Minimizing the number of rechargeable batteries for electrical ships using bin packing with conflicts, item fragmentation and negative items

Hidde Stek

Bachelor Thesis
Delft University of Technology | Electrical Engineering, Mathematics and Computer Science

July 2024

Supervisors: T.M.L. Janssen and R.J. Fokkink

Abstract

This thesis proposes a new solution to minimize the number of batteries needed to schedule ships depending on the presented harbour. The proposed solution is an extension to the bin packing problem. Item Fragmentation (BPP-IF) and Conflicts (BPPC) were added to allow ships to carry more batteries and to prevent batteries from being scheduled for multiple ships, respectively. Negative Items (BPP-NI) were added to allow for the batteries to be charged when they were not scheduled to be used. The combination of the three bin packing extensions (BPPC-IF-NI) formed the basis for the proposed solution. By a polynomial time reduction we showed that BPPC-IF-NI is NP-hard and therefore cannot be analytically solved in polynomial time. The solution can be approximated with an updated version of the heuristic presented by Ekici (2022).

The main idea in this heuristic approach is to generate a set of independent sets of the conflict graph and determine the packing of the bins using the independent sets with the help of the integer linear program ISB-IP. An iterative framework is used where L sets of T independent sets are generated at each iteration and considered together with the independent sets used in the best solution to obtain a better solution. After testing many possible combinations of L and T, it was found that $L = 50$ and $T = 10$ gives the best results, meaning that half of the number of the initial independent sets (L-value) containing a maximum of half the number of items in each set (T-value) where added in each iteration. When using these parameters, it was found that the solution of the heuristic was the same as the true optimal value. This was checked by inserting the power set without conflicts into the ISB-IP. The performance of the proposed solution was tested on generated instances. It was seen that there is no correlation between the computation time and the number of conflicts. It was also found that there was a strong correlation between the number of conflicts and the number of batteries used in the solution, without any clear outliers.

The proposed solution is able to very quickly approximate the minimum number of batteries needed to schedule the ships for the given harbour. Since most instances are approximated in a couple of seconds, it can also be used to update schedules when complication arise on a smaller scale. This would allow harbours to switch to electric ships, while reducing the cost by minimizing the number of expensive ship batteries.

Contents

1 Introduction 3

2 Bin Packing Theory and Literature 4
- 2.1 Bin Packing Problem Overview . . . 4
- 2.2 Item Fragmentation . . . 4
- 2.3 Conflicts . . . 6
- 2.4 VSBPPC-IF . . . 6
- 2.5 BPP-NI . . . 6
- 2.6 BPPC-IF-NI . . . 7

3 Generating input data 8
- 3.1 Assumptions . . . 8
- 3.2 Generating the data . . . 8
- 3.3 Conflict graph . . . 9

4 Proposed solution 10

5 Algorithm 12
- 5.1 Generate independent sets . . . 12
- 5.2 Generate additional random independent sets . . . 12
- 5.3 Iterative Independent Set Selection Heuristic . . . 13

6 Results 14
- 6.1 Theoretical versus computational optimum . . . 19
- 6.2 Computation time against number of conflicts . . . 20
- 6.3 Number of batteries used depending on the number of conflicts . . . 21

7 Conclusion 23

8 Discussion 24

9 Appendix 25

1 Introduction

The maritime sector presently accounts for about 2.8% of all global greenhouse gases (GHG) emissions (Faber et al., 2020). Innovations in battery capacity now make it possible to create container-sized batteries that can facilitate the energy needed for inland marine voyages. Vessels powered by these container batteries already exist and are used on a small scale (Zero Emission Services). The charging of the battery containers does not have to be done with the battery on board. The harbour can facilitate switching the batteries for charged ones which significantly reducing the waiting time of the electrified vessels and the number of occupied harbour spots. Switching the batteries can be done during the usual loading and reloading of cargo. Charging container batteries is done by attaching them to a specifically designed charging station. The harbour should thus have a certain amount of extra batteries and a number of charging stations at their disposal. The charging stations are much cheaper then the battery containers. It therefore makes sense to try to optimize the number of batteries in a system (and not the number of charging stations). As mentioned before, the battery container powered ships are currently only used on a small scale. To model such a small scale, let us consider only a single port in the waterway system that is equipped with charging stations. Thus all batteries must be charged at this port. Furthermore, we assume the routes of the vessels and the associated energy consumption are predetermined, that currently this is the only port in this system that supplies battery containers and it has a large amount of charging stations. The battery containers can be considered distinct entities, separate from the freight vessels. The goal is to find a schedule of service and charging that minimizes the number of battery containers in the system.

This thesis addresses this problem by introducing a new customized ship battery scheduling algorithm using bin packing. Bin packing is a proven approach in resource allocation problems, but its inherent NP-completeness makes it a challenging problem to solve. This thesis models ships as items with specific battery requirements and scheduling constraints, using an Integer Linear Programming (ILP) formulation based on the bin packing problem with conflicts, item fragmentation and negative items. An algorithm is presented that minimizes the number of batteries used while meeting the operational requirements of ship scheduling. The following research questions are posed: How can rechargeable ship batteries be minimized given a set of trips. How can the trips be scheduled using the minimum number of rechargeable batteries. What are the practical implications of the model for harbour operations?

This thesis is important as it contributes to more sustainable and cost-efficient harbour operations, providing a solution to the challenges of integrating battery-powered ships into existing logistics networks. We employ a combination of theoretical modelling and practical evaluation to demonstrate the effectiveness of our algorithm. Existing findings on bin packing will be used as it has been well researched before, as well as new finding including the use of items with a negative weight to allow for the emptying of batteries. In practise this allows the ship battery scheduling to be modelled as a bin packing problem, since the use of negative items allows for charging of the batteries. As far as the literature review goes, negative items have not been used before in this context. The contributions presented in this paper are summarized as follows:

- A new variant on bin packing is presented. This BPP includes items with a variable negative weight. This allows for the emptying of the bins.
- A practical variant of BPP is presented to deal with the specialized ship battery scheduling problem (SBSP). This makes it essential to have the ability to empty the bins if necessary.
- The heuristic algorithm of Ekici (2022) is enhanced and rewritten to include negative items. This way it can be used as a specialized algorithm to schedule ship batteries.
- The performance of the algorithm is tested against the theoretical optimum, among other methods.

In the following chapters of this report, we present the relevant literature and theory about bin packing (Chapter 2), the generated data used to simulate a harbour (Chapter 3), the ILP that holds the constraints of the problem (Chapter 4), the algorithm design (Chapter 5), and the results are presented analysed (Chapter 6). Finally, we offer conclusions and suggestions for future research (Chapter 7).

2 Bin Packing Theory and Literature

Bin Packing Problem (BPP) is one of the most widely-studied problems in the field of combinatorial optimization. The optimization problem looks to minimize the number of bins used to pack certain items with different sizes. By looking at the ship battery scheduling problem (SBSP) as a BPP, we can use its implications to get a solution for the desired problem. Normal bin packing is of a very general form, that is not always a good resemblance of the problem you are trying to model. That is why a lot of extensions have been studied. Variable-Sized Bin Packing Problem with Conflicts and Item Fragmentation (VSBPPC-IF) (Ekici, 2022) is build using three different BPP extensions; with conflicts (BPPC), item fragmentation (BPPIF) and variable-sized bins (VSBPP). This BPP variant will form the basis for the proposed solution of the ship battery scheduling problem. Item fragmentation allows for multiple batteries per ship and the conflicts ensure that ships cannot use the same battery if they have overlapping time slots. We are now left to add the ability to charge a battery, to complete the bin packing extension to model SBSP. This will be done with items with a negative weight, to empty the bin.

This chapter will explain these extensions, along with their implications. We will also look at how we can formally phrase our ship battery scheduling problem as BPP. Then we can take a look at the proposed solution of Ekici (2022), but without the need to have variable bin sizes. Finally, everything will be combined to create the new BPP variant, bin packing problem with conflict, item fragmentation and negative items (BPPC-IF-NI).

2.1 Bin Packing Problem Overview

The standard bin packing problem involves fitting a set of items of different sizes into a finite number of bins with a fixed capacity, aiming to minimize the number of bins used. In the context of ship battery scheduling, batteries are analogous to bins, and trips are analogous to items. These terms will be used interchangeably. When a battery is empty, its corresponding bin is considered full. An item is packed when the total energy demand for a trip is met. If a trip requires more than one battery, the item must be fragmented. Overlapping trips share a conflict. Additionally, an empty battery can be recharged using items with negative weight, referred to as negative items. These concepts will be further explained in the following chapter.

Now, we know that the presented problem is a bin packing problem. Vazirani (2001) proves any algorithm approximating this problem has an approximation ratio of at best 3/2. This ratio indicates the worst-case scenario for the algorithm's performance. An approximation ratio of 3/2 guarantees that the algorithm will not use more than 50% additional bins compared to the optimal solution. It provides a bound on how far from the optimal solution the algorithm's solution can be, ensuring a level of efficiency and reliability in practical applications where finding the exact optimal solution may be computationally infeasible. The theorem and proof of Vazirani (2001) are provided to support these claims. The proof is based on the the partitioning problem, where a set of integers is divided into two subsets with equal sums. The goal is to determine if there exists a way to partition the set into two subsets such that the sum of the elements in each subset is the same. This problem is NP-complete.

Theorem 2.1 (Vazirani (2001))**.** *For any $\epsilon > 0$, there is no approximation algorithm having a guarantee of $3/2 - \epsilon$ for the bin packing problem, assuming $\mathbf{P} \neq \mathbf{NP}$.*

Proof. If there were such an algorithm, then we show how to solve the **NP**-hard problem of deciding if there is a way to partition n nonnegative numbers $a_1, ..., a_n$, for $i = 1, 2, 3, ..., n$ into two sets, each adding up to $\frac{1}{2}\sum_i a_i$. This is the same as deciding if we can pack items $i = 1, 2, 3, ..., n$ with weight $a_1, ..., a_n$ into 2 bins with size $\frac{1}{2}\sum_i a_i$. If the answer is 'yes' the $3/2 - \epsilon$ algorithm will have an optimal packing, and thereby solving the partitioning problem. □

2.2 Item Fragmentation

In BPPIF, items can be fragmented, and each fragment can be packed into a separate bin. Item fragmentation in ship battery scheduling refers to the ability to split the energy demand for a trip into smaller pieces divided over several batteries. A fragmentation in this case is size-preserving, meaning that the size of all fragments of an item is equal to the size of the original item. This way trips are not restricted to the capacity of one battery. Item fragmentation is not only useful if a trip requires more than one battery, but also to more efficiently allocated the batteries in a system. This could result in a minimized solution with even fewer batteries needed for a given harbour, than without item fragmentation. If you however, add the ability to fragment items without any restrictions, the problem becomes trivial.

Theorem 2.2. *BPP-IF is trivial, if:*

$$\textit{number of fragmentations item } i \geq \lfloor \frac{w_i}{W} \rfloor + 1$$

,where w_i is the size of item i and W is the bin capacity.

