Using a hybrid genetic algorithm to solve an extended Pickup and Delivery Problem

Wim Goossens, Daan Hiemstra

Abstract

This paper proposes a Hybrid Genetic algorithm for solving a new variant of the Pickup and Delivery Problem (PDP). In the PDP, different goods need to be picked up and delivered at different locations respecting capacity constraints. This problem is extended by adding time windows, a limited driving range and transfers. The Electric Multi Trip Pickup and Delivery Problem with Time Windows and Transfers (E-MTPDPTWT) is solved with a Hybrid Genetic algorithm, which is aimed at minimising the total distance travelled. The proposed algorithm is split into two phases: child generation, containing the new Sub Group Exchange Crossover (SGXX) operator, and education, which makes use of local search operators. The algorithm introduces a novel representation for the dependencies between the different trips made in the solution. This dependence forest can be used to efficiently deal with different stages in the algorithm and can also be used for many other related routing problems with transfers. Experiments performed on a set of 594 instances for the E-MTPDPTWT demonstrate that the algorithm is able to improve the solution quality of its starting solution by up to 37.5%. When no prior feasible solutions are present, the algorithm sometimes is able to find feasible solutions.

1 Introduction

The recent growth and developments of the transportation and distribution industry has resulted in rapid expansion of complexity issues. Due to an increasing demand and number of suppliers, the distribution problems are becoming larger and larger. In addition, in present-day demand, demand time is a restrictive factor and the origin of goods that are to be distributed has become more and more spread. This makes managing distribution logistics extremely difficult. In addition to these complexity issues, companies have now adopted environmental friendly business strategies, requiring the application of Green Logistics, which adds to the complexity and diversity of these distribution problems. Green logistics makes the goals and constraints of these distribution problems often very different. Electric vehicles (EV) are often used in Green Logistics, which adds the complexity of limited driving ranges and recharging to the array of transportation problems.

These complex distribution tasks are generally represented by extensions of the widely studied Pickup and Delivery Problem (PDP). The PDP is a problem that consists of a number of requests, each having an origin, known as the pickup location, and a destination, the delivery location. Adding several extensions allow inclusion of all different complexity issues such as deadlines, vehicle capacity and recharging.

In this paper, the PDP under consideration has a new combination of extensions, which will be explained in the following. The first extensions adds the time window constraint in order to include deadlines for the delivery of the requests. Secondly, adding extra vehicles with limited capacity creates the Multi Trip PDP with Time Windows (MPDPTW). The vehicles used for this problem are electric and though EV's have developed rapidly over the past years, they still are restricted by a limited driving range. The MPDPTW in a setting with electric vehicles is known as the E-MPDPTW. This recharging constraint while having access to more vehicles makes it interesting to transfer goods from one vehicle to another. Therefore, this problem can be defined as the Electric Multi Trip Pickup and Delivery Problem with Time Windows and Transfers (E-MTPDPTWT).

The complexity of the E-MTPDPTWT makes exact solution methods impossible to solve well-sized instances. Therefore, to solve this problem a Hybrid Genetic Algorithm will be proposed that aims at minimising distances travelled. With a new Sub Group Exchange Crossover (SGXX) ,the algorithm first recombines different "chromosomes" (representing trips) of two solutions in order to generate new solutions, combining good aspects of one solution with the good aspects of the other. Hereafter, resulting defects are repaired and the "child" is scheduled such that it meets all deadlines. Finally, the algorithm performs local search moves to allow them improving themselves to its full potential. This iterative evolutionary process with survival will continue until the desired solutions are found.

The transfer of goods makes different trips of vehicles dependent on each other. This makes performing operators such as a crossovers, local search moves and scheduling difficult. Therefore, a novel representation of the solution describing the dependencies is introduced by making use of a dependence forest. This representations allows the algorithm to efficiently deal with these aforementioned operations.

The performance of the algorithm has been tested on different newly designed instances and has shown prosperous results. In the remainder of this paper, we will start by sketching the context of our problem, based on existing literature. Using these findings, we will formulate our problem. After the problem has been formulated, our approach for a the algorithm that generates good solutions for the E-MTPDPTWT will be presented. Using our algorithm, we will generate results and interpret these. Then, a discussion of our research will follow. The end of this paper will contain a conclusion of all the previous and an outlook on future research.

2 Literature Review

The Vehicle Routing Problem (VRP) has been deeply studied since the 1950's and has shown to be extremely hard to tackle. Studies on the variant of the VRP, the Pickup Delivery Problem (PDP), started to arise in the past two decades. Many variants of the PDP introducing different constraints have been tackled with different meta-heuristics. Despite this, the E-MTPDPTWT has not yet been tackled yet. The following paragraphs review the already studied problems related to the E-MTPDPTWT.

The relevant E-MTPDPTWT is an extension of many existing optimisation problems, namely the Pickup and Delivery Problem (Parragh et al., 2008), the Pickup and Delivery Problem with Time Windows (PDPTW) (Ropke and Pisinger, 2006; Bent and Hentenryck, 2006; Li and Lim, 2003), the Pickup and Delivery Problem with Transfers (PDPT) (Cortés et al., 2010) and the Electric Vehicle Routing problem with Time Windows (E-VRPTW) (Schneider et al., 2014). Different meta-heuristics have been developed in order to tackle these NP-Hard optimisation problems.

Nanry and Wesley Barnes (2000) were one the first to tackle the PDPTW with a meta heuristic. In their research, they introduced a reactive Tabu search approach combining three different neighbourhood moves. Li and Lim (2003) modified this tabu method and combined it with a simulated annealing algorithm. This approach restarts its search procedure from the current best solution after a fixed amount of non-improving search steps.

Bent and Hentenryck (2006) introduced a two-hybrid algorithm to solve the PDPTW. The first stage contains a simple simulated annealing approach to reduce the number of routes while the second stage introduces a Large Neighbourhood search (LNS) to minimise the travel distance. Another successful extension of this LNS approach is the Adaptive Large Neighbourhood search heuristic of Ropke and Pisinger (2006). The historic performance of the removal and insertion operators of the LNS determines the frequency of their application. An application of the Adaptive Large Neighbourhood Search on Pickup and Delivery Problem with Transfers has been reported by Masson et al. (2013). They propose new insertion heuristics in an adaptive large neighbourhood search that are capable of inserting transfers efficiently into the solution.

This variable neighbourhood search has also been applied to an Electric Pickup and Delivery

Problem (EVP). Schneider et al. (2014) introduced recharging stations in PDP's in where they combine the Variable Neighbourhood Search algorithm with a Tabu search heuristic.

Despite the successful applications of the biology-inspired genetic algorithm on the (Multi-)VRP problem (Vidal et al., 2012; Nalepa and Blocho, 2017; Cattaruzza et al., 2014), studies on genetic algorithms for PDP's have remained scarce. (Vidal et al., 2012) introduced a chromosome as a giant tour representation for the VRP, which is used in the crossover operator of this paper. An equivalent to this research's split procedure between crossover and education of the algorithm was introduced by Cattaruzza et al. (2014). In addition, Cattaruzza et al. (2014) introduced interesting local search operators.

Pankratz (2005) firstly tackled the PDP with a genetic algorithm and proposed an encoded group of request to represent a chromosome. However, little attention was paid on the intensifying education phase of the algorithm. Nalepa and Blocho (2017) later proposed an island-model parallel memetic algorithm on the pickup and delivery. Co-operation between processes ensures the adoption of the high quality genes in next generations which improves the convergence of the solution method. However, it fails to diversify its population when implemented in a genetic algorithm for our problem.

A novel representation for individuals (solutions) and chromosomes is introduced by Ting and Liao (2013). It uses a sequence order for the chromosomes which makes it possible to use a random cut in the crossover phase causing to diversify while maintaining good properties. Phan and Suzuki (2016) introduced an unassigned set of requests in the individual representation for a PDPTWD in order to efficiently deal with infeasibility after the crossover phase.

Blocho and Nalepa (2017) introduced a LCS-Based Selective Route Exchange Crossover for the Pickup and Delivery Problem with Time Windows. This operator recombines different chromosomes (trips) from different parents and selects the best combination to generate children.

