Their method decomposes the original MDVSP into an SDVSP for every depot. This problem is constructed and solved in the following way:

- The capacity of the depot is equal to the sum of all depot-capacities of the MDVSP.
- Only those trips are considered, which can be executed from depot.

After the SDVSP sub-problems are solved for each depot, their solutions are used to create stable chains. If a sequence of trips appears in the same order in all solutions, then they are considered as a chain. Using these stable chains as single trips, a smaller MDVSP model is built that has the following properties:

- The number and capacity of the depots are the same as in the original problem.
- The set of trips of the new problem consists of the trips that are not included in any of the stable chains, and a newly created trip for each stable chain. Their costs are the sum of all the trips these chains represent. The departure time and starting location of the first trip of the chain, and the arrival time and ending location of the last trip of the chain are used for this new trip as starting and ending data. These trips can be executed from any depot.

After this new MDVSP is solved, the trips in the stable chains have to be substituted back instead of the new trips, to acquire the final solution.

We chose to develop a heuristic based on the idea of variable fixing, as it can model the "application oriented" aspect of the problem. This is done by fixing trips in the same chain "that should belong together in the final solution". We can also control the amount of time between two consecutive trips of a chain by not adding a possible trip to a chain if that would leave too little, or too much gap in between.

As the basis of the method, we solved a simplified model of the problem that we call a "quasi-multiple depot" model. Though we use only a single depot in this model, two trips are connected only if they would be connected in the multiple depot case as well. This means that the trips have to be compatible, and they must also share a common depot from which they can be served. The cost of the arc between these two trips in our model is calculated using the cheapest possible cost of all their common depots. The capacity of the depot is the sum of the capacities of all depots in our original problem. Pull-out and pull-in arcs of the depot have the weight of the minimal deadhead trip from and to any of the depot locations of

the original problem. Once this "quasi-multiple depot" model is constructed, it is solved by an MILP solver.

We experimented with three different approaches for finding stable chains in the solution of the above problem:

- Build chains with regards to depot costs.
- Fix trips with the same depot-compatibility in a chain.
- Assign trips of the same bus-line to a chain.

The methods will be presented in the following subsections. We also illustrate on a small example their difference in building chains.

3.1 Building chains using depot costs

This was the first heuristic we developed for solving the MDVSP. Depots are ordered into a list increasingly according to the following cost:

$$\frac{1}{\epsilon} * cost(daily) + cost(km)$$

where $cost(daily)$ is the daily cost of a single vehicle from the depot, $cost(km)$ is the cost of that vehicle to travel 1 km, and $\epsilon > 0$ is a parameter.

For every depot in this order, the algorithm examines all the vehicle schedules in the result of the "quasi-MDVSP". If subsequent trips are found which can be executed from this depot, they are considered together in stable chains. These trips are flagged, and cannot be the part of other stable chains. The description of this algorithm can be seen in Algorithm 1.

This algorithm is only the basis of finding the chains, further constraints can be introduced:

- We can give a limit to the number of trips in the chains.
- The length of the chains can be maximized.
- The minimum/maximum gap in time between two trips of the chain can be given.

Experience shows, that limiting the length of the stable chains with the above constraints results in a solution with better cost, but has an increase in running time. The running time of the heuristic was very fast, but the quality of the solutions was far from what we have expected. Because of this further changes have been experimented with to improve the cost of the solution, with a minimal increase in the running time.

Algorithm 1 Variable fixing using depot costs.

```
1: Determine the order D of depots
2: S ← ∅, V ← ∅, L ← ∅
3: for each d ∈ D do
4:   for all trips j ∉ V the solution do
5:     while j can be executed from d do
6:       L ← j
7:       V ← j
8:       j = nexttrip(j)
9:     end while
10:    if |L| > 1 then
11:      S ← L
12:    end if
13:    L ← ∅
14:  end for
15: end for
16: return S
```

3.2 Building chains based on depot-compatibility

Using our experience gained from the method presented above, we tried to find a way to fix trips that have some property in common instead of using a cost function. In our second method, we tried to construct stable chains based on similar depot-compatibilities of the trips. We first examined those trips from the solution of the "quasi-MDVSP" that can be executed from all d depots. Then all that are compatible with $d-1$ depots, and so on. Two subsequent trips are assigned to the same chain if they have exactly the same depot-compatibility in the solution. This algorithm is described in Algorithm 2. The same additional extra constraints that we have shown at Algorithm 1 can also be introduced here.