Proof. If the number of fragmentations item $i \geq \lfloor \frac{w}{W} \rfloor + 1$, then we can fragment item i even if $w_i \leq W$. This allows us to take all items and put them in a list in an arbitrary order. Then it is always possible to fragment the list into perfectly filled bins, as can be seen in figure 1. The maximum number of bins needed is $\lceil \frac{\sum_i w_i}{W} \rceil$. Thus making BPP-IF trivial. □

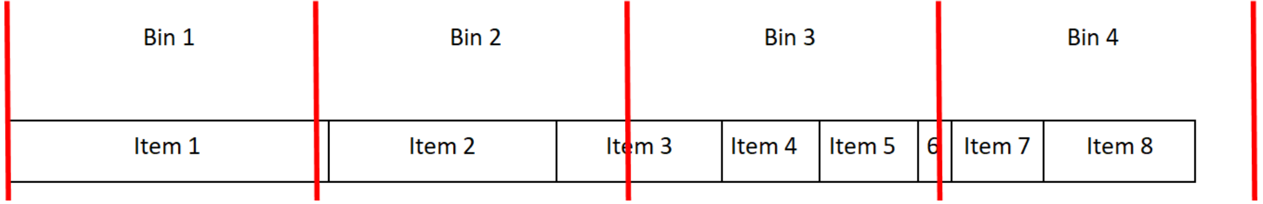

Figure 1: BPP-IF without restrictions

This result was initially found during this thesis. It was later found that this result had already been proven by Bertazzi et al. (2019).

We have seen that in the case of theorem 2.2, the problem becomes trivial. Now we are going to take a look at BPP-IF, but this time the number of fragmentation for item i are restricted to $\lfloor \frac{w_i}{W} \rfloor$. The remaining problem is not trivial, it is in fact still bin packing.

Theorem 2.3. *BPP-IF with items with $w \geq W$ and the number of fragmentations item $i = \lfloor \frac{w_i}{W} \rfloor$, is trivial to reduce to standard BPP, which is NP-hard.*

Proof. Take all items i^* for which $w_i \geq W$. Pack bin-size-fragments in its own bin until the remaining weight is smaller than W. Now all remaining items can be defined by $\delta_i = w_i \mod W$, thus $\delta_i \leq W, \forall i$, since these items cannot be fragmented anymore, due to the restriction, this results in the standard BPP. □

Since the resulting set of item with $\delta_i \leq W, \forall i$ is BPP, there is an algorithm with approximability ratio 3/2. Then we can compute the approximation ratio. We use that Opt_{BPP-IF} is the optimal value for bin packing with $\lfloor \frac{w_i}{W} \rfloor$ fragmentations for item i applied to items i with weight w_i. This is equal to the optimal value for bin packing Opt_{BPP} applied to items i with weight δ_i plus the number of bins n^* that are packed optimally by the fragmentations. And since Opt_{BPP} can be approximated by an algorithm with approximability ratio 3/2. We get $\text{Alg}_{BPP} = \frac{3}{2}\text{Opt}_{BPP}$, where Alg_{BPP} is the approximation value given by the algorithm.

Theorem 2.4. *The approximability ratio for BPP-IF, with restrictions on the fragmentations is*

$$Ratio_{BPP-IF} = \frac{n^* + \frac{3}{2}Opt_{BPP}}{n^* + Opt_{BPP}}$$

For any n^*, where n^* is the number of bin-size-fragments in its own bin.

Proof. The number of optimally packed bins automatically increases, due to the ability to fragment an item such that a fragment is equal to the bin size. If you fragment all items until all remaining items are smaller than a bin, the remaining items can be packed according to standard bin packing, which is NP-complete and it cannot be approximated with an approximation algorithm with a ratio smaller than 3/2 unless P = NP (Vazirani, 2001). Now let n^* be the number of items that are optimally packed in its own bin. The remaining items can be packed using normal bin packing which has an approximation rate of 3/2. Therefor the approximation rate of BPP-IF is

$$\text{Ratio}_{BPP-IF} = \frac{\text{Alg}_{BPP-IF}}{\text{Opt}_{BPP-IF}} = \frac{n^* + \text{Alg}_{BPP}}{n^* + \text{Opt}_{BPP}} \leq \frac{n^* + \frac{3}{2}\text{Opt}_{BPP}}{n^* + \text{Opt}_{BPP}}$$

□

We have now seen that the addition of item fragmentation, can often lead to trivial problems. In the following chapter, conflicts are introduced. By combining BPP-IF and BPPC, the fragmentations don't result in a trivial result (Ekici, 2021).

2.3 Conflicts

In BPPC, certain pairs of items cannot be packed into the same bin, and the goal is to minimize the number of bins used. In the realm of ship battery scheduling, conflicts between trips occur when their time slots overlap. In order for two items to be packed into the same bin, their corresponding time slots, the time between their departure and arrival, have to be disjunct. BPPC is introduced by Jansen (1999). They modify the well-known algorithms proposed for BPP to address the item conflicts and hybridize these algorithms with vertex colouring heuristics. Ekici (2021) provides a proof to show that item fragmentation and conflicts combines, result in a NP-hard problem.

Theorem 2.5 (Ekici (2021))**.** *BPPC-IF is NP-hard.*

Proof. The proof follows by a reduction from the 3-colouring problem (Garey and Johnson, 1979). An instance of 3-colouring is as follows:
Instance: An undirected graph $G(V, E)$ with vertex set V and an edge set E.
Question: Can the vertices of graph G be coloured using three colours such that no two adjacent vertices have the same colour? We prove that the decision version of BPPC-IF is NP-complete by creating an instance of BPPC-IF from an instance of 3-colouring (in polynomial time) as follows. For each $i \in V$, we create an item with unit weight ($w_i = 1$) and assume that items i and j are conflicting if the corresponding vertices in graph G share a common edge, thus items can be packed in the same bin as long as they don't share an edge in G. Finally, we set the bin capacity to $\|V\|$. A solution using 3 bins to the BPPC-IF instance exists if and only if a solution to the 3-colouring instance exists. □

2.4 VSBPPC-IF

VSBPPC-IF is a practical variant of the Variable-Sized Bin Packing Problem (VSBPP) that has applications in heterogeneous vehicle routing, memory allocation in parallel processing and operating file systems. In this variant, certain pairs of items have conflicts and conflicting items cannot be packed into the same bin. Moreover, items can be fragmented, and each fragment can be packed into a separate bin. We call this variant the Variable-Sized Bin Packing Problem with Conflicts and Item Fragmentation (VSBPPC-IF). A lower bounding scheme is proposed by (Ekici, 2022), based on a maximal clique of the conflict graph. The proposed lower bounding scheme provides better lower bounds especially when the conflict graph is dense. A novel heuristic algorithm is developed, called the Iterative Independent Set Selection Heuristic (IISSH). The main idea in this heuristic approach is to generate a set of independent sets of the conflict graph and determine the packing of the bins using the independent sets with the help of a mathematical model. An iterative framework is given, where a set of independent sets are generated at each iteration and considered together with the independent sets used in the best solution to obtain a better solution.

2.5 BPP-NI

Negative items have been researched before, but never in the context of battery charging. Negative items in ship battery scheduling are used to add the ability to charge the batteries that have been used. If a battery is (partly) empty, it can be recharged when it is not being used. This part is crucial for the real world application of ship battery scheduling. Different from the items that resemble trips, the weights of the negative items are not predetermined. This is because the time the batteries can be charged is dependent on the final schedule. And every possible charging time is possible, as long as it does not charge a battery when it is being used for a trip or overcharge a battery. The latter means that a battery cannot be charged more than its initial capacity. So although they are not predetermined, they are restricted. There are two different ways a battery can be charged.

- *Option 1:* Charge during the time that an item is not packed. For example, if item i with departure time 09:00, arrival time 10:00 is not scheduled for battery B, battery B can be charged between 09:00 and 10:00. This is called o_i.
- *Option 2:* Charge in between possible packed items. For example, we have items i and j, with time slots 09:00-10:00 and 11:00-12:00 respectively. Regardless of whether i and/or j are scheduled for battery B, battery B can be charged between 11:00 and 12:00. This is called o_{ij}.

2.6 BPPC-IF-NI

If there are no restrictions imposed on the use of the negative items, then BPP-IF-NI becomes trivial. Since this variant of bin packing does not have conflicts, we can pack all items into the same bin, adding the required number of negative items to cancel the surplus of weight.

Theorem 2.6. *BPP-IF-NI is trivial and can be packed in an arbitrarily small bin.*

Proof. Let $n \in \mathbb{R}$. Then for any set of items $i = 1, 2, ..., n$ with weight w_i with $i \in 1, 2, ..., n$. Take another set of n items with weights $o_i = -w_i$ with $i \in 1, 2, ..., n$. The lack of conflicts allows us to packed all positive and negative items in the same bin, resulting in a total net weight of zero. □

If the conflicts are imposed, the BPP becomes hard to solve. Since batteries cannot be charged while the are being used, this would result in a conflict between the positive and negative item. Also restrictions are imposed on the negative items. Negative items has conflicts with every other item in the same time slot. The negative items have $0 \geq o_i \geq -W, \forall i$.

Theorem 2.7. *BPPC-IF-NI is NP-hard*

Proof. To prove that BPPC-IF-NI is NP-hard, we show a polynomial-time reduction from the Variable Sized Bin Packing Problem with Conflicts and Item Fit (BPPC-IF), which is already known to be NP-hard (Ekici, 2021). This can be done by forcing the weights of the negative items to $o_i = 0$. Therefore, the remaining problem is BPPC-IF. □

3 Generating input data

Since the use of batteries for shipping is only being used on a small scale, there is no available data for harbours of a more relevant size. Data has to be generated, so there is a way to test the scheduling algorithm presented in chapter 4. This chapter will talk about how this data was generated, what assumptions were made to model real life data, and a breakdown of the code.

3.1 Assumptions

In order to create the data, there are several components we need to simulate. Every trip needs to have a departure time, an arrival time, and an energy demand. For this thesis, a theoretical harbour is created to test the algorithm of chapter 4 on. The departure times are assumed to be between 06:00 and 03:00. For bigger harbours, such as the Rotterdam Port, there are ships departing and arriving at every hour during the day (Port of Rotterdam, 2024). To have a clear cut of at 03:00, allows for every batteries to be fully charged during the night. Another reason for this assumption, is the fact that this allows the algorithm to schedule a more limited amount of ships. If this assumption had not been made, the ships would constantly and consecutively arrive making the input set too large to compute. The energy demand per trip for this harbour is assumed to be bimodally distributed. This allows for the distinction of two groups, the short distance trips and the long distance trips. In real life, this could be caused by domestic trade and foreign trade. As a result of this, the data generating algorithm takes two different means and two standard deviations. So the specific needs can be modelled more closely to resemble the required real life situation. The arrival times need to be modelled. It is assumed that the required time to finish a trip, is proportionate to the required energy. This is not necessarily an assumption based on real life data, since the current of a river can greatly impact the relation between energy demand and expected duration of a trip. Lastly, it is assumed that the ships depart and arrive at the predetermined times, using exactly as much energy as predetermined. This is not a realistic assumption, but a fully realistic model of reality is deemed outside of the scope of the research.

3.2 Generating the data

Based on these assumption, an algorithm can be made to generate the desired data. This data will serve as the input for the ILP presented in chapter 4. The required input with its corresponding symbols are introduced in table 1. The pseudo code shows how the data is generated. Where μ_i and σ_i can vary, to get different data instances.