To our knowledge, up to now, no studies on genetic algorithms have tried to include transfer points and recharging stations in their PDP's. The dependence constraint between vehicles caused by the transfer possibility makes especially the crossover phase of the algorithm difficult. Therefore, we introduce a novel representation of transfer dependencies in the crossover, education, repair and scheduling phases. For the local search operators in the education and repair phase other studies on Pickup and Delivery problems with Transfers and/or Recharging stations can be used to complement the unexplored part of the genetic algorithm in order to construct a good approach. Besides this, a Sub-Group Exchange Crossver (SGXX) is introduced in this paper and recombined with a repair procedure where Tarjan's Algorithm (Tarjan (1971)) becomes useful.

3 Problem Formulation

In this section, the E-MTPDPTWT is described mathematically. First, all parameters and variables of interest will be introduced. Next, the constraints a solution has to satisfy are presented.

3.1 Parameters and Variables

To avoid ambiguity, we define all parameters and variables of interest in this section. We start by introducing the parameters, after which we describe the variables we are going to use. Finally, we define our goal objective.

3.1.1 Parameters

Consider a given set of n transportation requests. Request r, $r = 1, 2, ..., n$, corresponds to the transport of goods of weight 1 from unique pickup location L_r to unique delivery location L_{r+n}. Hence, the pickup locations are labelled $1, ..., n$, the delivery locations are labelled $n + 1, ..., 2n$,

such that pickup location i and delivery location $i+n$ correspond to the same request, request r_i. Location L_0 denotes the depot location. Except for the depot, each location should be visited only once. Furthermore, every request r has a deadline d_r.

The distance from location L_i to location L_j is denoted by d_{ij}, where $i, j = 0, 1, 2, ..., 2n$. Of course, $d_{ij} = 0$ when $i = j$. Travelling d_{ij} distance takes a vehicle $t \cdot d_{ij}$ units of time.

There are K electric vehicles available, each having a loading capacity Q. The driver of a single vehicle cannot drive more than T units of time per day. All vehicles are equipped with a battery that has capacity D, where driving one unit requires e capacity. The depot serves as a transfer facility, i.e., when a vehicle returns to the depot, its battery is recharged. Recharging happens at a constant rate g per unit of time. After the vehicle is fully recharged, it is available for another tour.

The depot also serves as a transfer facility. Hence, goods from a pickup location can be dropped of at the depot after a tour, where a new tour can pick them up and bring them to delivery location. Transferring can only happen at the depot. A short overview of all variables and parameters of interest can be seen in Table 1.

Table 1: Parameter descriptions.

r	Request, $r = 1, 2, ..., n$
d_i	The deadline of request i
L_i	Location i, with $i = 1, ..., n, n+1, ..., 2n$
L_0	Depot location
d_{ij}	The distance from L_i to L_j, where $i, j = 0, 1, ..., 2n$
t	Travel time per unit of distance
K	Number of electric vehicles available
Q	Loading capacity of a vehicle
D	Battery capacity of a vehicle
e	Capacity usage per driven unit of distance for a vehicle
g	Capacity recharge rate per unit of time
T	Time duration of one day

3.1.2 Variables

Now that all parameters of the problem have been described, we can move on to the part of the problem that we can influence: the decision variables and their associated properties. A solution for our problem basically consists of m tours, each one scheduled at a certain moment to a certain vehicle. Hence, we can focus on the description of tours. In Table 3.1.2 a description of all variables of the tour can be seen.

We can define the amount of tours m before creating tours, however, a more natural way is to create good tours and see what m becomes afterwards. Let T_k denote tour k, with $k = 1, ..., m$. Of course, every tour starts and ends at the depot. Before returning to the depot, n_k locations are visited. Both pickup locations and delivery locations can be visited in one tour. The i-th location that is visited by T_k, $i = 0, ..., n_k, n_k + 1$, is denoted by l_i^k. Locations 0 and $n_k + 1$ denote the start and the return at the depot, respectively. Hence, a tour has the following route:

$$L_0 \longrightarrow l_1^k \longrightarrow l_2^k \longrightarrow ... \longrightarrow l_{n_k}^k \longrightarrow L_0$$

The time at which l_i^k is visited is denoted by t_i^k. Of course, $t_i^k < t_{i+1}^k$, since all locations are unique. Now, the departure time of the tour, S^k, obviously coincides with t_0^k. However, we decided to keep S_k as a separate variable, since this enhances the readability of a solution to our problem.

Besides the locations and arrival/departure times, we are also interested in the total freight carried by the vehicle at each point in time. Hence, we define f_i^k to be the total weight of all carried

goods after visiting location i. Since a vehicle drops off any goods it still carries when it arrives at the depot again, $f^k_{n_k} = 0$. Regarding the carried goods, we also want to know which goods are required to be present at the depot before starting the tour and which goods are dropped off at the depot after finishing the tour. These are indicated by $RG(T_k)$ and $DG(T_k)$, respectively. Finally, we assign tour T_k to some vehicle, namely V_k, with $V_k = 1, 2, ..., K$.

Table 2: Variable descriptions for tour k, denoted by T_k.

	Decision variables:
n_k	The number of locations that T_i visits
l^k	An $n_k \times 1$ vector that contains all locations T_k visits, in the order of visit
S_k	Starting time of T_k
V_k	Vehicle we assign T_k to
	Tour properties:
$\Delta(T_k)$	The total length of T_k
t^k_j	Time at which the j-th location of T_k is visited after starting T_k, $j = 1, ..., n_k, n_k + 1$
$RG(T_k)$	Goods required at the start of T_k
$DG(T_k)$	Goods that are dropped off at the depot after T_k
f^k_i	Total weight of carried goods after visiting location i, $i = 0, 1, ..., n_k$

3.2 Constraints

Of course, a solution has to satisfy multiple constraints. In the following, we will use the notation of Tables 1 and 2. Hence, for all T_k, with $k = 1, ..., m$, we have:

$$e \cdot \Delta(T_k) \leq D \tag{1}$$

$$f^k_i \leq Q \qquad \text{for all } i = 1, ..., n_k \tag{2}$$

$$t^k_i < t^k_j \qquad \text{if } l^k_i + n = l^k_j \tag{3}$$

$$t^k_j \leq d_{l^k_j} \qquad \text{for all } l^k_j \in \{n+1, ..., 2n\} \tag{4}$$

$$S_k \geq t^{\hat{k}}_j + g \cdot (D - e \cdot \Delta(T_{\hat{k}}) \qquad \text{if } V_k = V_{\hat{k}} \text{ for some other tour } T_{\hat{k}} \tag{5}$$

The above constraints are basically constraints for one vehicle/tour. Eq. (1) and (2) denote the driving range and loading capacity of a vehicle, respectively. Of course, a pickup has to be performed before a delivery, which is indicated by Eq. (3). Eq. (4) denotes the deadline constraint, whilst Eq. (5) indicates that a vehicle can only travel one tour at a time. The following constraints are cross-tour constraints:

$$\frac{\sum_{i=1}^{m} n_i}{2} = n \tag{6}$$

$$\sum_{i=1}^{m} \mathbf{1}_{V_i = j} \cdot (t^i_{n_k} - S_i) \leq T \qquad \text{for all vehicles } j = 1, ..., K \tag{7}$$

$$S_k \geq t^{\hat{k}}_{n_{\hat{k}}+1} \qquad \text{if } \exists x \text{ such that } x \in RG(T_k) \text{ and } x \in DG(T_{\hat{k}}) \tag{8}$$

Here, Eq. (6) denotes that all requests should be processed, Eq. (7) indicates that no vehicle should drive longer than the duration of one day, T, and Eq. (8) states that a transfer is only possible when goods are delivered at the depot before they are used by some other tour.

3.3 Goal

Using the variables and constraints defined above, we can define our objective. The objective of this study is to minimise the total driven distance by all vehicles, whilst satisfying all constraints.

Alternatively, we can write:

$$\min \sum_{i=1}^{m} \Delta(T_i) \qquad \text{subject to Eq. (1) - (8)} \tag{9}$$

4 Algorithm Outline

This section describes the proposed Hybrid Genetic algorithm for the E-MTPDPTWT. The algorithm combines intensifying capabilities of different local search algorithms and the diversifying capabilities of the genetic algorithm. In Algorithm 1, the general outline of the algorithm can be seen.