3.3 Building chains using trips of the same bus-line

This method of constructing the stable chains is closer to the schedule building practice of transportation companies. A driver usually uses the same vehicle during his shift, and he is carrying out consequent trips of the same bus-line. Some changes might occur in his schedule, but their number remains low. However, when solving the MDVSP using an MILP solver, the resulting vehicle schedules usually have a high number of line changes. Though this can not be modeled in costs directly, it puts some load on the driver itself.

We tried to build stable chains using this as a guideline. We fixed those trips in a chain only that belong to the same bus-line. We also set a maximum time limit of the gap between two such trips. If two subsequent trips belong to the same bus-line, but are far from each other in time, then they are not fixed in the same chain. The description of this method can be seen in Algorithm 3.

Algorithm 2 Variable fixing based on depot compatibilities.

$S \leftarrow \emptyset, V \leftarrow \emptyset, L \leftarrow \emptyset$
for $d = numof(depots)$ **downto** 1 **do**
  for all $j \notin V$ trips compatible with exactly d depots **do**
    $L \leftarrow j$
    $V \leftarrow j$
    $k = nextrip(j)$
    while j and k have the same depot-compatibility **do**
      $L \leftarrow k$
      $V \leftarrow k$
      $j = k$
      $k = nexttrip(j)$
    end while
    if $|L| > 1$ **then**
      $S \leftarrow L$
    end if
    $L \leftarrow \emptyset$
  end for
end for
return S

Algorithm 3 Variable fixing based on same bus-lines.

$S \leftarrow \emptyset, V \leftarrow \emptyset, L \leftarrow \emptyset$
for all $j \notin V$ trips **do**
  $L \leftarrow j$
  $V \leftarrow j$
  $k = nextrip(j)$
  while $line(j) = line(k)$ and $dt(k) - at(j) < limit$ **do**
    $L \leftarrow k$
    $V \leftarrow k$
    $j = k$
    $k = nexttrip(j)$
  end while
  if $|L| > 1$ **then**
    $S \leftarrow L$
  end if
  $L \leftarrow \emptyset$
end for
return S

3.4 An illustrative example of building chains

In this subsection we show the differences between the three variable fixing methods presented above. Let us consider a vehicle scheduling problem with 3 depots, 8 trips and 4 geographical locations (A,B,C,D).

Table 1: Details of the trips.

Trip	From	To	Departure	Arrival	Depots
1	C	B	10	12	1,2,3
2	B	A	12	14	1,2,3
3	B	C	12	14	2,3
4	A	B	14	16	1,2,3
5	B	A	16	18	1,2
6	A	B	18	20	1,2
7	C	D	22	24	2,3
8	D	C	24	26	2

Table 1 gives every detail of the trips, including their start and end geographical locations, their departure and arrival times and the depots that they are compatible with. Furthermore, suppose that trips between the same geographical locations belong to the same bus-line. This gives us the following three lines:

- Line A-B: trips 2,4,5,6.
- Line B-C: trips 1,3.
- Line C-D: trips 7,8.

Let the deadhead distance between any pair of geographical locations, and the pull-in and pull-out distance for all depots be 2 minutes. The depot costs of the problem are the following:

- Depot 1: 100 daily cost and 10/minute distance cost
- Depot 2: 200 daily cost and 20/minute distance cost
- Depot 3: 300 daily cost and 30/minute distance cost

The structure of the above problem can be seen on Figure 1. The horizontal lines represent the geographical locations, while the arrows between them correspond to the trips. All three heuristics solve a quasi-multiple depot problem, which results in the following two vehicle schedules:

- Schedule 1 executes trips 1,2,4,5,6.
- Schedule 2 executes trips 3,7,8.