Symbol	**Definition**
V	Set of all items
$\|V\|$	Number of ships
W	Capacity of bin
w_i	Weight of item i
t_i^b	Departure time for trip i
t_i^e	Arrival time for trip i
μ_1	Mean of distribution 1
μ_2	Mean of distribution 2
σ_1	Standard deviation of distribution 1
σ_2	Standard deviation of distribution 2
$G = (V, E)$	Conflict graph

Table 1: Table of input symbols

Algorithm 1 Generate Ship Data

1: **Input:** $\|V\|, \mu_1, \mu_2, \sigma_1, \sigma_2$
2: **Output:** w_i, t_i^b, t_i^e
3: **Generate** t_i^b
4: Generate $\|V\|$ random t_i^b uniformly distributed between 6 and 27
5: Sort in ascending order
6:
7: **Convert** t_i^b **to Hours and Minutes**
8: Extract the integer part of the t_i^b as hours
9: Calculate the fractional part of t_i^b and convert it to minutes
10: Combine the hours and minutes into a single time format (e.g., 14.30 for 14:30)
11:
12: **Generate** w_i
13: Generate w_i for $i = 1, ..., 1/2\|V\|$ using a normal distribution with μ_1 and σ_1
14: Generate w_i for $i = 1/2\|V\| + 1, ..., \|V\|$ using a normal distribution with μ_2 and σ_2
15: Let minimum of w_i be 20 $\forall i$
16:
17: **Calculate** t_i^e
18: Convert t_i^b to minutes (e.g., 1:30 to 90)
19: Add w_i (in minutes) to t_i^b (in minutes)
20: Convert t_i^e back to the hour and minute format
21: **Return** w_i, t_i^b, t_i^e

3.3 Conflict graph

The departure times, arrival times and weights of the items are all generated. Based on this, the conflict graph can be build. This is necessary to prevent ships with overlapping time slots to use the same battery. Algorithm 2 builds the conflict graph.

Algorithm 2 Create Conflict Graph

1: **Input:** w_i, t_i^b, t_i^e
2: **Output:** $G = (V, E)$
3: Create an empty graph $G = (0, 0)$
4: $counter = 0$
5: Add a node in $G, \forall i \in V$
6: **for** i in range$(0, \|V\|)$: **do**
7:     **for** j in range$(i + 1, V)$ **do**
8:         **if** $(t_i^b, t_i^e) \cap (t_j^b, t_j^e) \neq \emptyset$ **then**
9:             $G.add_edge(i, j)$
10:             $counter+ = 1$
11:         **end if**
12:     **end for**
13: **end for**
14: **Return** $G = (V, E)$

To visualize the output of algorithm 2, an example is provided in figure 2 with the following departure and arrival times. Departure times: 1:00, 2:00, 3:00, 4:00, 6:00, 10:00, 11:00, 12:00. Arrival end times: 4:00, 5:00, 6:00, 7:00, 9:00, 13:00, 14:00, 15:00

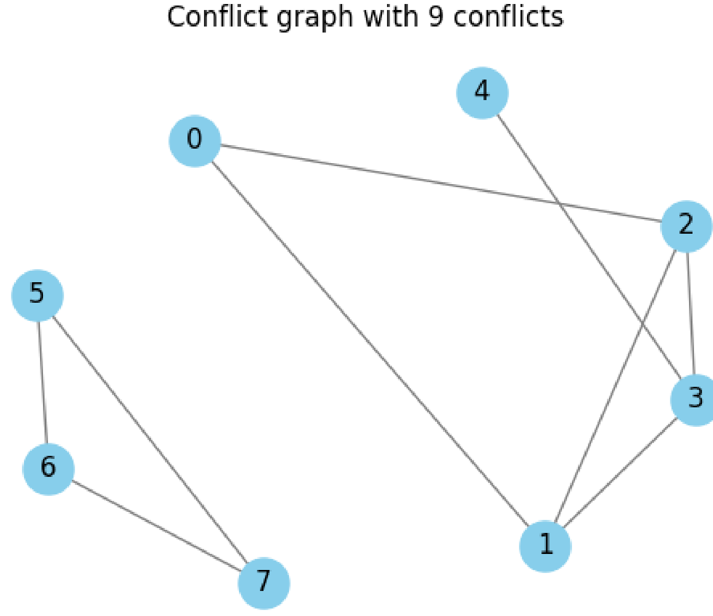

Figure 2: Example conflict graph for the given instance

4 Proposed solution

In this chapter, the proposed solution is presented. The heuristic is based on the Iterative Independent Set Selection Heuristic (IISSH). Where the main idea in the proposed solution approach is to (i) generate a set of independent sets of the conflict graph, V^p = Set of items in independent set p, P = Set of all independent sets and (ii) determine the bins used to pack the items in each independent set and the fragment size of each item by solving an integer linear program (Ekici, 2022). How the independent sets are generated will be explained in chapter 5. We first present the mathematical model solved to determine the bin packing solution, using the independent sets generated and then discuss the proposed implementation of the solution approach. In the proposed model, we use the following decision variables.

$$x_p = \begin{cases} 1, & \text{if independent set } p \text{ is used} \\ 0, & \text{otherwise.} \end{cases}$$

f_{ip} = Size of fragment of item i packed into a bin using independent set p

o_{ip} = Size of negative item i packed into a bin using independent set p

o^*_{ip} = Size of negative item between item i an i+1 i packed into a bin using independent set p

$$g_{ip} = \begin{cases} 1, & \text{if charging occurs instead of trip } i \text{ packed into a bin using independent set } p \\ 0, & \text{otherwise.} \end{cases}$$

M = Some sufficiently large number ('infinity')

Using these decision variables, we propose the following integer linear program to determine the content of the bins used to pack the items:

$$\textbf{ISB-IP:} \min \sum_{p \in P} x_p \tag{1}$$

$$\text{s.t.:} \sum_{p \in P | i \in V^p} f_{ip} = w_i \qquad \forall i \in V^p \tag{2}$$

$$f_{ip} + w_i g_{ip} \leq w_i \qquad \forall i \in V^p, \forall p \in P \tag{3}$$

$$o_{ip} - W g_{ip} \leq 0 \qquad \forall i \in V^p, \forall p \in P \tag{4}$$

$$\sum_{i \in V^p | t_i^e \leq t_k^b} \left[f_{ip} - o_{ip} - o_{ip}^* \right] \geq 0 \qquad \forall k \in V^p, \forall p \in P \tag{5}$$

$$\sum_{i \in V^p | t_i^e \leq t_k^b} \left[f_{ip} - o_{ip} - o_{ip}^* \right] + f_k \leq W \qquad \forall k \in V^p, \forall p \in P \tag{6}$$

$$\sum_{i \in V^p} f_{ip} + o_{ip} + o_{ip}^* + g_{ip} \leq M x_p \qquad \forall p \in P \tag{7}$$

$$x_p \in \{0, 1\} \qquad \forall p \in P \tag{8}$$

$$g_{ip} \in \{0, 1\} \qquad \forall i \in V^p, \forall p \in P \tag{9}$$

$$f_{ip} \geq 0 \qquad \forall i \in V^p, \forall p \in P \tag{10}$$

$$o_{ip} \geq 0 \qquad \forall i \in V^p, \forall p \in P \tag{11}$$

$$0 \leq o_{ip}^* \leq \min \left\{ 120, t_{i+1}^b - t_i^e \right\} \qquad \forall i \in V^p, \forall p \in P \tag{12}$$

In this independent set-based integer program (ISB-IP), the objective function (1) minimizes the total number of bin used. All fragments of item i in every bin p must be equal to the weight of the item, this is enforced by constraints (2). To decide whether you can charge a battery using option 1, constraint (3) ensures g_{ip} is only 1 if $f_{ip} = 0$, this ensures you can only charge a battery p when it is not schedules for a trip i. Constraints (5) makes sure that a bin does not get packed past its capacity. This constraint is added after every item to make sure this constraints holds at all times. Constraints (6) is similar to constraints (5), but this ensures that a bin is not charged pasted its maximum capacity. This constraint must also holds at all times, so it is added after every possible item. Constraints (7) make sure a bin cannot be packed or charged if a bin is not used, since all variables are positive. Lastly, constraint (12) bounds o_{ip}^* to only charge (using option 2) as much as is possible between two items.

The solution of this ILP depends on the independent sets used in the input. If we would use the power set of all items as the input sets, we would obtain the theoretical minimum. This would however require too much computational time. In chapter 5, we explain how the sets are selected. The proposed solution consists of three algorithms, the last one is iterative. This iteration step allows us to look at a lot of different input sets.

5 Algorithm

The ISB-IP needs to be solved to get a solution for the ship battery scheduling problem. It has also been said that the sets used as input, determine the solution given by the ISB-IP. It is not guaranteed that this is the theoretical minimum. To generate the independent sets of the conflict graph used as input, we use two algorithms, algorithm 3 to generate sets based on the number of conflict each item has , and algorithm 4 to generate additional randomly generated sets. Independent sets of the conflict graph mean that no items have a conflict in the set. Finally, algorithm 5 is used to combine algorithm 3, algorithm 4 and the ISB-IP, to get a solution. This chapter explains how these algorithms work.

5.1 Generate independent sets

Algorithm 3 Generate independent sets

1: **Input:** $V, w_i, W, G = (V, E)$
2: **Output:** P^I
3: $P^I = \emptyset$
4: **for** $i \in V$ **do**
5: $S = \{i\}$
6: $V' = V \setminus (\{i\} \cup \{j \in V : (i, j) \in E\})$
7: **while** $V' \neq \emptyset$ **do**
8: $G' = G[V']$
9: $d'_i = \|\{j \in V' : (i, j) \in E'\}\|, \forall i \in V'$
10: $i^* = argmin_{i \in V'}\{d'_i\}$
11: $S = S \cup \{i^*\}, V' = V' \setminus \{i^*\}$
12: $V' = V' \setminus \{j \in V' : (i^*, j) \in E'\}$
13: **end while**
14: Add $\lceil \frac{w_i}{W} \rceil$ copies of S to P^I
15: **end for**
16: **Return** P^I

Algorithm 3 is used to create independent sets of items. An item is selected and put in a set S. This item and all items that have a conflict with this item are removed from the list of possible items in step 6. A new conflict graph is created, only using the remaining items V' in step 8. This is the induced sub-graph $G[V']$ from G by V'. The item with the least amount of conflicts is added to the set S. This item and the items it has conflicts with, are again removed from the possible item. This continues until there are no more possible items to add to set S. Set S is then added to the set of independent sets P^I. Finally, enough copies of set S are added, so the total weight of item i can be packed. This way a feasible solution is ensured. By creating sets that are free of conflict and have enough copies to pack every item, it is always possible to find a solution.