Algorithm 1 Hybrid Genetic Algorithm Outline

```
 1: Initialise population P
 2: while Stopping criteria is not met do
 3:     Select parent chromosomes S_{p_1} and S_{p_2} from P
 4:     Generate children S_{C_1} and S_{C_2} using Crossover
 5:     if S_C is infeasible then
 6:         Repair S_C
 7:     end if
 8:     Educate S_C
 9:     Insert S_C into the population
10:     if Dimension of the population exceeds a given size then
11:         Select survivors
12:     end if
13: end while
```

The algorithm initialises both feasible and infeasible solutions with different constructive heuristics that are capable of generating a diversified initial population (Line 1). The feasible population will be known as P_1, the infeasible as P_2. Then, a binary tournament is conducted to select two parents, S_{p_1} and S_{p_2}, from the two populations (Line 3). Then, from these parents, two children S_{C_1} and S_{C_2} are generated using a crossover operator (Line 4). Based on two cuts that decompose the parent into two limbs and a body, the crossover operator inserts the body of S_{p_1} inside the limbs of S_{p_2} and vice versa. This way, two children are generated which first are repaired with respect to missing and duplicate locations, driving range and capacity constraints.

By now, children can only be infeasible with respect to time constraints. The tours of the children are now scheduled and, if infeasibility with respect to time constraints arises, also repaired (Line 6). After the repair, both children are educated using local search operators (Line 8) in order to exploit the generated existing solutions S_{C_1} and S_{C_2}. The children are inserted in either the feasible or infeasible population, based on their feasibility (Line 9). When the dimension of the populations exceeds the pre-specified size, the survivors are selected based on diversification and objective value (Lines 10-11). This iterative process (Lines 3-12) continues until certain stopping criteria are met and the algorithm is satisfied with the obtained solution (Line 2).

The main components of the algorithm are described in the following subsections. In Section 4.1 the representation of the population, individuals and chromosomes is described. Using these representations, Sections 4.2-4.3 describe the initialisation and the population survival. Then, in Sections 4.4-4.5, the selection and child generation with the crossover procedure is presented. Then, the scheduling, repair and education phases are described in Section 4.6-4.8. Finally, some light will be shed on the feasibility of solutions in Section 4.9.

4.1 Data Representation

As already mentioned, this section will elaborate on the representation of various aspects of our solutions. First, the population as a whole will be described. Then, the representation of an individual solution will be explained. Finally, the chromosomes, which represent parts of an individual, will be described.

4.1.1 Population Representation

The population contains individuals representing solutions of two classes: infeasible and feasible solutions. Including an infeasible population has proven in vehicle routing problems to contribute to the performance of the genetic algorithm, see for example Vidal et al. (2012). Hence, we have chosen to both include feasible and infeasible solutions, so that we have a feasible population P_1 and an infeasible population P_2. This should allow the algorithm to search on the border of infeasibility for good solutions.

The two populations of feasible and infeasible individuals, will both be sorted on goal value v. The size of the population of feasible individuals will be N_1, similarly, N_2 will denote the population size of infeasible individuals. Hence, we will have a populations:

$$P_1: \quad v_{(1)}^{P_1} \leq v_{(2)}^{P_1} \leq ... \leq v_{(N_1)}^{P_1}$$
$$P_2: \quad v_{(1)}^{P_2} \leq v_{(2)}^{P_2} \leq ... \leq v_{(N_2)}^{P_2}$$

Where $v_{(j)}^{P_i}$ denotes the goal value of the j-th best solution of population P_i. If an individual is inserted in population i and the population size N_i is exceeded, the j observations with the worst goal value are removed, that is, $v_{(N+1)}$ until $v_{(N+j)}$, for $j \geq 1$.

4.1.2 Individual representation

An Individual in the populations represents a feasible or infeasible solution. The individual contains all relevant information of a solution. As is done in existing literature (Vidal et al., 2012; Pankratz, 2005; Nalepa and Blocho, 2017) and described in Section 3.1.2, a solution is represented by m different tours, which will be named chromosomes. However, due to the time windows and possible transfers, the chromosomes also need to be scheduled at a certain moment in time and to a certain vehicle. Also, for the genetic algorithm the relations between these chromosomes and other properties of the solution need to be stored. Hence, for individual k, the following data is stored:

- An ordered sequence of n_k chromosomes. This representation is necessary for making a cut for the crossover which will be described in Section 4.5.
- The dependence forest, an $n_k \times n_k$ matrix Σ^k. This models the transfer dependence of chromosomes: $\Sigma_{i,j}^k = 1$ if chromosome i requires a drop-off of chromosome j, 0 else. The dependence forest, basically a directed graph, is necessary for the child generation in Section 5 and for the scheduling part described in Section 4.6.
- The schedule of all chromosomes. This contains the following:
  - A vector J^k of dimension $K \times 1$, where J_i^k denotes the number of chromosomes that are currently assigned to vehicle i.
  - Three $K \times n_k$ matrices:
    1. C^k: element $C_{i,j}^k$ denotes the j-th chromosome that is assigned to vehicle i;
    2. A^k: element $A_{i,j}^k$ denotes the time at which vehicle i arrives at the depot after completing the j-th chromosome;
    3. R^k: element $R_{i,j}^k$ denotes the time at which vehicle i is fully recharged after completing the j-th chromosome, i.e., when it is ready for another chromosome.

  This schedule is necessary for feasibility checks in the repair phase, as it allows us to check whether time constraints and transfers are handled properly.
- A table with visited locations. This shows which location is visited by which chromosome. The table is useful for the education and repairing phase in Sections 4.7-4.8.

- The resulting goal value v^k
  This is necessary for ordering the individual in the population.
- The individual's infeasibility. This is a number that indicates by how much the time window constraints are violated. If a solution is feasible, this equals zero obviously.

4.1.3 Chromosome representation

A Chromosome will basically hold all information of a single tour a vehicle makes from the depot, to some location(s) and back to the depot again. Building upon the mathematical formulation presented in Section 3.1.2, a chromosome k contains the following:

- A sequence S_k that contains all visited locations in the order they are visited. It also contains the time at which it arrives at each location.
- Attached to each location in the sequence is the (cumulative) arrival time at that location.
- The total distance of the tour Δ_i. This can also be used to calculate the recharging time after the tour.
- A list of locations that have to be visited and have their goods dropped off at the depot before this tour can start.
- List of locations that are visited and have their goods dropped off at the depot at the end of this tour.

The last two items indicate the transfer dependence of this specific chromosomes and are used to generate the dependence forest of an individual. In Figure 1, a visualisation of the complete data representation is presented.

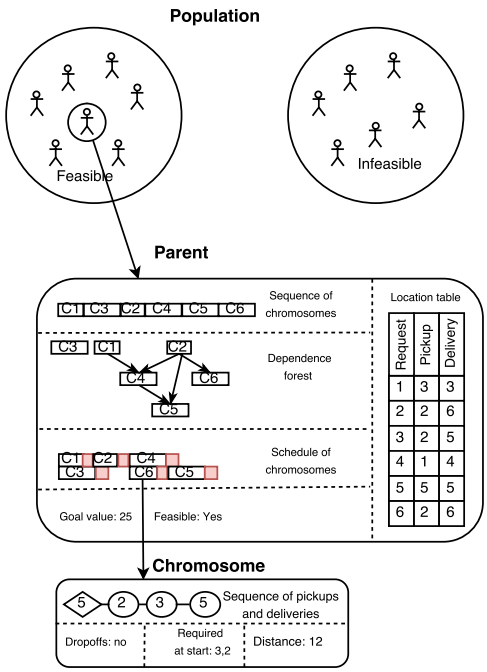

Figure 1: Visual data representation of the Hybrid Genetic algorithm.

4.2 Population Initialisation

To ensure a sufficiently diverse starting population, heuristics should be used to generate the initial population. For diversity purposes, we have chosen to include two initialisation heuristics, compared to for example (Cattaruzza et al., 2014), which used only one heuristic. Our first heuristic, H_1, focuses on the absence of transfers, whereas the second, H_2, forces transfers to be present in a solution. Algorithm 2 is based on the sequence heuristic of Hosny and Mumford (2012) modified on the driving range constraint. Algorithm 3 is newly introduced since it introduces the enforcement of transfers in the heuristic.