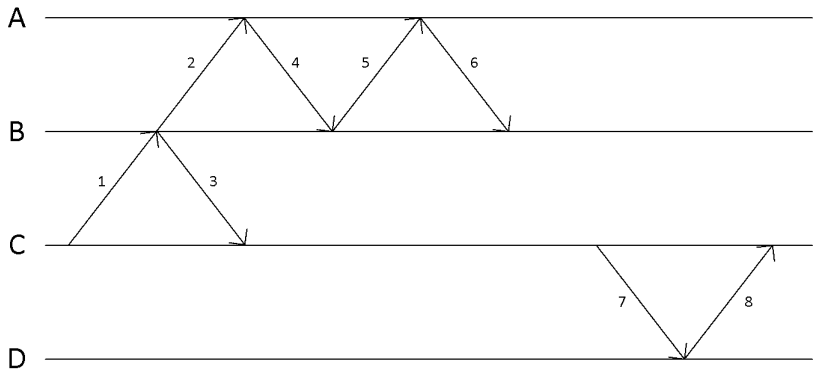

Figure 1: The structure of the problem

The heuristics will try to construct chains based on these schedules. Applying the *method based on depot costs*, the cost function will give the depot order 1,2,3 for an arbitrary $\epsilon > 0$. Using this order, the following chains are constructed:

- Chain 1: trips 1,2,4,5,6.
- Chain 2: trips 3,7,8.

If we build the chains with regards to *depot-compatibility*, first we examine trips that are compatible with all 3 depots, then the trips compatible with 2 depots, and finally the trips that are compatible with 1 depot only. This results in the following chains:

- Chain 1: trips 1,2,4.
- Chain 2: trips 5,6.
- Chain 3: trips 3,7.
- Chain 4: trip 8.

Considering *bus-lines* when building the chains, we have to examine all 3 bus-lines in the schedules. The method constructs the following chains:

- Chain 1: trips 2,4,5,6.
- Chain 2: trip 1.
- Chain 3: trip 3.
- Chain 4: trips 7,8.

4 Random instances

Though we were provided with the real-life instances of the transportation company of Szeged, that gives us only a limited database. As it is difficult to access real-life instances from other companies, the easiest way to acquire more test data that has the properties of the real-life input is to use an algorithm that generates it based on our needs. Many papers from literature that deal with the vehicle scheduling problem present the efficiency of their methods on random instances generated according to an algorithm by Carpaneto et al. [9]. However, our test experience shows that the structure of the data generated by their method was different from real-life instances in many aspects. Because of this, we propose an improved way of generating random data in this section. First, we describe the above method, and then give a new variation for it.

4.1 Generation method by Carpaneto et al.

The input of the algorithm is the n number of trips, and the m number of depots. The number of geographical locations is uniformly chosen from the interval $[\frac{n}{3}, \frac{n}{2}]$, their locations are chosen in a uniform random way on a $60*60$ grid. The deadhead trips between geographical locations p and q correspond to their $d(p,q)$ Euclidean distance.

The properties of every t_i trips is determined using the above. The start and end $sl(t)$ and $el(t)$ geographical locations are chosen uniformly from $[1, f]$. These locations determine the length of the trip, $d(sl(t), el(t))$. Trips can have two types: short trip, or long trip.

There is a 40% chance that a t trip becomes a short trip. Its $dt(t)$ departure time is also chosen randomly:

- with a 15% chance uniformly from [420,480]
- with a 70% chance uniformly from [480,1020]
- with a 15% chance uniformly from [1020,1080]

The $at(t)$ arrival time of a short trip is chosen uniformly from the interval $[dt(t) + d(sl(t), el(t)) + 5, dt(t) + d(sl(t), el(t)) + 40]$.

Long trips are generated with a 60% chance. Their $dt(t)$ departure time is chosen uniformly from $[300, 1200]$, while their $at(t)$ arrival time is chosen uniformly from $[dt(t) + 180, dt(t) + 300]$. Long trips have the same start and end location, which means that a value is assigned to $sl(t) = el(t)$ uniformly from $[1, f]$.