5.2 Generate additional random independent sets

If more sets are considered as input for ISB-IP, the better the chances to get the minimum as a result. Algorithm 4 generates random sets. The number of sets you want to generate and the maximum number of items per set can be changed by new input parameters. The parameters and the pseudocode are provided below.

$$L = \text{number of random sets}$$
$$T = \text{maximum number of items per set}$$

Algorithm 4 Generate additional random independent sets

1: **Input:** $V, w_i, W, G = (V, E), L, T$
2: **Output:** P^R
3: $P^R = \emptyset$
4: **while** $\|P^R\| < L$ **do**
5: Generate $T' =$ random integer $\in (2, T)$
6: $S = \emptyset$
7: $V' = V$
8: **while** $\|S\| < T' \wedge V' \neq \emptyset$ **do**
9: Take random $i \in V'$
10: $S = S \cup \{i\}$
11: $V' = V' \cup \{i\}$
12: $V' = V' \backslash \{j \in V' : (i, j) \in E\}$
13: **end while**
14: $P^R = P^R \cup S$
15: **end while**
16: **Return** P^R

This algorithm generates L random independent sets, with a maximum of $T(\leq \|V\|)$ items per set. Lines 11 and 12 ensure the sets are free of conflicts (independent sets of the conflict graph).

5.3 Iterative Independent Set Selection Heuristic

The next algorithm is called the Iterative Independent Set Selection Heuristic and is the last algorithm to get the solution to the ship battery scheduling problem. Here we combine the previous algorithms and the ISB-IP.

Algorithm 5 Iterative Independent Set Selection Heuristic

1: **Input:** $V, w_i, W, G = (V, E), t_i^b, t_i^e, L, T,$ IterationLimit
2: **Output:** A feasible solution
3: $P = algorithm3(V, w_i, W, G)$
4: $Z_best = \infty$
5: $P_B = \emptyset$
6: $IterationCounter = 0$
7: **while** $IterationCounter < IterationLimit$ **do**
8: $IterationCounter += 1$
9: $P^R = algorithm4(V, w_i, W, G, L, T)$
10: $P = P + P^R$
11: Solve ISB-IP using P as input, let $opt = solution$
12: **if** $opt < Z_best$ **then**
13: $Z_best = opt$
14: Update P_B so it only contains the sets that were used to get the solution
15: **end if**
16: $P = P_B$
17: **end while**
18: **Return** Z_best

The independent sets P are generated using algorithm 3. The best solution is set to ∞. P_B is the set used for the current best solution, which starts as an empty set. L random sets are added using algorithm 4. Using both set of sets, ISB-IP is minimized. The result is called opt. If $opt < Z_best$, the best solution is updated to be opt. P_B is also updated to only include the sets that are used to get this solution. Theses steps are iterated until the iteration limit is reached. The more random sets are included and the more iterations are performed, the better the chances to get the actual minimum.

6 Results

In this chapter, we present the results of the computational study conducted to evaluate the performance of the proposed solution approach. All the algorithms are implemented using Python, and the computational experiments are carried out on a 64-bit PC equipped with an Intel Core i7-8850H CPU running at 4.3 GHz and 32 GB RAM.

This chapter will look at the performance of the algorithm. We are going to see how the parameters must be used to ensure a good result. This will be done by plotting multiple instances with different parameter combinations to see which ones perform best. This combination will be tested to see if it actually produces the theoretical optimum. We will check how long the algorithm takes to produce the solution depending on different input instances. Lastly, we will compare the input to the optimal solution. Since the number of conflicts in the data is the most important factor in the data, we will we looking at several different data instances with each a different number of conflicts.

Convergence of the solution The performance of the proposed solution depends on the number of iterations algorithm 5 performs. It is therefore important to know how fast the solution generally converges. In this chapter we are going to look at different combinations of L and T. L determines how many random sets are added each iteration. T determines the maximum number of items in these sets. It is therefore likely that these parameters can play a big role in the rate of convergence. The number of sets taken as input in the ISB-IP is also dependent on the number of independent sets generated by algorithm 3, this is in turn dependent on the conflicts. That is why these different values of L will be tested;

- 0 times the number of independent sets
- 1/4 times the number of independent sets
- 1/2 times the number of independent sets
- 1 time the number of independent sets

Figure 3 demonstrates how the algorithm might work, depending on the input and choice of parameters.

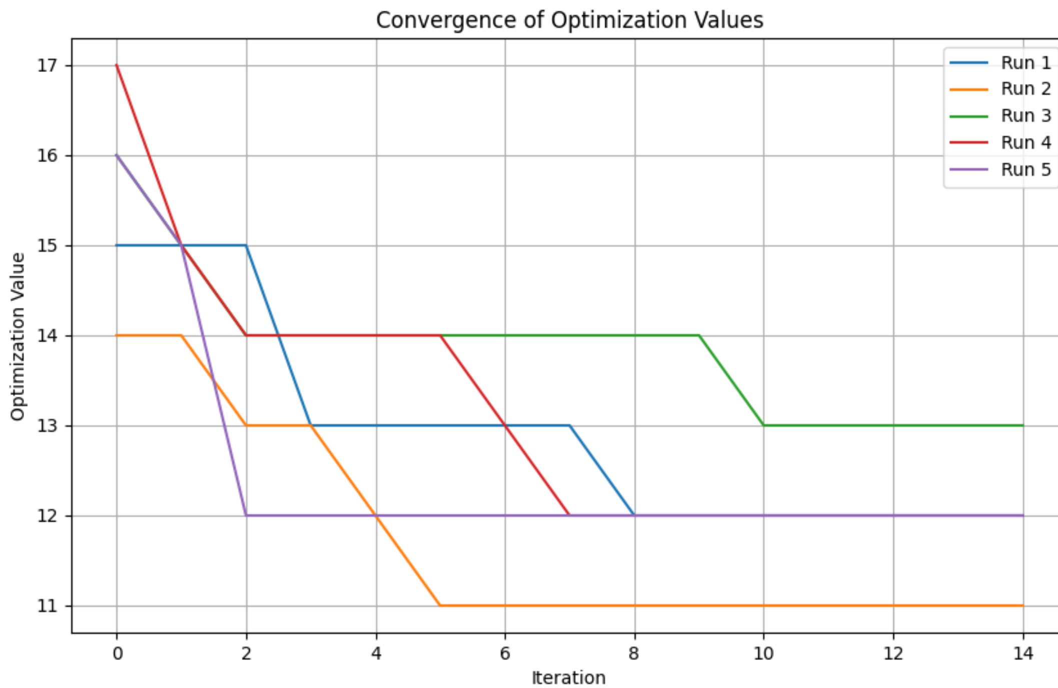

Figure 3: Example convergence graph with 20 items, 41 conflicts and 15 iterations

Another important factor is the value of T. The number of items in a set also depend on the number of conflicts, but the number of items in a set is always limited to the total number of items. That is why the values tested will be proportional to the number of items.

- 1/4 number of items
- 1/2 number of items
- 3/4 number of items

These 12 combinations of L-values and T-values will give useful insights on how many iterations to use. To compare the combinations, one instance is used as input for every combination. Because the L-values and the T-values are relative to the number of independent sets added and the number of items of an instance respectively, we assume the best combinations will also work relatively well on other instances. The data used to compare the combinations is as follows.

Departure times	**Arrival times**	**Required energy**
10.34	12.12	98.0
11.35	13.48	133.0
12.18	13.49	91.0
12.44	14.21	97.0
13.32	15.29	117.0
14.16	16.12	116.0
15.11	17.07	116.0
15.30	17.22	112.0
15.39	17.34	115.0
16.07	18.03	116.0
17.19	18.37	78.0
17.39	19.37	118.0
19.22	21.34	132.0
19.36	21.02	86.0
20.16	22.29	133.0
20.27	22.37	130.0
20.29	22.34	125.0
20.56	23.33	97.0
21.17	23.27	130.0
21.18	23.13	115.0

Table 2: Data instance with 53 conflicts

Important note chapter 5 explains that the solution given by the algorithm after each iteration is always at least as good as the previous one, i.e. it can never go up. However, in the figures below there are some graphs were this is the case. This is due to a time limit, to prevent the algorithm from endlessly computing. When the time limit is reached, the algorithm still produces the best solution that was found before the time limit was reached. These data points should not be taken into consideration, when analysing the data.