Algorithm 2 Initialisation Procedure H_1

Require: Set of all requests U, auxiliary empty set V
while Unassigned requests are available ($U \neq \emptyset$) **do**
  Create new empty chromosome C_k
  while $\Delta_k < Q$ and $U \neq \emptyset$ **do**
    Remove random request r from U or V
    if r can be added to C_k, given all constraints **then**
      Add r to C_k
    else
      Add r to V
    end if
    End the inner while loop with probability q
  end while
end while
while $V \neq \emptyset$ **do**
  Create new empty chromosome C_k
  for all Pickups p and deliveries d in V **do**
    Add as many p and d to C_k as possible
  end for
end while
Create dependence forest
Schedule the chromosomes and create the individual's sequence

Algorithm 3 Initialisation Procedure H_2

Require: Set of all requests U, auxiliary empty set V
while Unassigned requests are available ($U \neq \emptyset$) **do**
  Create new empty chromosome C_k
  Remove random location l_{start} from U or V, add it to C
  while $\Delta_k < Q$ and $U \neq \emptyset$ **do**
    if l_{start} is a pickup location **then**
      With probability p: select unvisited pickup location l
      With probability $1 - p$: select unvisited delivery location l
    else
      With probability p: select unvisited delivery location l
      With probability $1 - p$: select unvisited pickup location l
    end if
    End the inner while loop with probability q
    if l can be added to C_k, given all constraints **then**
      Add l to C_k
    else
      Add l to V
    end if
  end while
end while
while $V \neq \emptyset$ **do**
  Create new empty chromosome C_k
  for all Pickups p and deliveries d in V **do**
    Add as many p and d to C_k as possible
  end for
end while
Create dependence forest
Schedule the chromosomes and create the individual's sequence

4.2.1 Cheapest insertion heuristic

In Algorithm 2 and 3 and in the re-insertion described in Section 5, locations are inserted into a chromosome that already has a tour. The tour of a chromosome is determined using the nearest possible neighbour algorithm. An outline of this algorithm is presented in Algorithm 4.

Algorithm 4 Tour Generation

Require: Set of all tour locations L, distance matrix d
1: Set current location C equal to depot location L_0
2: Let $L'_C \subseteq L$ be the set of locations that can be visited from C
3: **while** $L'_C \neq \emptyset$ **do**
4: Pick $l \in L'$ such that $d_{C,l} \leq d_{C,k}$ for $k \in L' \setminus \{l\}$
5: Append l to the current tour
6: Set C equal to l
7: Update L'_C
8: **end while**

4.3 Survival

A new feasible individual will be inserted into the feasible population if its goal value is lower than highest goal value of the population. An infeasible solution will only be added to the population if both its infeasibility and goal value are lower than the maximum infeasibility and goal value of the population. A threat for the convergence of genetic algorithms, is the presence of clones (Prins, 2004). Of course, the larger our E-MTPDPTWT becomes, the less likely it is that two individuals are identical, but the possibility of clones cannot be ruled out altogether. Contrary to Vidal et al. (2012), who removed solutions with goal values close to each other, we acknowledge the fact that different solutions can yield (approximately) the same goal value. This could happen when some solutions are close to a local optimum. Therefore, when trying to insert a newly created individual into the population and it has the same goal value *and* chromosome sequence as another individual, the insertion is aborted. This approach should conserve diversity in our population.

4.4 Selection

At the heart of a genetic algorithm, there is the selection of individuals that will produce offspring together. Several selection procedures are available, of which an excellent overview can be seen in Blickle and Thiele (1996). Due to the balance of intensity and diversity, we opt for the binary tournament selection. That is, we choose two random individuals from the complete population and select the fittest one. This will be the first parent. The second parent will be chosen in the same manner: as the fittest of two randomly selected individuals from the total population. Note that the random selection is *with* replacement. These two parents will mate together, which will be described in Section 4.5.

4.5 Child generation

After the parent selection, the children can be generated. In Algorithm 5, the child generation process can be seen. In Sections 4.5.2-4.5.4, some important concepts and operators will be described in more detail.

Algorithm 5 Child generation

Require: Two parents σ_1 and σ_2, maximum amount of missing locations m

1: Create three subsets X_a^i, X_b^i, X_c^i from the chromosome sequences of parent i, $i = 1, 2$
2: Perform Crossover(X_a^1, X_b^1, X_c^1, X_a^2, X_b^2, X_c^2) and create children $j = 1, 2$ containing:
   Set of Chromosomes C_j
   Set of unvisited locations U_j
   Set of duplicate locations D_j
3: **if** $U_j \leq m$ **then**
4: **for** $j = 1$ to 2 **do**
5: Generate dependence forest from C_j and check whether it is correct (e.g. no loops)
6: **while** Dependence forest is not correct **do**
7: Fix with removal of duplicates and update D_j
8: Remove loop causing locations from C_j and add to U_j
9: **end while**
10: **while** $D_j \neq \emptyset$ **do**
11: Clean up duplicates $d \in D$ using removal operator:
  With probability p: RandomRemoval(d)
  With probability $1 - p$ BestRemoval(d)
12: Update dependence forest
13: **end while**
14: **while** $U \neq \emptyset$ **do**
15: Select $u \in U$
16: Check in the dependence forest in which subset of chromosomes C_j^{CD} the insertion is allowed.
17: Re-insert missing location u in C_j^{CD}:
  With probability p: RandomInsert(u)
  With probability $1 - p$: BestInsert(u)
18: **if** u cannot be inserted in any C_j **then**
19: Create new chromosome C_j and add u to C_j
20: **end if**
21: Update dependence forest
22: **end while**
23: **end for**
24: Schedule the chromosomes and create the child's sequence
25: **else**
26: Reject the child
27: **end if**

First, two random cuts creating three subsets are made. These subsets, which can be thought of as the limbs and body of a parent, are used for crossover in Line 2 and can be empty. The proposed crossover (SGXX) operator is explained in Section 4.5.1. In Line 3, the algorithm checks whether the child does not miss too many locations. This is so that the crossover does not act as a new initialisation procedure, but rather uses the good properties of both parents. Then, in Line 5, the dependence forest of the generated child is created. Note that at this stage, loops in the dependence can be present, which are fixed in Lines 6-9. In Section 4.5.2, a definition of a loop and its undesired property is provided and the algorithm used to check whether one is present in the forest is explained.

Then, in Lines 11-22, using random and best removal, locations that are visited twice are removed and missing locations are re-inserted. Inserting locations is performed using random and best insertion, in order to both explore and exploit whilst generating children. These insertion and removal heuristics use the dependence forest in order to perform feasible insertions and removals. These operators are described in more detail in Section 4.5.3-4.5.4.

Finally, in Line 12 and 21 the dependence forest is updated after the removal and insertion respectively. This means that when set U and set Q are both empty the dependence forest contains no unassigned dependencies anymore. Note that in this phase the children might be infeasible with respect to time windows since they are not scheduled yet, however all other constraints are met. The child's schedule, and as a result also its infeasibility, is calculated in Line 24.

4.5.1 Crossover

In the existing literature, various crossover operators are used. Cattaruzza et al. (2014) used an Order-based Crossover (OX), whereas Pankratz (2005) used a Group-based Crossover (GX). Another interesting type, the Subtour Exchange Crossover (SXX) is presented in Katayama et al. (2000) for a TSP problem. For the E-MTPDPTWT problem here, a combination of the GX and SXX is proposed, resulting in the Sub-Group Exchange Crossover (SGXX). Similarly as in Pankratz

(2005), the cut in Line 1 creates three subsets: X_a^i, X_b^i, X_c^i for each parent $i = 1, 2$. Then, with the SXX as in Katayama et al. (2000) a new set of chromosomes C_1 is created from the chromosomes in subsets X_a^1, X_b^2, X_c^1 for child 1, similarly, C_2 is created from X_a^2, X_b^1 and X_c^2. In Figure 2, a visual representation of the SGXX can be seen. This way, the diversification is done by combining the different set of chromosomes of two parents. Though, some good properties (e.g. first pickups and locations with early deadlines, useful transfers) of the parents are kept by selecting the subset containing the beginning group of the sequence of the chromosomes of one parent and the middle group of the sequence of the other parent and the end part of the sequence of the first parent (and of course vice versa for the other child). Note however that the chromosomes are not scheduled yet or placed in a sequence; this will be done in Section 4.6.