They also presented a possible placement of the depots for $m = 2, 3$. The number of vehicles in each depot is determined uniformly from $[3 + \frac{n}{3m}, 3 + \frac{n}{2m}]$.

4.2 Our new generation method

Our experience showed that the instances generated using the above method were very differently structured from the real-life data we were dealing with. We decided

to modify this method to become closer to those real-life instances. The main difference of this model from our data was that trips had no pre-assigned depot-compatibilities. For this, we introduced an additional input: a p_i probability for every $1 \leq i \leq m$ depot.

The p_i value gives the probability that a trip can be executed from depot i. When the trips are generated, they are assigned a $\mathbf{v} = (v_1, ..., v_m)$ depot-compatibility vector. For every v_i,

$$v_i = \begin{cases} \text{true} & \text{with } p_i \text{ probability} \\ \text{false} & \text{otherwise} \end{cases}$$

If all components of the $\mathbf{v}$ recieve false values, then exactly one of them is set as true. This is also decided using the given probabilities. A trip can be executed only from those depots, whose corresponding components have a true value.

Analyzing the trips of the original generator, we found that the average length of the trips was too high compared to our real-life data, and the trips were scattered geographically. To address this, we introduced some further changes.

The number of generated geographical locations was also very high compared to the number of trips, and two trips rarely followed each other at the same location in a small timeframe. After experimenting, we found the $[\frac{2n}{25}, \frac{3n}{25}]$ interval that gives an acceptable number of locations. However, because of the decreased number of geographical locations, we also had to decrease the area they are generated at. We used a $30 * 30$ grid for this.

To address the problem of the too long average length of the trips, we slightly modified the generation of the trips as well. The ratio of the long and short trips has been exchanged, and we generated short trips with a 60% chance, and long trips with only a 40% chance.

The length of the trips has been decreased. The $at(t)$ arrival time of a short trip is chosen uniformly from the interval $[dt(t)+d(sl(t), el(t)), dt(t)+d(sl(t), el(t))+20]$, while the $at(t)$ arrival time of a long trip is chosen uniformly from $[dt(t)+40, dt(t)+60]$.

Using the modification above, the random generated instances we received resembled more closely to the real-life data we were provided with by the transportation company of Szeged city.

5 Test Results

The different variable fixing approaches discussed earlier were tested on real-life data instances from the city of Szeged, Hungary, as well as on random data generated by an algorithm described in the previous section. We present their results below.

5.1 Real-life instances

Our real-life instances were taken from the transportation company of the city of Szeged, Hungary. The company uses 11 different day-types (called combinations) over its planning period. To get a complete schedule for a normal planning period (which is 2 months in the case of the company), the vehicle scheduling problem has to be solved for all 11 combinations. The daily driver schedules will depend on the combination type of the corresponding day. Of all the combinations 4 have been selected as test cases. The properties of their optimal solutions can be seen in Table 2. The combinations with higher number of trips (*szeged1* and *szeged4*) are workdays of the week, while *szeged2* and *szeged3* are instances taken from a Sunday and Saturday respectively.

The running time given in the table is the running time in seconds needed to find an optimal solution using the SYMPHONY solver on the time-space network model of the problem.

Table 2: Optimal solution of the instances.

Instance	Day type	Running time(s)	Vehicles	Dense schedules
szeged1	Weekday	872	96	4
szeged2	Sunday	431	44	3
szeged3	Saturday	250	55	6
szeged4	Weekday	1179	96	3

As it is visible, the running times of the weekday instances can reach 20 minutes, and solving all the 11 combinations of the company to optimality would take about 8500 seconds. The 2-2,5 hours of running time for calculating the vehicle schedules is not acceptable from the perspective of a decision support system, as there are the additional driver schedules and rosters that still have to be calculated for the whole planning period (which is usually several weeks or months).