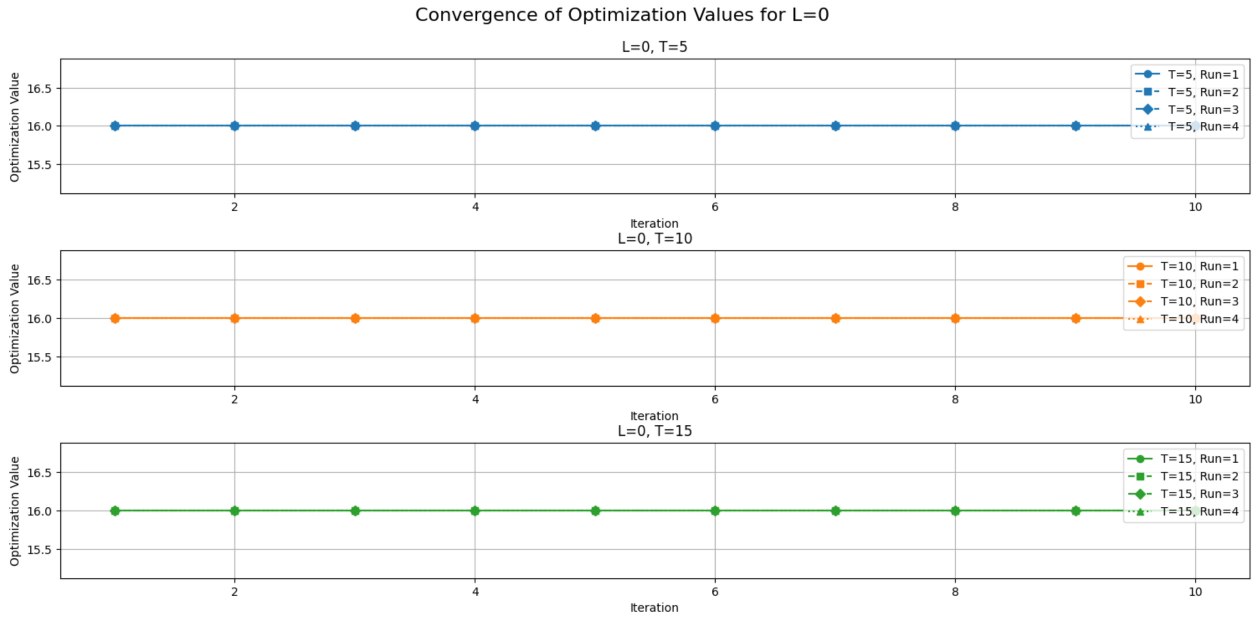

Figure 4: Convergence graph with zero additional random sets

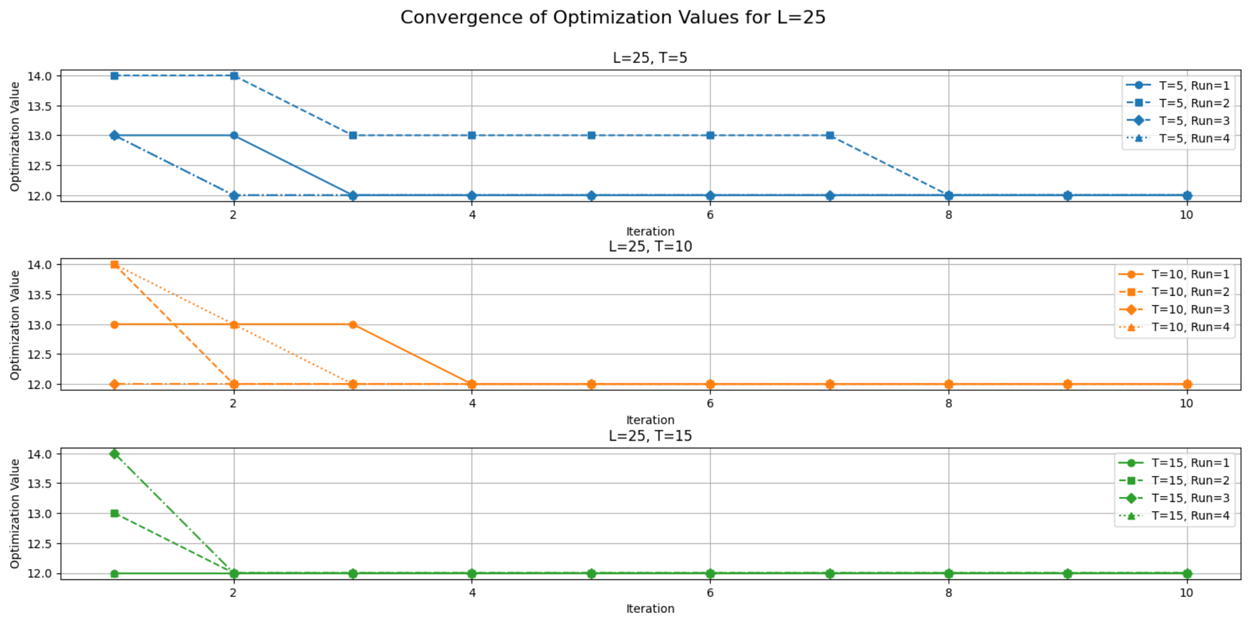

Figure 5: Convergence graph with 25 additional random sets

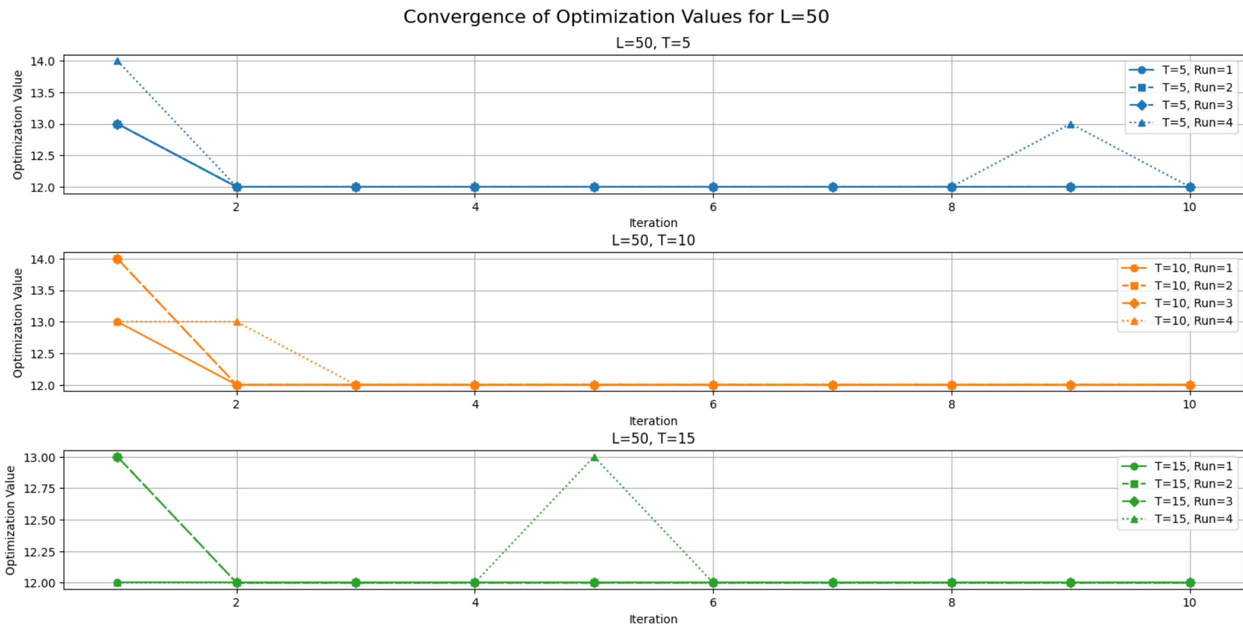

Figure 6: Convergence graph with 50 additional random sets

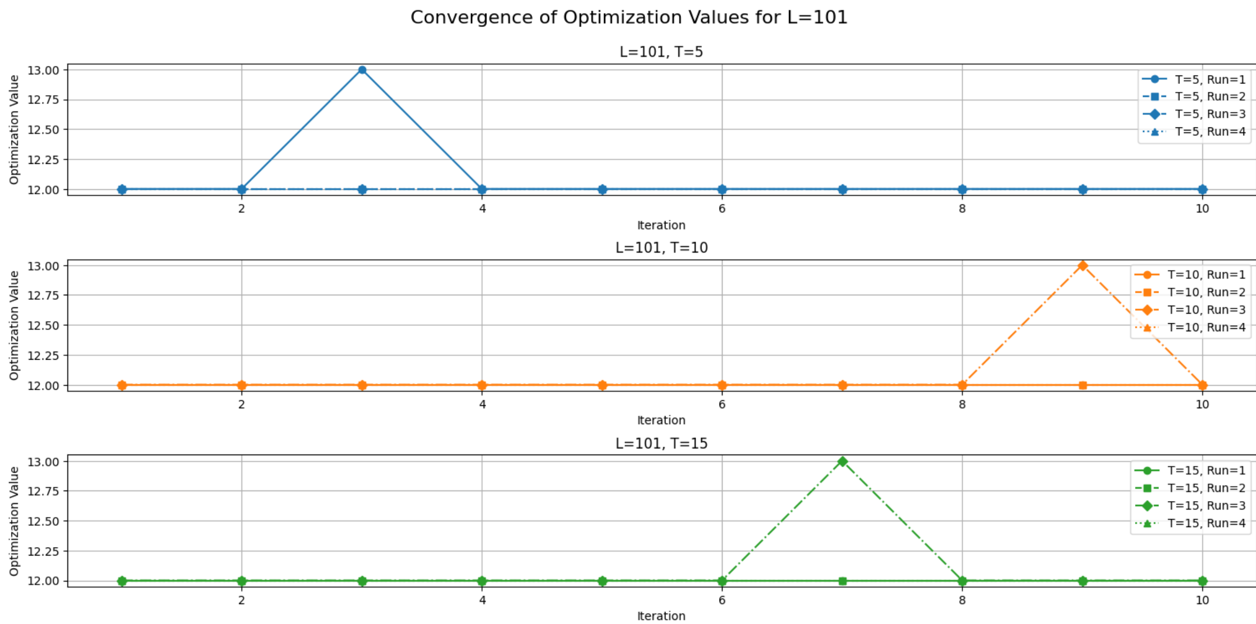

Figure 7: Convergence graph with 101 additional random sets

If we look at the figures above, a probable theoretical optimum would be 12. That the algorithm actually produces this optimum will be demonstrated in chapter 6.1. If we look at figure 4, we can see that the solution does not improve at all. This is expected as L=0 indicates that there are no random sets added, regardless of the iteration. This produces a solution of 16 batteries used, which is at least 25% more then the optimal solution. Figure 5 is more interesting. Here we can see the solution improves. Although every choice of T presented in the figure are performing relatively well, $T = 15$ results in the fastest and most reliable parameter choice. This is because the solution reaches the assumed optimum after 2 iterations, while $T = 5$ could need 8 iterations before reaching the assumed optimum. Figure 6 produces some good results, regardless of the choice of T, the algorithm returns the assumed optimum after only 3 iteration every time. Lastly, we look at figure 7 that adds the same number of random sets each iteration as the number of independent sets added by the algorithm. This results in the assumed optimal solution after the first iteration, regardless the choice for T.

There are several combinations of L and T that produce good results. However, due to computation time,

$L = 101$ will be less attractive to chose. L=25 and T=15, and L=50 and T=10 are both good choices. To be sure a good combination is chosen, we test these combinations on three new instances with a different number of conflicts. To ensure a good result, we will use use 5 iterations for both sets of parameters.