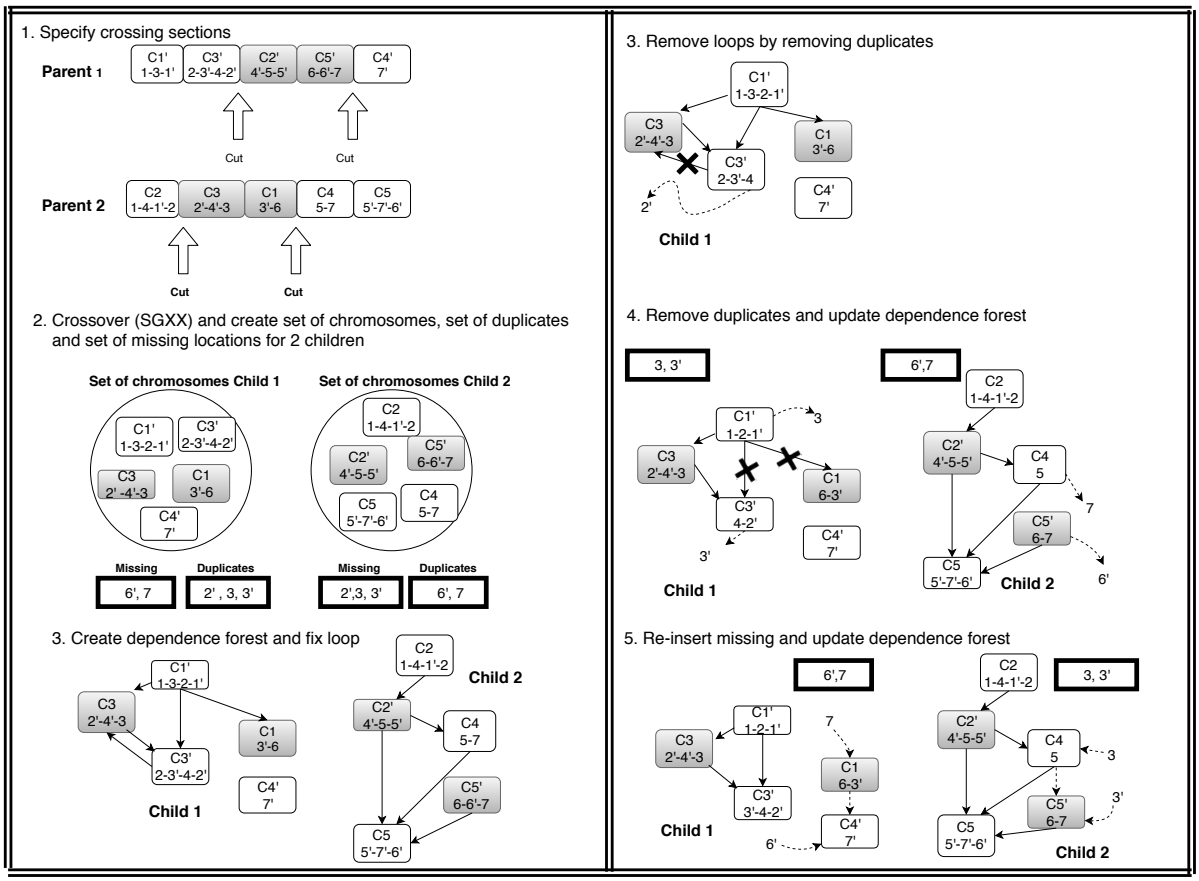

Figure 2: Visualisation of the generation of children using the Hybrid Genetic Algorithm. Steps 1 and 2 corresponds to the SGXX. Steps 3, 4 and 5 remove loops, remove duplicate locations and insert missing locations, using the generated dependence forest as a guideline for possible removals and insertions.

4.5.2 Loops in dependence forest

The dependence forest represents the dependence between the chromosomes in a particular solution. Viewing the dependence forest as a graph, where $\Sigma_{p,q} = 1$ denotes a directed edge from p to q, the following definition is used for this dependence:

Definition 1 *If there exists a path from i to j, chromosome i is lower dependent on chromosome j and chromosome j is said to be upper dependent on chromosome i.*

The dependence forest of the solution after child generation is added in order to schedule efficiently in Section 4.6. Therefore, the most important part of the repair phase in the child generation is

removing the so-called loops from the child. Loops in the dependence forest makes it impossible to schedule the chromosomes and therefore have to be removed. We define a loop as follows:

Definition 2 *A component of the dependence forest is defined as a loop if every vertex (chromosome) in the component is reachable from every other vertex (chromosome) in that component. That is, two or more vertices (chromosomes) form a strongly connected component.*

As an illustration, in Step 3 of Figure 2, a loop is present between $C3$ and $C3'$. In this proposed genetic algorithm these strongly connected components in the solution are identified at Line 6 of Algorithm 5, by making use of Tarjan's algorithm as introduced in Tarjan (1971). This depth first algorithm gives strongly connected components as output which in this algorithm will be removed by destroying the dependency of a selected chromosome k on another selected chromosome r in the component.

The component's chromosome that has the least lower dependence is selected as k and a chromosome that k is lower dependent on is selected as r. This dependency is broken by removing duplicate (or non duplicate) deliveries from k having a corresponding pickup in r and removing duplicate (or non duplicate) pickups location in r having corresponding delivery location in k.

4.5.3 Removal operators

For removing duplicates in Line 11 of Algorithm 5, two operators are used in order to both exploit and explore in the solution. The *random removal* randomly selects one of the two locations and removes it from that chromosome. *The best removal* selects the duplicate reducing the total distance the most after removal. The reason for including the random removal in the procedure, is that best removal is a greedy procedure and might not yield the best final result. Hence, a combination of diversifying and intensifying is suitable here.

4.5.4 Insertion operators

For inserting locations in Line 22 of Algorithm 5, again two operators are used for the same reason as in Section 4.5.3. The *random insertion* inserts the missing location in a random chromosome using the nearest possible neighbour heuristic described in Section 4.2.1. When the insertion violates a constraint (precedence, capacity, driving range) in the randomly selected chromosome, another random chromosome is selected. The *best insertion* inserts the missing location in all chromosomes and selects the one that is feasible and reduces the distance the most.

4.6 Scheduling

The scheduling phase is not very common in the literature on genetic algorithms for VRP's and PDP's, however, because of the transfers and time window constraints, it is necessary. It allows us to see whether the chromosomes of an individual can be scheduled such that it makes it deliveries of all requests within the deadline, satisfying transfer dependence. The representation of the schedule is described in Section 4.1.2. The dependence forest will become relevant in this procedure since it can be efficiently used to make sure all transfers are correct in terms of precedence. Therefore, another component in the dependence forest need to be defined:

Definition 3 *A chromosome is marked as the root chromosome if it is not lower dependent on any chromosome.*

This definition becomes relevant in selecting the to-be scheduled chromosomes. The scheduling is done based on the 'tightness' of a chromosome, which is a proxy for the priority of the chromosome. We come to the following definition:

Definition 4 *For a chromosome i that is lower dependent on chromosome j, let τ_j^i denote the total driving time of all chromosomes between i and j. Then, the tightness ω_i of chromosome i is*

calculated by:

$$\omega = \min_{j \in l^i}(d_j - t^i_j) - \max_{j \in C_L}(\tau^i_j + t^j_{n_j})$$

Here, C_L is the set of chromosomes that chromosome i is lower dependent on.

Using the above definitions, we can present the scheduling algorithm. The scheduling process is shown in Algorithm 6.

Algorithm 6 Scheduling

Require: Empty schedule with K vehicles and maximum time T, dependence forest Σ from child k
1: **while** Chromosomes are not yet scheduled **do**
2: **Probability p**: Select the chromosome i with the lowest tightness ω_i
3: **if** i is not root chromosome **then**
4: Select root chromosome j with the highest $\tau^i_j + t^j_{n_j}$
5: **end if**
6: **Probability** $1 - p$: Select random root chromosome
7: Schedule selected chromosome and place behind current sequence
8: Remove selected chromosome from the dependence forest
9: **end while**

Iteratively, root chromosomes are scheduled and removed from the dependence forest such that new chromosomes become root chromosomes. With probability p, the selection of a root chromosome is based on the tightness of the chromosomes. With probability $1 - p$, a random root chromosome is selected. At the end, all chromosomes are removed from the dependence forest and hence they are all scheduled.