We will examine three aspects of the results given by the heuristics:

- The gap in cost of the result from the optimal solution of the MDVSP.
- The ratio of the running time of the heuristic compared to the running time of the IP solver.
- The structure of the schedules.

When vehicle schedules are used as an input in driver scheduling, different driver constraints have to be fulfilled. The most important of these are the maximum consecutive driving time without any rest, and the total length of the schedule given to a driver. We analyze the structure of the vehicle schedules using these aspects. If a schedule violates any of the mentioned constraints, it is labelled as a "dense" schedule.

The results of the variable fixing heuristic of Gintner et al. can be seen in Table 3. Every solution shows a decrease in running time: the average running time of the instances is about 40% of the original, which would mean a running time of 3000-3500 seconds (almost 1 hour) for all the vehicle schedules. The gap from the optimum varies between 0,25%-0,40%.

Table 3: Solution of the variable fixing heuristic of Gintner et al..

Instance	Gap from opt.	Running time ratio	Vehicles	Dense schedules
szeged1	0,27%	57,45%	96	6
szeged2	0,41%	31,55%	44	3
szeged3	0,37%	42,80%	55	6
szeged4	0,25%	35,69%	96	4

The heuristic that uses a depot-cost function for building the chains decreases the running time to around 8,8% of the time needed for the IP solver. This means that a solution is obtained in a couple of minutes (which is at most 4-5 in all test cases). The total running time for all 11 combinations is between 10-15 minutes, which is really good. However, the gap from the optimal solution has risen significantly: in some cases, it was greater than 2,5%. As opposed to the variable fixing heuristic, the greedy method fixes significantly more trips ($\sim$ 66% in comparison with $\sim$ 33%) into stable chains, which greatly reduce the size of the problem. However, the method is less precies because of the fact that more trips are fixed in chains. Limiting the chain construction with the before mentioned alternative constraints (e.g. limit the size/length of the chains, or the types of chosen trips) will lead to a solution with a better cost. On the other hand, less fixed trips also mean a greater problem size, which results in an increase in running time. The results of this method can be seen in Table 4.

Table 4: Solution using chains based on depot costs.

Instance	Gap from opt.	Running time ratio	Vehicles	Dense schedules
szeged1	2,63%	9,29%	100	3
szeged2	1,20%	3,71%	46	5
szeged3	1,12%	10,40%	57	7
szeged4	2,32%	8,40%	98	3

Building the chains based on depot-compatibility shows a more ordered structure than the method discussed above. Though more trips remain single, which comes with a slight increase in running time (in average 11,63% of the IP solution, which is about 15-20 minutes for all the combinations), it is still acceptable. The gap became also significantly smaller, it is at most around 1,25%. This value can

be acceptable, though still seems a bit high. The cost can be improved in the same way as in the previous case, but this will also result in an increase in running time. However, the rate of decrease in the cost will be much smaller with the inclusion of additional constraints. The results of this heuristic can be seen in Table 5.

Table 5: Solution based on depot-compatibility.

Instance	Gap from opt.	Running time ratio	Vehicles	Dense schedules
szeged1	1,14%	12,84%	97	2
szeged2	0,34%	6,50%	44	4
szeged3	0,38%	16,00%	56	8
szeged4	1,26%	19,42%	97	2

Taking into consideration the experience of the previous solution methods and analyzing their difference from the solution of the IP solution, we decided to apply a more structural method for building the chains. Using trips of the same bus-line in a chain again leads to less fixed trips, which means an overall decrease in running time to 24,41% of the original. This results in about 30-35 minutes to solve all the combinations. On the other hand, the gap of the solutions from the optimum is very favourable, not more than 0,60% in any of the instances, but there are much lower ones around 0,20%, or even below. The results of this method are found in Table 6.

Table 6: Solution using chains based on bus-lines.

Instance	Gap from opt.	Running time ratio	Vehicles	Dense schedules
szeged1	0,58%	16,28%	97	3
szeged2	0,03%	21,35%	44	5
szeged3	0,23%	26,00%	55	7
szeged4	0,59%	10,86%	97	3

The solution methods give about the same number of badly structured vehicle schedules as the original IP solutions did. Using additional constraints that limit the length of the chains, the number of the trips in the chains, or the minimal/maximal idle time between two subsequent trips in a chain result in a lower number of "dense" schedules besides the smaller gap from the optimum. However, this would still affect the running time of the methods.

