An Iterated Local Search Approach to the Container Pre-Marshalling Problem

John Farrelly
Computer Science Dept.
Munster Technological University
Cork, Ireland
j.farrelly@mycit.ie

Dr. Diarmuid Grimes
Computer Science Dept.
Munster Technological University
Cork, Ireland
diarmuid.grimes@mtu.ie

***Abstract*—Port congestion can be a major bottleneck in logistics and supply chain optimization. One key metric used to determine a port's efficiency is how quickly container ships are loaded. As containers may arrive at the port up to one week before their scheduled departure, containers that need to leave port first can often be buried underneath containers with later departure dates. This can lead to lengthy delays in loading a ship, where containers that are blocking access to containers due to leave port must be moved out of the way first. To alleviate this problem, ports sort the containers in a yard by their departure date before a ship arrives, leading to speedier loading. This sorting of containers in the yard is called pre-marshalling, with the goal of sorting the containers in the bay in as few crane moves as possible. We propose a novel iterated local search approach to solve the pre-marshalling problem, and demonstrate empirically the benefits of the approach on a well-studied benchmark.**

***Index Terms*—Iterated Local Search, Logistics, Container Pre-Marshalling**

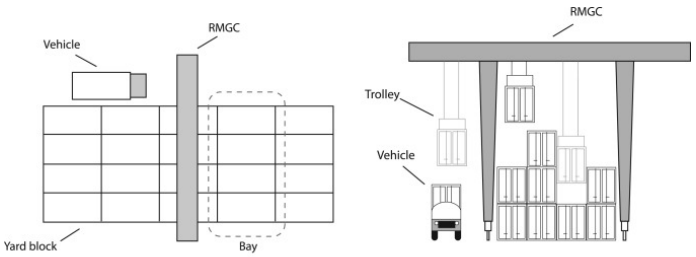
I. Introduction

Much of the world's international cargo is transported using shipping containers. With the standardisation of container sizes in the 1950s, they became inter-modal, i.e. they can easily be transferred between different types of transportation such as trains, trucks and ships. In 2019, global container throughput reached approximately 802 million twenty-foot equivalent units[1]. The increase in container transportation has resulted in more congestion at the ports. This bottleneck can cause delays in loading container ships, at significant expense for shipping companies.

A further incentive for loading/unloading efficiency for a port operator is that ports are in competition with each other. A key metric used to compare ports is the berthing times of ships (how long they are in port). Much of the berthing time of a ship consists of the time it takes to unload existing containers and load new containers destined for other ports.

The issue of port congestion has garnered considerable attention in recent years, with numerous works in the literature studying different aspects that contribute to the problem [1]–[4]. In this work, we are interested in the redistribution of containers in a yard, in order to minimise delays to loading.

A typical port configuration involves storing blocks of containers in the port yard, divided into bays (rows) of containers stacked on top of each other. Containers arrive at the storage bays by truck, and a rail mounted gantry crane (*RMGC*) lifts the container from the truck into the bay, as shown in Fig 1.



(a) Top view of an RMGC. (b) Front view of an RMGC.

Fig. 1. A rail-mounted gantry crane over a yard block [5]

Due to differing pre-departure arrival times, container bays are frequently not sorted in their loading order; containers with higher priority groups can be below ones with a lower priority group. This means that the RMGC would need to move lower priority containers out of the way to get at higher priority ones so that they can be loaded in order. These "re-handles" are slow and can cause significant delays in loading a ship.

To avoid performing these expensive re-handles while a ship is berthed, ports perform "pre-marshalling" to re-sort the bays into their priority order before the ship arrives. This is the container pre-marshalling problem (CPMP) [6]. The goal is to sort the stacks in a bay in as few moves of the RMGC as possible. More formally, given n containers, with associated priorities (one of g priority groups) and initial stack location in one of S stacks, redistribute stacks such that all stacks are in non-decreasing priority order subject to a restriction on the maximum number of tiers T allowed in a stack.

Based on analysis of the behaviour of existing algorithms, we propose a dedicated iterated local search approach to solve the CPMP. It combines a greedy hill climbing approach with a stack clearing perturbation step to escape local minima. Our empirical evaluation demonstrates the ability of the approach

[1]https://www.statista.com/statistics/913398/container-throughput-worldwide/

978-1-6654-5227-4/22/$31.00 ©2022 IEEE

2022 33rd Irish Signals and Systems Conference (ISSC) | 978-1-6654-5227-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISSC55427.2022.9826140

to find a good tradeoff between solution quality and computational effort (averaging less than a second) and its scalability for large instances.

II. Background

A. Container Pre-Marshalling Problem

The container pre-marshalling problem involves moving one container at a time until there is no *mis-overlaid* container in the bay. A mis-overlaid container is one which is higher in a stack than a container with a higher loading priority. Figure 2 shows a sample problem, three containers are mis-overlaid (2, 4 and 5), requiring 3 moves to sort the bay. In general, given an initial bay configuration, the goal is to find the minimum number of container moves to sort all the containers in a bay, such that there are no mis-overlaid containers.

When moving containers, the following must be considered:

- A yard block has a maximum number of containers that can be placed in a stack. This is due to the RMGC generally having a static height, and stack stability concerns. The number of "tiers" in a bay is used to denote the maximum number of containers that can be in any stack.
- In the CPMP, containers can not be moved between bays in the yard block. Due to health and safety restrictions in most ports, cranes are forbidden from moving containers from one bay to another.
- Priority group 1 is the highest priority; these containers must be loaded in the ship first, followed by priority group 2, 3 etc.