4.7 Schedule Repairing

After all chromosomes are scheduled, the individual might not make all deliveries before their deadlines. The repair procedure for fixing this can be seen in Algorithm 7.

Algorithm 7 Repair outline

Require: Individual k with schedule S_k and Set Q of unmet deadlines.
1: Create a new schedule S_{New}, using rescheduling with $p = 1$
2: Select the schedule that yields the least infeasbility from S_k and S_{New}
3: **for** q in Q **do**
4: Check whether a repair operator removes the violation q
5: **if** There is one or more that satisfies the constraint **then**
6: Perform the repair operator that yields the best goal value
7: **else**
8: Mark solution as infeasible
9: **end if**
10: Update the solution and remove q from Q
11: **end for**

This procedure is only performed for infeasible children and starts by trying to reschedule with $p = 1$. This way, the schedule procedure always schedules chromosomes containing the chromosome with the earliest deadline first, in order to increase the probability of meeting all deadlines. If this is not successful, the solution can be repaired in other ways. These alternative methods of repairing a solution to reach feasibility (Line 4-6) are described in Section 4.7.1. If no feasibility operator can remove the violation of the specific constraint the child will be marked as infeasible and the algorithm will check if it can remove other violations, granted they exist. After the repair operators, the schedule will be updated and the next unmet deadline will be removed from the set of to be repaired deadlines (Line 4).

4.7.1 Feasibility repair operators

For Line 4 of Algorithm 7, the following five types of feasibility repair operators are proposed.

- R_1: If a vehicle is available at that time span, split the chromosome containing the unmet deadline location and distribute the locations optimally over two instead of one vehicle, given deadlines, precedence and capacity constraints.

- R_2: If the late request is in a chromosome i that is lower dependent on other chromosome(s) j, try if the deliveries in i can be fit into j, or if the pickups in j can be fit into i.
- R_3: Swap the late request with a request that is processed earlier but has a later deadline, given all constraints.
- R_4: If no vehicle is available at that time span, remove the unmet deadline pickup and delivery, create a separate chromosome for them and reschedule.
- R_5: using the education operators that will be presented in Subsection 4.8, try whether or not the late request can be inserted in another chromosome, both in upper dependent chromosomes or non-lower dependent chromosomes.

Initially, all operators are possibly successful. One operator is selected with probability $p = \frac{1}{5}$, if it does not succeed, one of the remaining possibly successful operators is chosen probability $p = \frac{1}{4}$, and so on. These five repair operators were designed to be not too intrusive for the solution and they should give a good balance between preserving transfers and removing transfers.

4.8 Education

If a child has been repaired, that possibly has made the solution worse. If no repair was needed, the child might still have some unfulfilled potential. Either way, the child will be educated in order to allow it to find its local optimum. In Algorithm 8, the education outline can be seen. The education operator searches for improvements (Line 2) in the neighbourhood of the child solution and therefore does not allow for moves that lower the quality (Line 8).

Algorithm 8 Education

Require: Chromosome sequence and dependence forest of child K :
1: **for** M desired local search moves **do**
2: Select a random new pair (u, v) of two locations
3: **for** Every education operation **do**
4: Check feasibility
5: Calculate goal value
6: **end for**
7: Accept best feasible education operator
8: Check improvement of solution and accept/reject education
9: **end for**

The education operators that search for local improvements are performed M times and selected randomly (Line 3). Some of the education operators constructed in Prins (2004) and also used by Cattaruzza et al. (2014) will be used in this algorithm. Given a pair of locations u and v, we can educate the Individual using one of the following operations:

- M_1: Swap u and v
- M_2: Place u after v
- M_3: Place v after u
- M_4: Place u before v
- M_5: Place v before u

Education is feasible when it satisfies the transfer dependencies, the maximum load, the maximum driving range and deadlines (Line 4). Then the best operator is selected and performed when it improves the goal value (Line 7-8). This education procedure is repeated for a fixed number of times M for each child.

4.9 Feasibility

As a result of our algorithm design, generated solutions will always satisfy weight, transfer, precedence and driving range constraints. The only infeasibility will come from requests that are deliv-

ered beyond their deadline, or vehicles that have to drive longer than the duration of a day. The main advantages of this approach are tractability, that is, only the time constraints can be relaxed, and the absence of needing to compare infeasibility in one direction with infeasibility in another. Especially the second advantage is beneficial, as the requirement of determining realistic penalty parameters disappears. Whilst our approach limits the flexibility of searching for feasible solutions on the border of infeasibility, we believe that the benefits outweigh its limitations. Furthermore, since the weight, transfer, precedence and driving range constraints are technical constraints, compared to the deadlines and maximum driving time that are more 'human-made', it makes more sense to prioritise satisfying the laws of nature over satisfying constraints imposed by humans.

4.10 Parameter settings

In this section, we will indicate what parameter values were used and explain the reasoning behind it. In Table 3, the parameter values can be seen. Here, we will elaborate on the parameter settings for respectively the initialisation procedures, the crossover and the genetic algorithm.

Table 3: Parameter values used in our software implementation.

Parameter	Value
Initialisation Procedures:	
Probability of adding another location	$1 - \frac{1}{Q}$
Probability of adding an extra pickup location (Procedure H_2 only)	$1 - \frac{1}{Q}$
Crossover:	
Maximum missing pickups	$\sqrt{n}$
Maximum missing deliveries	$\sqrt{n}$
Genetic Algorithm:	
Feasible population size	125
Infeasible population size	125
Probability of using Initialisation Procedure H_i	$\frac{1}{2}$ for $i = 1, 2$
Number of iterations	250
Probability of choosing a feasible parent	0.8
Number of education local search moves	n

For the initialisation procedures as described in Section 4.2, we want the length of a particular trip to be dependent on the freight capacity Q of a vehicle. That is, if a vehicle can carry more goods, we want longer trips than if a vehicle can carry less. Hence, an intuitive value for the probability of adding an extra (pickup) location is $1 - \frac{1}{Q}$. Using this parameter on various instances showed good results.

The crossover has only two adjustable parameters, namely the maximum allowed missing pickups and the maximum allowed missing deliveries. Of course, for larger instances these should be higher, but not too high, since computational complexity becomes an issue. Therefore, we want sublinear growth of these values, for example $\sqrt{n}$ or $\log(n)$. However, for the instances at hand, with up to 500 requests, $\log(n)$ accepts a maximum of two missing locations, which is too strict. Therefore, we opted for $\sqrt{n}$ for both parameters.

For the genetic algorithm, we have to decide on the size of the two populations, the probability of choosing a certain initialisation procedure, the number of iterations and the probability of choosing a feasible parent. A good balance between computing time and solution quality was found to be at population sizes of 125 and using 250 iterations, using trial and error. Furthermore, since there was no preference for one of the initialisation procedures, we gave them equal probability of being used. Since decent feasible solutions are preferred over good infeasible ones, we want to use more feasible solutions as parents. Therefore, we gave the algorithm an 80% chance of picking a feasible parent and a 20% chance of picking an infeasible one.

Finally, the number of education local search moves should be related to the instance size, as

larger instances possibly have more directions in which the solution can be improved. A linear relationship between the number of requests and the number of education iterations seemed to perform quite well, which is why this parameter was set equal to n.

5 Results

We have tested our method on a standard benchmark set of 594 instances, the results of which can be found in a separate file. The algorithm was coded in `Python 3.6` and run on the University of Groningen's Peregrine computing cluster. We would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

When running the algorithm, we found that on larger instances, the runtime of our algorithm explodes. Whether this is due to our algorithm design or the specific implementation of it, remains unclear. However, this means that not for all instances, feasible solutions have been found. In Section 6, we will elaborate on possible reasons for the fact that the algorithm was unable to find feasible solutions

5.1 Solution quality

Finding feasible solutions is of course to be desired, but it is also interesting to see what the quality of the feasible solution is. From this point on, by 'solution', a feasible solution is meant. It is interesting to compare the best solution of our algorithm with the best solution of the initial population, granted that it exists. Furthermore, we are interested in the development of the solution quality over the iterations of the Hybrid Genetic Algorithm.