With the use of these "dense" schedules we tried to give a formal way of evaluating the "goodness" of the vehicle schedule structure with regards to driver scheduling. While the number of badly structured schedules stays approximately the same using any of the shown heuristics, the impact it has on a driver scheduling algorithm can be really different depending on which vehicle heuristic is used. We

experimented with several sequential vehicle and driver methods for the problem, among which our most recent research can be found in [2]. Test results show that using any of our proposed variable fixing methods gives a better cost at the end of the driver scheduling phase than using either the optimal MDVSP solution, or the heuristic of Gintner et al.

5.2 Solutions on random data input

As we mentioned earlier, we also tested the algorithms on random data instances. We tried different problem sizes with 50, 250, 500 and 1000 trips respectively. Out of the 4 methods above, the heuristic of Gintner et al. failed to find any chains in all cases, while our heuristic using bus-lines rarely fixed any trips, and it always fixed only less than 5 trips using these inputs. This means that both methods ended up solving the original (or almost exactly the original) MDVSP, and thus their results can not be analyzed properly.

The heuristic of Gintner et al. needs a large number of trips in the input that can be executed from any of the depots. Besides this, these trips have to be close enough to one another so that every solved SDVSP sub-problem schedules them in the same sequence. If the trips that are compatible with every depot are scattered on the timeline of the problem, then none or only some of the trips will be fixed in chains. This scenario is likely to happen in the proposed random instances, which explains the failure of the heuristic in finding chains.

The method based on bus-lines has the same problem on this randomly generated input. Real-life instances have different bus-lines, which roughly mean that there are given p and q geographical locations, between which trips occur back and forth with a given frequency. Random generated instances will not have this kind of order in their timetable, thus this heuristic is likely to fail too.

The results of the other two heuristics can be seen in Table 7. The column marked with (cost) represent the heuristic that uses a cost function, while the column marked with (depot) give results for the heuristic based on depot-compatibility. The heuristic using a cost function arrived at about the same results, as on the real-life instances, while the heuristic based on depot-compatibility also fixed fewer trips than usual. As this method also depends on trips sharing the same depot, it also has a more difficult time finding chains.

Table 7: Solution on random instances.

Instance	Gap (cost)	Gap (depot)
random_50	0%	0%
random_100	0%	0%
random_500	1,57%	$9 * 10^{-6}$%
random_1000	1,54%	0,02%

Test experience on the random instances show that the data generated by our

method is still different from real-life instances in some structural aspects. Heuristics that are based on structural properties appearing in real-life data (Gintner et. al, bus-lines) can not be applied effectively to most of the generated input. This means that we need a method that models the properties of real-life instances more closely. The two main parts that have to be improved in the random generator are the possibility to create entire bus-lines randomly, and to model the depot-compatibility distribution of trips in a better way.

6 Conclusions and Future Work

We examined the vehicle scheduling problem and its existing models and methods in literature, to find one that fits the framework of an interactive decision support system. Such a method is important, as transportation companies do not only need to have an efficient long term planning software, but they must also have a way to assist important decisions and give suggestions in a reasonable time. Running time was an important criterion for such a method, but the value of the solution had to stay close to the optimum as well.

We developed several solution approaches based on the core idea of an efficient heuristic by Gintner et al. Each of our solution approaches became more refined as the previous one, as their results were analyzed and taken into consideration at every step. Our final heuristic comes with both an acceptable running time, and a small gap from the optimal solution. Moreover, our test experience shows that they also work well in a sequential decision support system.

We also examined the commonly used random generator method of Carpaneto et al. However, our experience showed that the distribution of trips generated by their algorithm was very different from real-time instances. We proposed an improved version of this algorithm so that its output has a structure that is closer to real-life. Extensive testing on these instances showed that some of our presented methods also work nicely on random data as well, and produce an acceptable solution for it.