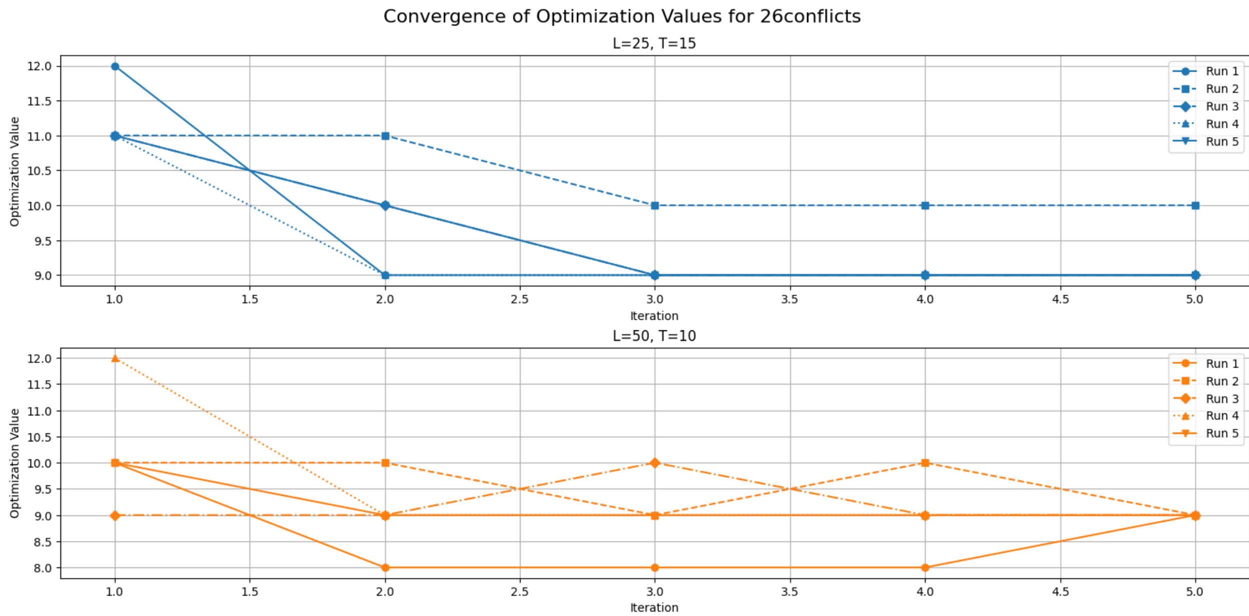

Figure 8: Convergence graph for 20 items with 26 conflicts

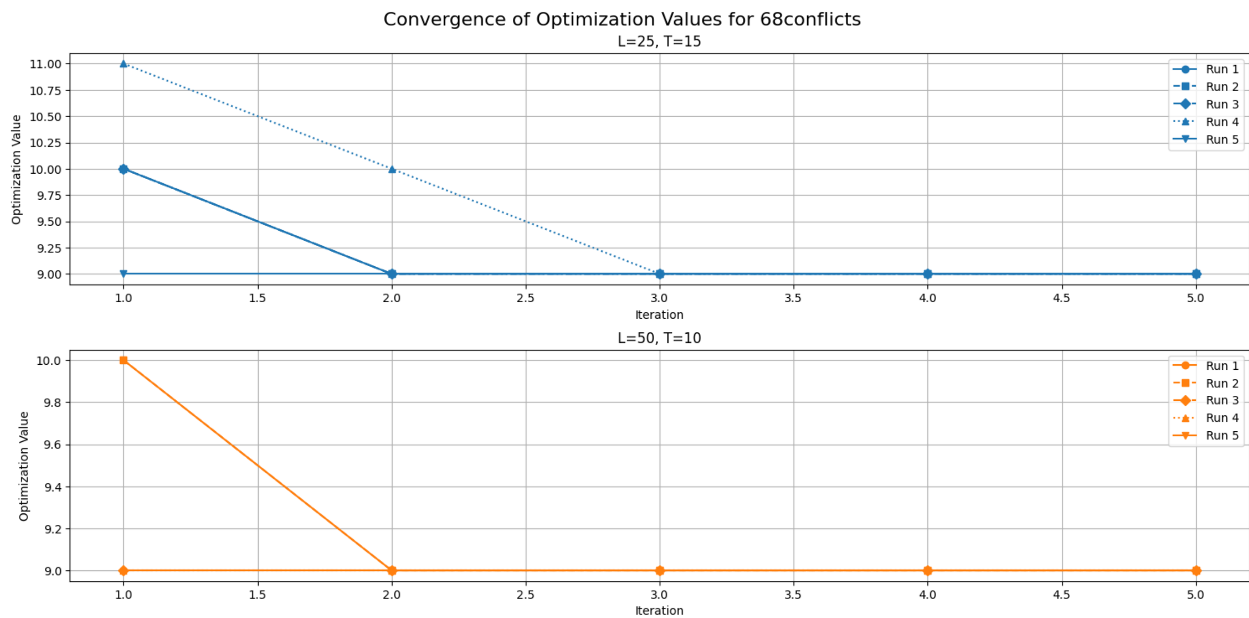

Figure 9: Convergence graph for 20 items with 68 conflicts

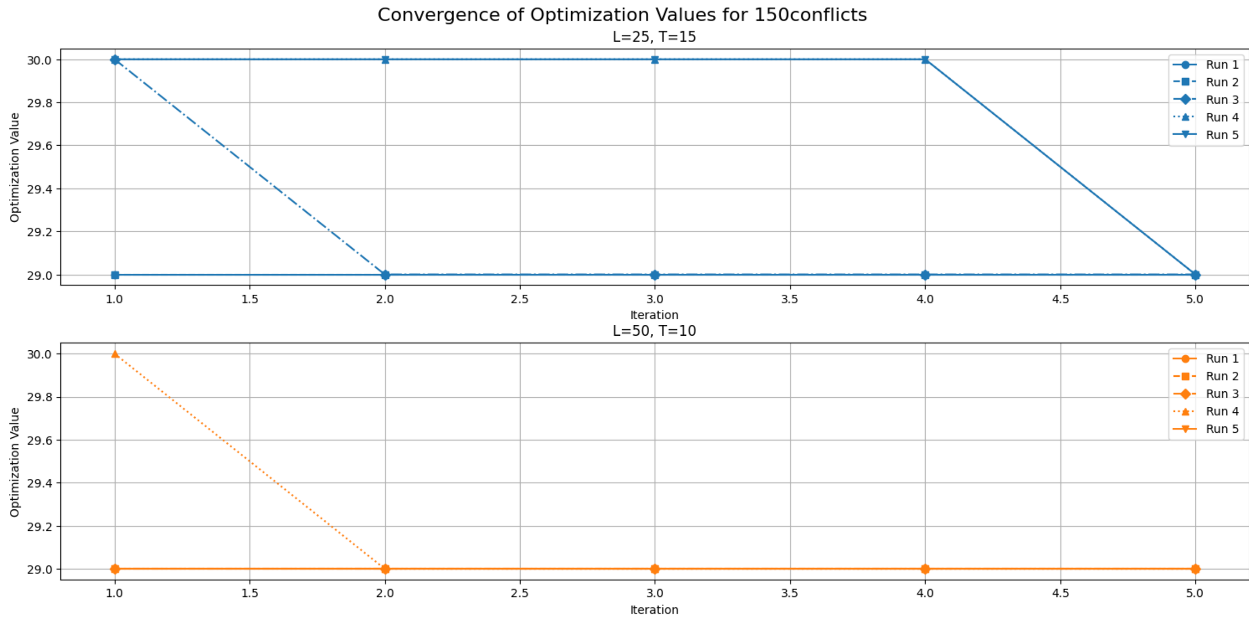

Figure 10: Convergence graph for 20 items with 150 conflicts

We see with 26 conflict, which is a very small amount, the algorithm is struggling more. This is why the time limit is often reached, resulting in the jumps in the graph. $L = 50$ and $T = 10$ seem to be best, reaching 8 and most reaching 9. With 68 conflict the algorithm is experiencing less trouble, producing 9 after 2 iterations for $L = 50$ and $T = 10$. 150 conflicts, means that almost every items is at conflicts with almost every other item, So the optimal solution is going to be higher. Still, $L = 50$ and $T = 10$ reaches it after 2 iterations. $L = 50$ and $T = 10$ is the best combination, meaning half of the number of independent sets is added each iteration with a maximum of half the number of items in each set.

6.1 Theoretical versus computational optimum

To be sure the algorithm produces the true optimum, using the combination of $L = 50$ and $T = 10$ we look at the difference between the solution of the algorithm compared to the theoretical optimum. To get the theoretical optimum from this algorithm, we need to calculate the ISB-IP with every possible set. This can be done by taking the power set of the set of every item and removing the ones that have internal conflicts. The remaining sets is every possible set, thus we get the theoretical optimum.

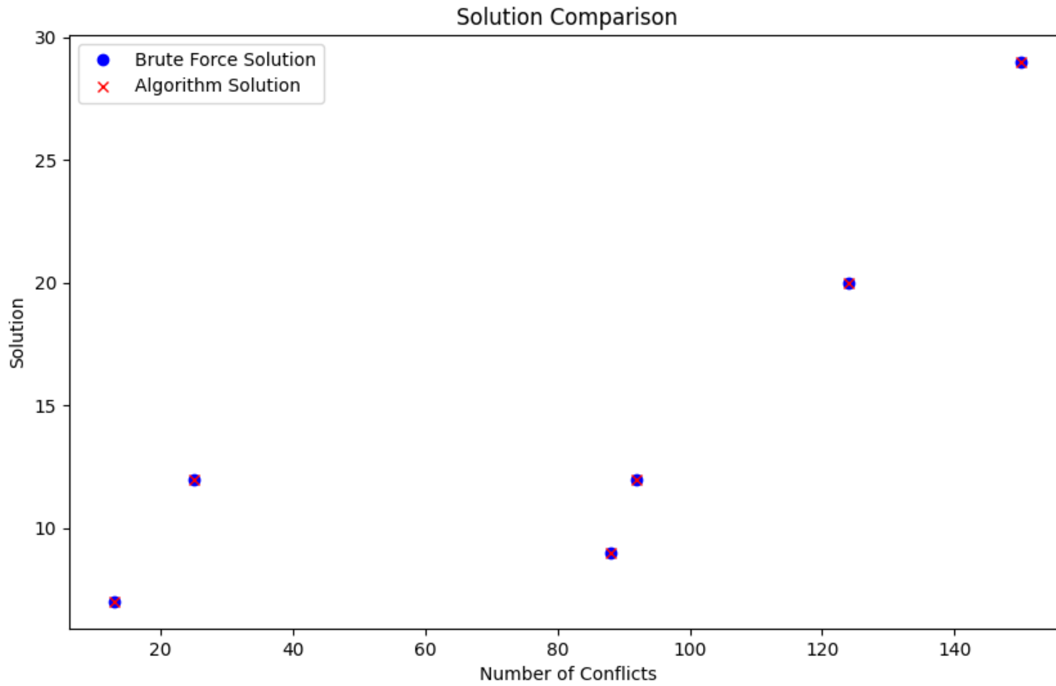

Figure 11: Theoretical and computational solutions

A hand full of different instances have been calculated using the power set without conflicts as input for the integer linear program. This is not possible for every instance, due to the long computation time. As we can see in figure 11, the algorithm has produced the true optimal solution each time it was tested. It appears to be accurate, although testing more instances would make it more rigorous.

6.2 Computation time against number of conflicts

Now we have a good combination for L, T and the number of iterations. We have strong reason to believe the algorithm produces the optimal solution. From now on this will be the standard combination used. This will be important when comparing the computation time of the proposed solution. We will compare the computation time of the algorithm with a different number of conflicts for each entry. In figure 12 the computation time of 45 different instances is plotted against the number of conflicts in the data. We can see that most instances are solved in less than 50 seconds and a big part of those within a few seconds, no matter the number of conflicts. However, for the instances ranging between 25 and 100 conflicts there are some instances that take considerably longer to compute.

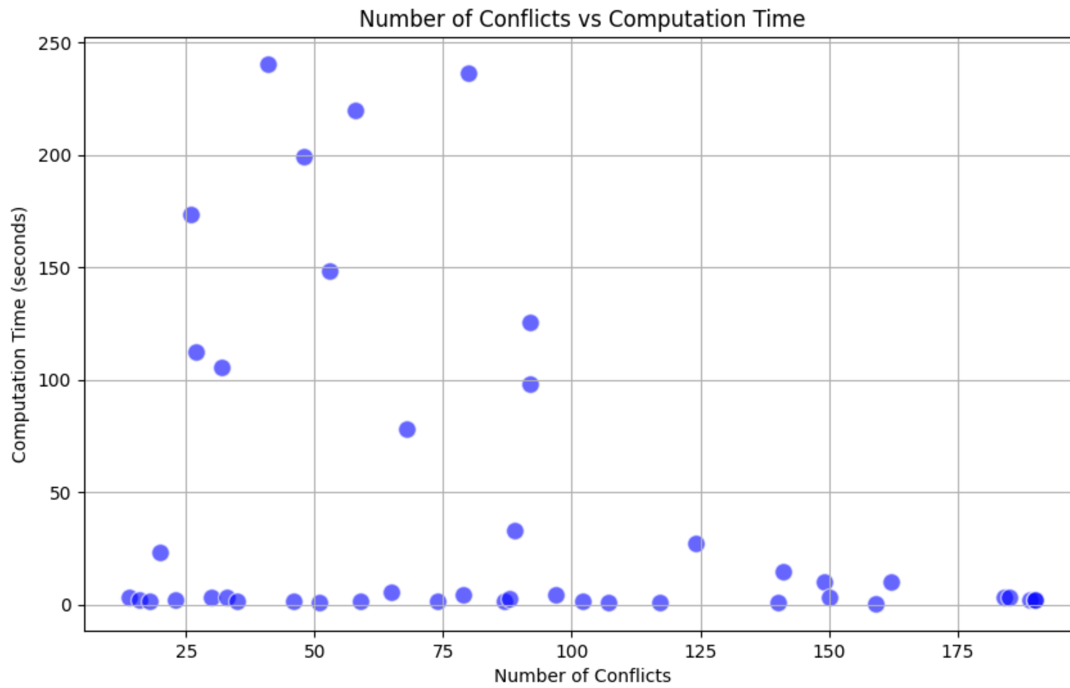

Figure 12: The computation time depending on the number of conflicts

6.3 Number of batteries used depending on the number of conflicts

Figure 13 shows a clear correlation between the number of bins used and the number of conflicts in the problem. Since Figure 12 shows us that there is not a clear correlation between the number of conflicts and the computation time, we can conclude that the computation time doesn't seem to depend on the total number of bins used, but primarily on the number of conflicts.