To measure the solution quality, we selected four similar instances that had generated feasible solutions. These four instances can be seen in Table 4. As can be seen, we have two instances with ten requests: one with and one without deadlines. The remaining two instances both have 25 requests, and again one has deadlines and one does not. Note that the travel time parameter differs between the instances with 10 requests and the instances with 25 requests, but this did not seem to impact the behaviour of the solution quality much. The parameters as described in Section 4.10 were used.

Table 4: Used instance and their settings.

Parameter	Instance 13	Instance 323	Instance 68	Instance 378
Deadlines	No	Yes	No	Yes
n	10	10	25	25
T	6	6	6	6
K	3	3	3	3
Q	5	5	5	5
D	300	300	300	300
e	1	1	1	1
g	400	400	400	400
t	0.005	0.005	0.002	0.002
Average runtime (mm:ss)	01:33	03:17	19:51	47:47
Std. deviation runtime (seconds)	3.4	9.1	64.1	157.0

Each instances was run 25 times with different seed values. The calculation times and their standard deviations can be seen at the bottom of Table 4. Each iteration, the best feasible goal value was stored, granted that it existed. Then, for each iteration where all instances had generated at least one feasible solution, the average of all 25 best feasible goal values was calculated. Let us write

μ_i for the average best goal value of iteration i, where $i = 0$ denotes the end of the population initialisation. Then, we can define the relative average solution quality λ_i of μ_i as follows:

$$\lambda_i = \frac{\mu_i}{\max_i(\mu_i)} \times 100\%$$

Hence, the worst iteration has a $\lambda_{worst} = 100\%$. Since the best average solution cannot decrease as iterations increase, we have $\lambda_i \geq \lambda_j$ for $i < j$. In Figure 3, the relative average solution quality (RASQ) can be seen for all four instances over a horizon of 250 iterations.

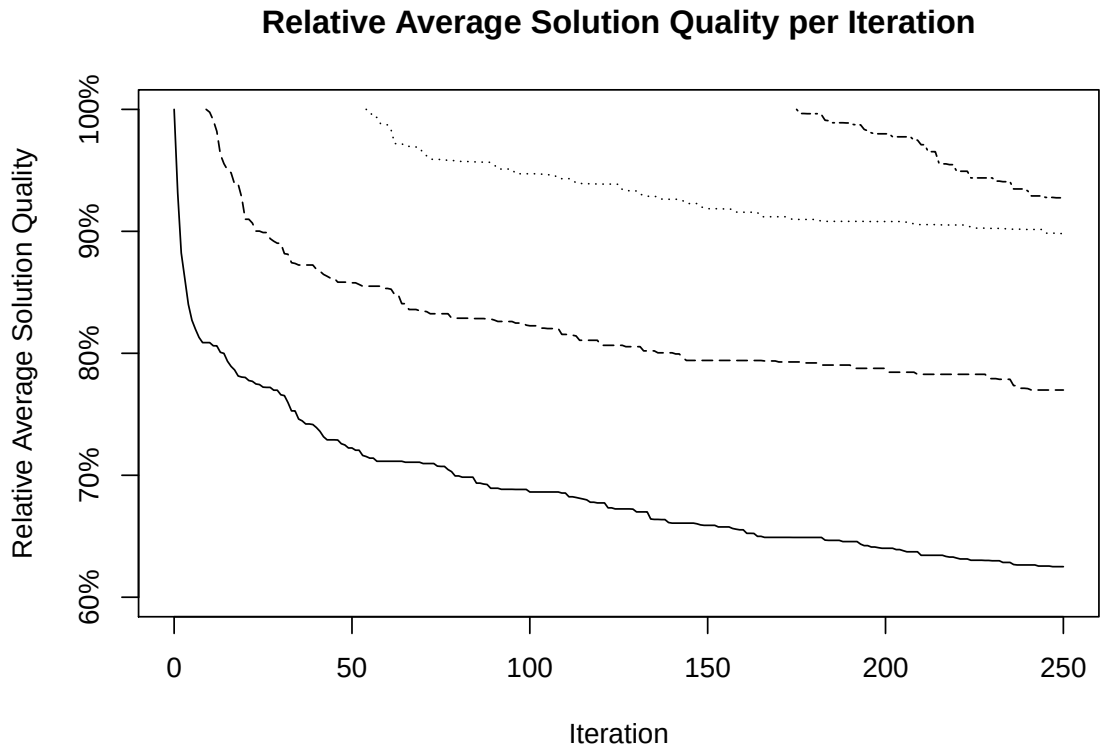

Figure 3: The average relative solution quality of various instances per iteration. The quality of a solution is calculated as its goal value divided by the worst goal value. Nothing is plotted when not all observations have at least one feasible solution. The solid line corresponds to instance 13, the dotted line to instance 323, the dashed line to instance 68 and the dashdotted line to instance 378.

Instance 13 is not a very complex instance. Therefore, it does not come as a surprise that the initial population contains feasible solutions in all 25 runs. It takes less than 10 iterations to improve the best initial solution by approximately 20%, which indicates that our initial population does not contain very good solutions. The RASQ keeps decreasing for the rest of the 250 iterations and finishes with a solution that has 62.5% of the goal value of the solution of iteration 0.

Adding deadlines to instance 13 creates completely different behaviour. Only in 8% of the algorithm runs, an initial feasible population was created. In the worst case, it takes 54 iterations of the Hybrid Genetic algorithm to find a solution. This shows that our algorithm is not only able to improve solutions, but also to find feasible solutions from a starting point of infeasible solutions only. After 250 iterations, the average solution has been improved by almost 10%.

In the case of an instance with more requests, the complexity also increases. The initialising procedures struggle with creating an initial feasible population for instance 68 in 76% of the runs. However, after 9 iterations, all algorithm runs had found a solution. The solution quality improves

quite fast during the first 75 iterations, but does not improve much afterwards. After 250 iterations, the RASQ is 77%.

It should not come as a surprise that adding deadlines to the large instance adds even more complexity. Therefore, the worst case for instance 378 is that it takes 175 iterations to find a feasible solution. In no algorithm run, there was a feasible initial population present. In the last 75 iterations, the Hybrid Genetic Algorithm manages to lower the RASQ to 92%, however, it looks like adding extra iterations might have lowered the RASQ even more.

When looking at the calculation times in Table 4, we see that for instances 13, the calculation time is a little over one minute and thirty seconds. Adding deadlines, creating instance 323, doubles the calculation time. Instance 68, that has an extra 15 locations compared to instance 13, has an average calculation time of almost twenty minutes - more than ten times that of instance 13. Adding deadlines to instance 68 again doubles the calculation time. These results were generated on a single core of an Intel Xeon E5 2680v3 @ 2.5 GHz with 500MB memory.

6 Discussion

This section discusses various aspects of the proposed hybrid genetic algorithm and the used implementation. First of all, one should note that the algorithm that was executed on the instances, was not completely equal to the algorithm presented in the above. Due to the time constraints of this course, various aspects of it have been simplified or even left out. The most important changed aspects are:

- In the SGXX, instead of two cuts separating a parent in a body and two limbs, there is only one random cut used, separating a parent in a head and a tail. The head of the first parent is connected to the tail of the second parent.
- The scheduling was only done using the tightness of chromosomes, no random scheduling took place.
- The random duplicate removal and missing location insertion operators are not used.
- Not all repair operators were used, only R_2, R_3 and R_5.
- Not all education operators were used for every pair (u, v), instead, an operator was randomly chosen and if it yielded a better child, it was used. If it did not yield a better child, another operator was chosen.

Compared to our current, simplified implementation, an extended implementation could possibly add more diversity to our population, create more feasible solutions and intensify solutions better. The exact importance and influence of these simplified aspects remains unclear at this moment in time, however.

6.1 Infeasible results

As already mentioned, our algorithm was not able to find feasible solutions to all problem instances. This mainly had two reasons: the first being the calculation time exceeding our pre-specified limit of 5 hours, the second being that the algorithm started without a feasible initial population. As for the calculation time, this might be due to flaws in our implementation, the use of inefficient built-in functions or the nature of our algorithm.