Test results show that the heuristic methods can be slightly improved. In order to do this, more analysis has to be carried out into the structure of the solutions and their differences from the schedules of the original IP solution. Further experiments can also be made with the different parameters and limiting constraints discussed in the paper.

The metric of "dense" schedules that we introduced to measure the effect of our algorithms on future driver schedules turned out to be poorly defined. A different analysis has to be carried out into the interaction of vehicle and driver schedule. We have to give another metric by identifying the exact types of schedules that are expensive to transform in the driver phase.

The random data generating algorithm also has to be improved further. As we have seen at our test cases, the structure of results on random instances differs from real-life cases in important elements. The most important of these is the inclusion of bus-lines, which are crucial to methods that take this structural property into consideration. This requires additional study of the original timetables of our

real-life cases, and tuning the parameters that affect the number and position of geographical locations, and the frequencies of the trips.

Acknowledgments.

This work was partially supported by the Szeged City Bus Company (Tisza Volán, Urban Transport Division) and Gyula Juhász Faculty of Education, University of Szeged (project no. CS-004/2012).

The second author was partially supported by the European Union and co-funded by the European Social Fund through project HPC (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0010).

References

[1] Ahuja, R.K., Magnanti, T.L., Orlin, J.B. Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, 1993.

[2] Árgilán, V., Balogh, J., Békési, J., Dávid, B., Krész, M., Tóth, A. Driver scheduling based on driver-friendly vehicle schedules. in *Proceedings of OR 2011 International Conference on Operations Research*, pages 323-328, Springer-Verlag, 2011.

[3] Bertossi, A.A., Carraresi, P., Gallo, G. On Some Matching Problems Arising in Vehicle Scheduling Models. *Networks* 17, pages 271–281, 1987.

[4] Békési, J., Brodnik, A., Pash, D., Krész, M. An integrated framework for bus logistic management: case studies. in *Logistik Management*, pages 389-411, Physica-Verlag, 2009.

[5] Bodin, L., Golden, B. Classification in vehicle routing and scheduling. *Networks* 11, pages 97–108, 1981.

[6] Bodin, L., Golden, B., Assad, A., Ball, M. Routing and Scheduling of Vehicles and Crews: The State of the Art. *Computers and Operations Research* 10, pages 63–212, 1983.

[7] Borndörfer, R., Löbel, A., Weider, S. A bundle method for integrated multi-depot vehicle and duty scheduling in public transit. *Computer-aided Systems in Public Transport*, pages 3-24, 2008.

[8] Bunte, S., Kliewer, N. An overview on vehicle scheduling models. *Journal of Public Transport* 1(4), pages 299-317, 2009.

[9] Carpaneto, G., Dell'Amico, M., Fischetti, M., Toth, P. A branch and bound algorithm for the multiple depot vehicle sheduling problem. *Networks* 19, pages 531-548, 1989.

[10] Dávid, B. Heuristics for the Multiple-Depot Vehicle Scheduling Problem. in *Proceedings of the 2010 Mini-Conference on Applied Theoretical Computer Science*, pages 23-28, 2011.

[11] Gintner, V., Kliewer, N., and Suhl, L. A Crew Scheduling Approach for Public Transit Enhanced with Aspects from Vehicle Scheduling. *Computer-aided Systems in Public Transport*, pages 25-42, 2008.

[12] Gintner, V., Kliewer, N., and Suhl, L Solving large multiple-depot multiple-vehicle-type bus scheduling problems in practice. *OR Spectrum* 27, pages 507-523, 2005.

[13] Kliewer, N., Mellouli, T., Suhl, L. A time-space network based exact optimization model for multi-depot bus shceduling. *European Journal of Operational Research* 175, pages 1616-1627, 2006.

[14] Pepin, A.-S., Desaulniers, G., Hertz A., Huisman, D. Comparison of Heuristic Approaches for the Multiple Depot Vehicle Scheduling Problem. *Journal of Scheduling* 12, pages 17-30, 2009.