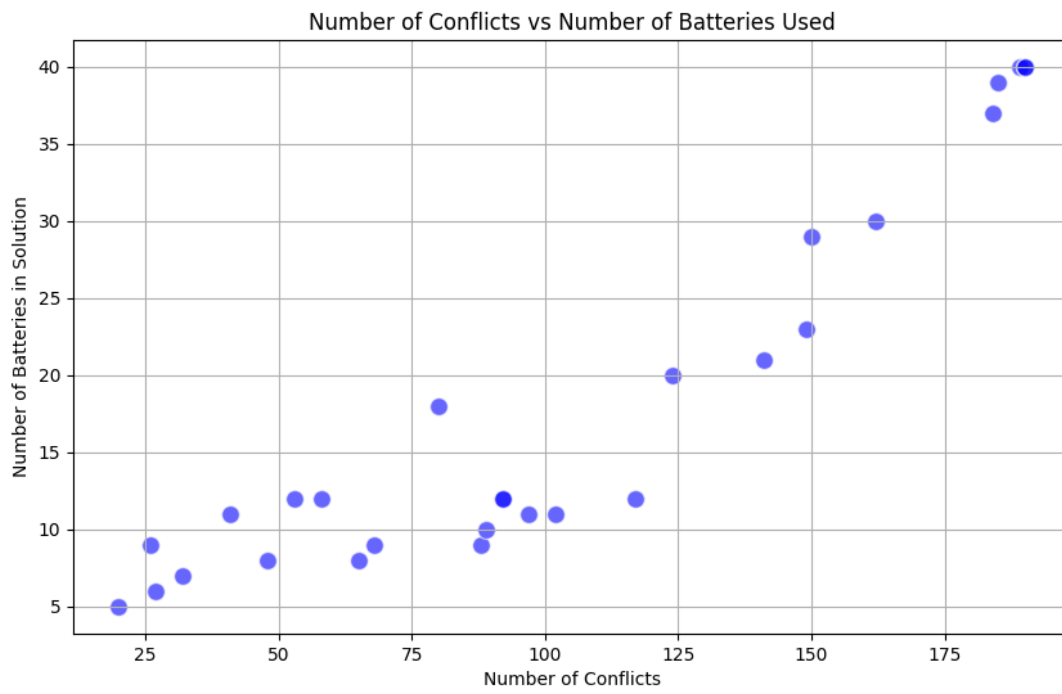

Figure 13: The scheduled number of batteries of each instance, depending on the number of conflicts

7 Conclusion

This thesis proposes a new solution to minimize the number of batteries needed to schedule ships depending on the presented harbour. The proposed solution is an extension to the bin packing problem. Item Fragmentation (BPP-IF) and Conflicts (BPPC) were added to allow ships to carry more batteries and to prevent batteries from being scheduled for multiple ships, respectively. Negative Items (BPP-NI) were added to allow for the batteries to be charged when they were not scheduled to be used. While it was eventually discovered to be an existing extension, this was not known at the time the negative items were added to the problem. The combination of the three bin packing extensions (BPPC-IF-NI) formed the basis for the proposed solution. By a polynomial time reduction we showed that BPPC-IF-NI is NP-hard and therefore cannot be analytically solved in polynomial time. The solution can be approximated with an updated version of the heuristic presented by Ekici (2022).

The main idea in this heuristic approach is to generate a set of independent sets of the conflict graph and determine the packing of the bins using the independent sets with the help of a mathematical model. An iterative framework is used where L sets of T independent sets are generated at each iteration and considered together with the independent sets used in the best solution to obtain a better solution. After testing many possible combinations of L and T, it was found that $L = 50$ and $T = 10$ gives the best results, meaning that half of the number of the initial independent sets (L-value) containing a maximum of half the number of items in each set (T-value) where added in each iteration. When using these parameters, it was found that the solution of the heuristic was the same as the true optimal value. This was checked by inserting the power set without conflicts into the integer linear program.

The performance of the proposed solution was tested on a 45 randomly generated instances, varying in the number of conflicts. It was seen that there is no correlation between the computation time and the number of conflicts. Most instances were solved in a few seconds while other instances could run for a very long time before reaching their time limit. It was also found that there was a strong correlation between the number of conflicts and the number of batteries used in the solution, without any clear outliers.

The proposed solution is able to very quickly approximate the minimum number of batteries needed to schedule the ships for the given harbour. Since most instances are approximated in a couple of seconds, it can also be used to update schedules when complication arise on a smaller scale. This would allow harbours to switch to electric ships, while reducing the cost by minimizing the number of expensive ship batteries.

8 Discussion

This thesis presents a solution to tackle the ship battery scheduling problem, which is still a theoretical problem. The aim is to make the solution better, so it can be used on more varying real life problems. Improvements and recommendations for further research are listed in this chapter.

In the proposed solution, ships could take as many batteries a needed. In real life, this would not be feasible. Although, the finale solution almost always would take 1 or 2 batteries, it would be interesting to limit the number of fragmentation for item i. If the number of fragmentation was added as a parameter, the model would be suitable for ships of varying capacity. There you could limit the number of fragmentation to 1 for example, if the ships you are modelling can only take 2 batteries at most.

Additionally, it would not be feasible to use more batteries while specifying a fixed charge that each battery must supply to the ship. In a real-life scenario, all the batteries would contribute the same amount of energy to a trip, or alternatively, the next battery would only be used when the previous one is empty. However, this issue can easily be addressed by adding an extra constraint to ensure one of these methods is followed for using the batteries.

As of now, it is assumed that the ships arrive and depart exactly on the scheduled time. This would not be realistic and therefore it would be a good addition to the proposed solution to take delays into account. A buffer between scheduled batteries could be added to take care of the smaller delays and the algorithm would probably have to be run again for bigger delays. That is why it is smart to see how to make the proposed solution more suited to these re-computations. One way to deal with this problem could be to check if the updated conflict graph changes the sets used in the former solution. If this is the case, only a few sets could be changed to get rid of the conflicting items, this was however outside the scope of this thesis.

Similarly, the required energy for each trip is based on the scheduled arrival times and departure times. For a more realistic model of the real world problem, this would be a good addition. It would be interesting to see what happens to the solution or the computation time. This way, a trip upstream might demand more energy per time unit, therefore increasing the number of batteries needed in the solution.

In the generated data, it was assumed there was a clear cut-off at 03:00 and would start again at 06:00. This was done to deal with each day as a separate scheduling problem. This way all batteries would be charged the next day. If delays are added different days could get overlapping ships. If this happens, you could consider the delayed ships non-charged and conflicting items for the next day. One good addition that would help this approach, is a new input for the charged capacity at the start of the day.

The goal of this thesis was to minimize the number of batteries used to schedule ships. This was motivated by the fact that batteries are more expensive than the charging stations. It would be interesting for further research to include the number of charging stations in the proposed solution. This could be done by adding another conflict graph, representing the availability of the charging stations. This way the charging stations can be scheduled as well.

All results were derived using instances with 20 ships. This was done to limit the computation time and due to the limited time available to generate results. For more rigorous results, a larger variation of instance should be looked at. The chosen combination of L and T should be reconsidered, although they were chosen to be proportional to the input values. It would therefore be interesting to see if these parameters are also the best for other instances.

Due to the limit time, the relation between computation time and number of conflicts was not further researched. This would the most interesting further research, since this way it could be predicted when the computation time would exceed its limit. It could not be concluded from the limited results generated.

9 Appendix

Listing 1: Algorithm 1

```
def algorithm_1(V, w, W, G):
    P_I = [] # Initialize set of independent sets
    V = set(V) # {0,1,2}
    num_vertices = len(G) # 3
    for i in V:
        S = {i} # Initialize independent set # {0}
        neighbors_i = {j for j in range(num_vertices) if G[i][j] == 1} # 0, empty set
        V_prime = V - {i} - neighbors_i # Remove neighbors of i from V # V prime = {1,2}
        while V_prime:
            d_prime = {j: sum(G[j][k] for k in V_prime) for j in V_prime} # Compute degrees
            i_star = min(V_prime, key=lambda l: d_prime[l]) # Find i* satisfying min degree
            S.add(i_star) # Add i* to independent set {0,1}
            V_prime -= {i_star} # Remove i* from V' # vprime = {2}
            neighbors_i_star = {j for j in V_prime if G[i_star][j] == 1}
            V_prime -= neighbors_i_star # Remove neighbors of i* from V'
        total_weight = sum(w[j] for j in S)
        copies = math.ceil(total_weight / W) # round up number of copies of independent set S
        P_I.extend([list(S.copy())] * copies) # Add copies of S to P_I

    return P_I
```

Listing 2: Algorithm 2

```
def algorithm_2(V, w, W, G, L, T):
    P_R = []
    num_vertices = len(V)

    is_adj_matrix = isinstance(G, np.ndarray)
    while len(P_R) < L:
        T_prime = random.randint(2, T)
        S = set()
        V_prime = set(V)

        while len(S) < T_prime and V_prime:
            i = random.choice(list(V_prime))
            S.add(i)
            V_prime.remove(i)

            if is_adj_matrix:
                # Adjacency matrix case
                neighbors_i = {j for j in V_prime if G[i][j] == 1}
            else:
                # Dictionary of edges case
                neighbors_i = {j for j in V_prime if (i, j) in G or (j, i) in G}

            V_prime -= neighbors_i
        total_weight = sum(w[j] for j in S)
        copies = math.ceil(total_weight / W) # round up number of copies of independent set S
        P_R.extend([list(S.copy())] * copies) # Add copies of S to P_R
    return P_R
```

Listing 3: Algorithm 3

```
def algorithm_3(V, w, W, G, t_b, t_e, L, T, IterationLimit):
```

```
    P_set = algorithm_1(V, w, W, G) # gives a list containing independent lists.
    z_best = math.inf
    P_B = []
    IterationCounter = 0

    while IterationCounter < IterationLimit:
        IterationCounter += 1
        P_R = algorithm_2(V, w, W, G, L, T) # list containing random independent lists
        P_set = P_set + P_R
        P = len(P_set)
        opt = ILP_solution(V, w, W, P_set, t_b, t_e)

        if opt.fun is not None:
            if opt.fun < z_best: # checks if the optimal solution found is smaller than the current best
                z_best = opt.fun
                P_B = sets_used(P_set, len(V), len(P_set), opt) # P_B is a list of all the lists that are

        P_set = P_B

    return opt, P
```