With regard to the lack of a feasible population, we noticed that our population initialisation had a hard time generating feasible solutions for complex problem instances, so that the actual genetic algorithm started with only an infeasible population. In some instances, our crossover, repair and education managed to create a feasible solution after some iterations. With at least one feasible solution, more feasible solutions were likely to follow. This shows that our algorithm has added value in these instances. In other instances, our algorithm did not manage to create any feasible

solution. Increasing the number of iterations might have solved this issue, but as the runtime was already quite high, this was not preferred.

Another note on the infeasibility, is that a solution can only be infeasible with respect to time deadlines. Whilst this is beneficial for the implementation and tractability, it does require good scheduling or else the algorithm will have difficulties generating feasible solutions.

6.2 Implementation

In its current implementation, scheduling is done inefficiently and often in the Hybrid Genetic algorithm. Suppose we change two chromosomes C_{First} and $C_{Seconds}$, where C_{First} is the chromosome that appears the first in the chromosome sequence. Then, rescheduling the whole sequence is not necessary: only reschedule the chromosomes C_{First} and its successors in the chromosome sequence. However, the current implementation does reschedule everything.

Our implementation of the genetic algorithm uses only one CPU, whereas genetic algorithms can possibly make excellent use of multi-threading. Say we have $T > 1$ threads available on our computer and we want to perform N iterations of our genetic algorithm. Then, instead of generating children one by one, we can generate a batch of T children simultaneously and insert them in the population at once. As the insertion in the population is quite cheap in computing resources, this should yield a total calculation time that is a little over $\frac{N}{T}$ of calculation time of the single-threaded implementation. If the number of threads is large compared to the population size however, a downside of multi-threading is that the solutions are based on worse parents than the solution of a single-threaded genetic algorithm. This can be countered by adding a few extra iterations, which lowers the benefits of multi-threading. Furthermore, since we lack the required knowledge of parallel computing in general and, more specifically, its implementations in `Python 3`, we avoided multi-threading and opted for the single-threaded alternative.

One last remark about our implementation, is that it made extensive use of `Python 3`'s `deepcopy` function. In hindsight, this turned out to be a very slow function and the total calculation times could have been decreased by avoiding this function or implementing a faster version ourselves.

7 Conclusion

In this paper we proposed a Hybrid Genetic algorithm for the E-MTPDPTWT. This variant of both the problem has, to our knowledge, not been solved in contemporary literature. Furthermore, no previous implementations of genetic algorithms for PDP's with transfers and recharging stations exist.

In Section 3, the E-MTPDPTWT was described mathematically. Then, in Section 4.1, a new representation for a solution with the addition of the dependence forest was introduced. The dependence forest is used in different phases in the algorithm: Scheduling, repairing, education, crossover. This research has shown the representation to be very effective in representing a routing problem that contains transfer options. Furthermore, in Section 4.5.1, the novel Sub Group Exchange Crossover (SGXX) was presented, aimed at preserving good properties of parent solutions. The crossover has shown to have both good diversification and intensification characteristics.

The algorithm has been executed on 594 problem instances on the University of Groningen's Peregrine computing cluster. Calculation times formed a serious issue for the algorithm, as well as generating decent initial populations. On the problem instances for which our algorithm found feasible solutions, our results show that solution quality improved compared to the initial population, by up to 37.5%. The algorithm was also able to improve the best solution even more, as more iterations were performed. Finally, in some instances where there was no prior feasible population, the Hybrid Genetic algorithm managed to find feasible solutions.

7.1 Future research

An improvement that could be added to our algorithm is solution transferring, based on transfer learning which is commonly used in machine learning (Pan and Yang, 2010). As our solutions can only be infeasible with respect to time constraints, one could first solve the problem without deadlines and then use these solutions as (part of) the initial population for the problem with deadlines.

Currently, the crossover uses random cuts to recombine chromosomes of two parents. This robust random method is used in order to diversify the population, though it is interesting to evaluate the impact of the introduction of a more selective method for the crossover as is done in Blocho and Nalepa (2017). The chromosomes of the first parent could be selected randomly whereafter the selection of the chromosomes of the second chromosome could be done with a critical searching procedure. This selection could for example complement as much missing locations with chromosomes of the second parent as possible in order to reduce the number of reparation operators.

Another suggestion for further research is related to the order of the different phases of the algorithm. For the operators used in the repair and education phase, a lot of time-consuming steps are necessary to check the feasibility of each operation. In future research, feasibility could be checked only before the child is inserted in the population. After the crossover phase one could for example parallel-educate a large number of different children in different ways. Then, in the end all children are checked on feasibility and goal value. This way, the time-consuming repair operators are not performed at all for these educated children and the feasibility checks are only performed in the end. The creation of a large number of educated children could make generating good feasible solutions much faster.

Another way to decrease the algorithm's calculation times, would be to implement it in a faster language. Whilst `Python 3` helps the user in building the algorithm quite fast, it is not the fastest language for the job. An implementation of the algorithm in a low-level object-oriented language like Java or C++ would most probably decrease the calculation times by quite a bit.

References

Bent, Russell and Pascal Van Hentenryck (2006). A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers & Operations Research 33*(4), 875 – 893. Part Special Issue: Optimization Days 2003.

Blickle, Tobias and Lothar Thiele (1996). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation 4*(4), 361–394.

Blocho, Miroslaw and Jakub Nalepa (2017). LCS-based selective route exchange crossover for the pickup and delivery problem with time windows. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pp. 124–140. Springer.

Cattaruzza, Diego, Nabil Absi, Dominique Feillet, and Thibaut Vidal (2014). A memetic algorithm for the multi trip vehicle routing problem. *European Journal of Operational Research 236*(3), 833 – 848. Vehicle Routing and Distribution Logistics.

Cortés, Cristián E., Martín Matamala, and Claudio Contardo (2010). The pickup and delivery problem with transfers: Formulation and a branch-and-cut solution method. *European Journal of Operational Research 200*(3), 711 – 724.

Hosny, Manar I. and Christine L. Mumford (2012). Constructing initial solutions for the multiple vehicle pickup and delivery problem with time windows. *Journal of King Saud University - Computer and Information Sciences 24*(1), 59 – 69.

Katayama, K, H Sakamoto, and H Narihisa (2000). The efficiency of hybrid mutation genetic algorithm for the travelling salesman problem. *Mathematical and Computer Modelling 31*(10-12), 197–203.

Li, Haibing and Andrew Lim (2003). A metaheuristic for the pickup and delivery problem with time windows. *International Journal on Artificial Intelligence Tools 12*(02), 173–186.

Masson, Renaud, Fabien Lehuédé, and Olivier Péton (2013). An adaptive large neighborhood search for the pickup and delivery problem with transfers. *Transportation Science 47*(3), 344–355.

Nalepa, Jakub and Miroslaw Blocho (2017). A parallel memetic algorithm for the pickup and delivery problem with time windows. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pp. 1–8. IEEE.

Nanry, William P. and J. Wesley Barnes (2000, February). Solving the pickup and delivery problem with time windows using reactive tabu search. *Transportation Research Part B: Methodological 34*(2), 107–121.

Pan, Sinno Jialin and Qiang Yang (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering 22*(10), 1345–1359.

Pankratz, Giselher (2005). A grouping genetic algorithm for the pickup and delivery problem with time windows. *Or Spectrum 27*(1), 21–41.

Parragh, Sophie N., Karl F. Doerner, and Richard F. Hartl (2008, Apr). A survey on pickup and delivery problems. *Journal für Betriebswirtschaft 58*(1), 21–51.

Phan, Dũng H. and Junichi Suzuki (2016, Feb). Evolutionary multiobjective optimization for the pickup and delivery problem with time windows and demands. *Mobile Networks and Applications 21*(1), 175–190.

Prins, Christian (2004). A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research 31*(12), 1985 – 2002.

Ropke, Stefan and David Pisinger (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science 40*(4), 455–472.

Schneider, Michael, Andreas Stenger, and Dominik Goeke (2014). The electric vehicle-routing problem with time windows and recharging stations. *Transportation Science 48*(4), 500–520.

Tarjan, Robert (1971). Depth-first search and linear graph algorithms. pp. 114–121.

Ting, Chuan-Kang and Xin-Lan Liao (2013). The selective pickup and delivery problem: formulation and a memetic algorithm. *International Journal of Production Economics 141*(1), 199–211.

Vidal, Thibaut, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei (2012). A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research 60*(3), 611–624.