Listing 4: ILP Solution for Independent Set Constraints

```
def ILP_solution(V, w, W, P_sets, t_b, t_e):
    n = len(V) # Number of items
    # Define the number of sets P
    P = len(P_sets)
    # Objective function coefficients (minimize sum of x_p). We have a list of
    c = np.zeros(3 * n * P + (n - 1) * P + P)
    c[3 * n * P + (n - 1) * P:] = 1

    # Inequality Constraints
    lhs_ineq = [] # Inequality constraint coefficients
    rhs_ineq = [] # Inequality constraint bounds

    # Equality Constraints
    lhs_eq = [] # Equality constraint coefficients
    rhs_eq = [] # Equality constraint bounds

    # list of variable [f_ip, o_ip , o_ijp^*, g_ip , x_p   ]
    # length per variable [0:np-1 , np:2np-1, 2np:3np-1, 3np:4np-1, 4np:4np+p]

    # Constraint 1: sum(f_ip - o_ip - o_ijp^*) - W * x_p <= 0 for all p
    for p in range(P):
        constraint = np.zeros(3 * n * P + (n - 1) * P + P) # create a new constraint
        for i in range(n):
            if V[i] in P_sets[p]:
                constraint[i * P + p] = 1 # f_ip
                constraint[n * P + i * P + p] = -1 # o_ip
                constraint[3 * n * P + (n - 1) * P + p] = -W # x_p
        for i in range(n - 1):
            if are_items_in_same_list(P_sets[p], V[i], V[i + 1]):
                constraint[2 * n * P + i * P + p] = -1 # o_ijp^*
        lhs_ineq.append(constraint)
        rhs_ineq.append(0)

    # Constraint 2: sum(f_ip) = w_i
    for i in range(n):
        constraint = np.zeros(3 * n * P + (n - 1) * P + P)
```

```
        for p in range(P):
            if V[i] in P_sets[p]:
                constraint[i * P + p] = 1 # f_ip
        lhs_eq.append(constraint)
        rhs_eq.append(w[i])

    # Constraint 3: f_ip + w_i * g_ip <= w_i * x_p # this is changed!!!!!!!!!
    for p in range(P):
        for i in range(n):
            if V[i] in P_sets[p]:
                constraint = np.zeros(3 * n * P + (n - 1) * P + P)
                constraint[i * P + p] = 1 # f_ip = 1
                constraint[2 * n * P + (n - 1) * P + i * P + p] = w[i] # g_ip = w_i
                # constraint[3 * n * P + (n - 1) * P + p] = -w[i] # x_p
                lhs_ineq.append(constraint)
                rhs_ineq.append(w[i]) # leq w_i

    # Constraint 4: o_ip - W * g_ip <= 0
    for p in range(P):
        for i in range(n):
            if V[i] in P_sets[p]:
                constraint = np.zeros(3 * n * P + (n - 1) * P + P)
                constraint[n * P + i * P + p] = 1 # o_ip = 1
                constraint[2 * n * P + (n - 1) * P + i * P + p] = -W # g_ip = -W
                lhs_ineq.append(constraint)
                rhs_ineq.append(0) # leq 0

    # Constraint 5: sum(-f_ip + o_ip + o_ijp^*) <= 0 for t^e_i <= t_k^b
    t_b.append(np.inf)              # add inf to also add the last end time to the constraint
    extra_item = max(V)+1           # make an item that is one bigger than the previous last item
    for p in range(P):
        P_sets[p].append(extra_item) # add one item at the end
        for k in range(n):
            if V[k] in P_sets[p]:
                constraint = np.zeros(3 * n * P + (n - 1) * P + P)
                for i in range(n-1):
                    if V[i] in P_sets[p]:
                        if t_e[i] <= t_b[k]:
                            constraint[i * P + p] = -1 # f_ip = -1
                            constraint[n * P + i * P + p] = 1 # o_ip = 1
                for i in range(n - 2):
                    if V[i] in P_sets[p] and V[i + 1] in P_sets[p]:
                        if t_e[i] <= t_b[k]:
                            constraint[2 * n * P + i * P + p] = 1 # o_ijp = 1
                lhs_ineq.append(constraint)
                rhs_ineq.append(0)
        del P_sets[p][-1] # remove the added item
    del t_b[-1] # remove the extra item again

    # Constraint 6: sum(f_ip - o_ip - o_ijp^*) <= W * x_p for t^e_i <= t_k^b
    t_b.append(np.inf)              # add inf to also add the last end time to the constraint
    extra_item = max(V) + 1         # make an item that is one bigger than the previous last item
    for p in range(P):
        P_sets[p].append(extra_item) # add one item at the end
        for k in range(n):
            if V[k] in P_sets[p]:
                constraint = np.zeros(3 * n * P + (n - 1) * P + P)
                for i in range(n - 1):
```

```
                if V[i] in P_sets[p]:
                    if t_e[i] <= t_b[k]:
                        constraint[i * P + p] = 1                 # f_ip = 1
                        constraint[n * P + i * P + p] = -1        # o_ip = -1
                        constraint[3 * n * P + (n - 1) * P + p] = -W # x_p = -W
            for i in range(n - 2):
                if V[i] in P_sets[p] and V[i + 1] in P_sets[p]:
                    if t_e[i] <= t_b[k]:
                        constraint[2 * n * P + i * P + p] = -1    # o_ijp = -1
            lhs_ineq.append(constraint)
            rhs_ineq.append(0)
    del P_sets[p][-1] # remove the added item
del t_b[-1]                                                   # remove the extra item again

# Constraint 7: sum(o_ip + o_ijp^* + g_ip <= x_p * inf
# To force that all variable are 0 if bin p is not chosen
for p in range(P):
    constraint = np.zeros(3 * n * P + (n - 1) * P + P) # create a new constraint
    for i in range(n):
        if V[i] in P_sets[p]:
            constraint[n * P + i * P + p] = 1 # o_ip
            constraint[2 * n * P + (n - 1) * P + i * P + p] = 1 # g_ip = 1
            constraint[3 * n * P + (n - 1) * P + p] = -100000 # x_p
    for i in range(n - 1):
        if are_items_in_same_list(P_sets[p], V[i], V[i + 1]):
            constraint[2 * n * P + i * P + p] = 1 # o_ijp^*
    lhs_ineq.append(constraint)
    rhs_ineq.append(0)

# Add variable bounds

o_star_up_bnd = [] # create bounds for o_ijp
for i in range(n - 1):
    for p in range(P):
        if are_items_in_same_list(P_sets[p], V[i], V[i + 1]):
            o_star_up_bnd.append(min(2, (t_b[i + 1] - t_e[i])))
        if not are_items_in_same_list(P_sets[p], V[i], V[i + 1]):
            o_star_up_bnd.append(0)

bounds = []
for i in range(2 * n * P):
    bounds.append((0, None)) # f_ip, o_ip
for i in range((n - 1) * P):
    bounds.append((0, 60 * o_star_up_bnd[i])) # o_ijp and change hours to minutes
for i in range(n * P + P):
    bounds.append((0, 1)) # g_ip and x_p

# define decision variable (0 = continuous, 1 = integer)
integrality = np.zeros(3 * n * P + (n - 1) * P + P)
for i in range(n):
    for p in range(P):
        integrality[p + (i * P)] = 0 # f_ip
        integrality[n * P + p + (i * P)] = 0 # o_ip
        integrality[2 * n * P + (n - 1) * P + p + (i * P)] = 1 # g_ip
        integrality[3 * n * P + (n - 1) * P + p] = 1 # x_p
for i in range(n - 1):
    for p in range(P):
        integrality[2 * n * P + p + (i * P)] = 0 # o_ijp
```

```
    # Solve the problem
    result = linprog(c=c, A_ub=lhs_ineq, b_ub=rhs_ineq, A_eq=lhs_eq, b_eq=rhs_eq, bounds=bounds,
                  integrality=integrality)
    # , options={'time_limit': time_limit}
    return result
```

Listing 5: Ship Data Generation

```
def generate_ship_data(amount_ships, mean_energy_1, mean_energy_2, sd_energy_1, sd_energy_2):

    # Departure Time Generator
    begin_times = np.sort(np.random.uniform(12, 21, amount_ships))

    # Rewrite hours and minutes
    begin_hours = np.floor(begin_times).astype(int) % 24
    begin_minutes = np.round((begin_times % 1) * 60).astype(int)

    begin_times = begin_hours + begin_minutes / 100

    # Energy Demand Generator
    required_energy_1 = np.random.normal(mean_energy_1, sd_energy_1, amount_ships // 2)
    required_energy_2 = np.random.normal(mean_energy_2, sd_energy_2, amount_ships - amount_ships // 2)
    required_energy = np.maximum(np.round(np.concatenate([required_energy_1, required_energy_2])), 20)

    # Calculate end times based on required energy
    begin_hours = np.round(begin_times, 0) * 60 # change the hours to the amount of minutes
    begin_minutes = begin_hours + 100 * (begin_times - np.floor(begin_times)) # add hours and minutes
    extra_minutes = required_energy
    end_minutes = begin_minutes + extra_minutes
    end_times = np.round(((end_minutes // 60) % 24 + (end_minutes % 60) / 100), 2)

    # Create the generated data
    generated_data = pd.DataFrame({'begin_times': begin_times, 'end_times': end_times,
                             'required_energy': required_energy})

    # Show the DataFrame
    return generated_data
```

Listing 6: Conflict Graph Creation

```
def create_conflict_graph(generated_data):

    # Function to check if two intervals overlap
    def is_overlap(interval1, interval2):
        return not (interval1[1] <= interval2[0] or interval2[1] <= interval1[0])

    # Create the conflict graph
    G = nx.Graph()
    # conflict counter
    counter = 0

    # Add nodes for each ship
    for i in range(len(generated_data['begin_times'])):
        G.add_node(i)

    # Add edges for overlapping intervals
    for i in range(len(generated_data['begin_times'])):
        for j in range(i + 1, len(generated_data['begin_times'])):
            if is_overlap((generated_data['begin_times'][i], generated_data['end_times'][i]),
```

```
                (generated_data['begin_times'][j], generated_data['end_times'][j])):
            G.add_edge(i, j)
            counter += 1

    adj_matrix = nx.to_numpy_array(G)
    return adj_matrix, counter
```

References

L. Bertazzi, B. Golden, and X. Wang. The bin packing problem with item fragmentation: a worst-case analysis. *Discrete Applied Mathematics*, 261:63–77, 2019.

Ali Ekici. Bin packing problem with conflicts and item fragmentation. *Computers Operations Research*, 126:105113, 2021. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2020.105113. URL https://www.sciencedirect.com/science/article/pii/S0305054820302306.

Ali Ekici. Variable-sized bin packing problem with conflicts and item fragmentation. *Computers Industrial Engineering*, 163:107844, 2022. ISSN 0360-8352. doi: https://doi.org/10.1016/j.cie.2021.107844. URL https://www.sciencedirect.com/science/article/pii/S0360835221007488.

J. Faber, S. Hanayama, et al. *Fourth IMO GHG Study*. IMO, 2020.

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman, 1979.

K. Jansen. An approximation scheme for bin packing with conflicts. *Journal of Combinatorial Optimization*, 3(4): 363–377, 1999.

Port of Rotterdam. Up-to-date information: Arrival and departures, 2024. URL https://www.portofrotterdam.com/en/up-to-date-information/arrival-and-departures. Accessed: 2024-07-23.

Vijay V. Vazirani. *Approximation Algorithms*, volume 1. Springer, 2001.

Zero Emission Services. Zero emission services homepage. https://zeroemissionservices.nl/en/homepage/. Accessed: 2024-07-22.