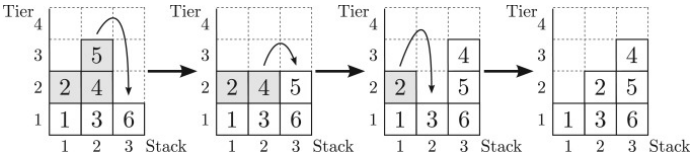


Fig. 2. Sample pre-marshalling problem with solution moves, with container priority as indicated in each cell and mis-overlaid containers in grey [7]

There are also a number of other assumptions for the CPMP, which are typically used in the literature for this problem [8]:

- All containers in a bay are the same dimensions.
- The initial configuration of the bay is known.
- Only containers at the top of a stack can be moved.
- Containers can be placed on top of each other, or on the ground.
- The loading priority groups of the containers are known and multiple containers can share the same loading priority group.

1) Definitions and Notation: Let L be a matrix that represents the current layout of a loading bay. It has T rows, the maximum number of tiers in the bay, and S columns, the number of stacks in the bay. $L(t, s)$ represents a location in the bay at tier t and stack s, and contains an integer that represents the priority group of the container located at that position. $L(t, s) = 0$ means that there is no container at that location. All tiers above an empty tier must also be empty; if $L(t, s) = 0$, then $L(t', s) = 0$ must hold for all $t' = t + 1, \ldots, T$. G is the largest priority group of any container in L.

A move m is defined as taking the top container from a source stack s_{src} and moving it to the lowest free location in a destination stack s_{dest}, where the source and destination stacks are different: $1 \leq (s_{src}, s_{dest}) \leq S, s_{src} \neq s_{dest}$. The source stack must have a least one container in it, $L(0, s_{src}) > 0$, and the destination stack must have room for at least one container, $L(T, s_{dest}) = 0$. A sequence of moves $(m_1, \ldots, m_p), p \geq 1$ is valid if move m_1 is valid for layout L producing L', and move m_2 is valid for layout L' etc.

A valid solution is such that there are no mis-overlaid containers in the bay. That is, $L(t, s) \geq L(t + 1, s)$ for all $1 \leq t < T$, $1 \leq s \leq S$. The objective is to find a valid solution in the minimum number of moves from a given initial configuration of the loading bay.

B. Related Work

There are a number of tree search approaches to solve the pre-marshalling problem. Bortfeldt and Forster (2012) [9] use a heuristic tree search algorithm, and define a formula for calculating a lower bound number of moves necessary to sort the bay. The work in Tierney et al. (2017) [5] uses the lower bound heuristic from [9] as the cost estimation heuristic to an A* and IDA* implementation. They also add branching rules which prevent transitive moves and domain specific symmetry breaking rules to reduce the number of nodes explored. Hottung et al. (2020) [10] employs a deep learning heuristic tree search algorithm: a deep neural network is trained via supervised learning on optimal and near-optimal solutions to CPMP problem instances and is used to determine which branches to search and which to prune in their tree search procedure.

Some solutions focus on exact method approaches. In Lee and Hsu (2007) [6], an integer programming model is created which uses an embedded multi-commodity network flow representation. Caserta and Voß (2009) [11] introduce a "corridor method" in conjunction with an exact method to generate solutions. Exact methods have difficulty with the problem because of the size of the search space, so [11] introduces the concept of "corridors"; slices of the bay configuration which are small enough to run an exact method on. An exact method is run on the corridor to find what would be the best container to move within that corridor. A new corridor is selected, and the process is repeated until solved.

Wang et al. (2015) [12] introduced a greedy constructive heuristic, referred to as the target-guided heuristic (TGH). This selects the container with the lowest priority that is mis-overlaid, selects a set of candidate destination stacks using various criteria, and moves the container to the candidate stack moving any containers from that stack out of the way if required. They also proposed embedding this approach within beam search (BS-B), where a re-evaluation of the best container and destination stack is done after every individual container move, meaning it may find more promising moves

while in the middle of a different container relocation. This produces much better solutions, albeit at up to orders of magnitude increase in runtime. Wang et al. (2017) proposed an improvement to the target based heuristic in [13] which they refer to as a feasibility based heuristic (FBH), using concepts around state feasibility, container stability and dead-end avoidance.

Some of the best results to date have been achieved by the biased random key genetic algorithm (BRKGA) proposed by Hottung and Tierney (2016) [7]. Each gene in a chromosome is a variable which is fed into different parts of a heuristic solution construction algorithm, which is the "decoder" of the chromosome. Each successive generation optimises the parameter values encoded in the chromosomes to guide the decoder to produce better solutions to the specific problem instance that is being solved. This outperforms the constructive heuristic approaches and beam search in terms of solution quality and much faster than the latter, albeit still taking over one minute for larger instances.

More recently, Hottung et al. (2020) [10] found more improvements than their BRKGA through a proposed deep learning approach to the CPMP. A Deep Learning Heuristic Tree search algorithm was developed: a deep neural network is trained via supervised learning on optimal and near-optimal solutions to CPMP problem instances and is used to determine which branches to search and which to prune in their tree search procedure. The search strategy algorithm does not contain any CPMP specific heuristics; the training teaches the DNN the solution strategies to solve CPMP instances. This approach yields the best heuristic solutions to the CPMP to date with gaps-to-optimality below 2%. Their results, for the benchmarks they give, outperformed all other approaches.

III. Iterated Local Search for the CPMP

Iterated Local Search (ILS) attempts to alleviate the issue of standard local search becoming trapped in a local optimum. It involves a sequence of alternating local search and perturbation phases, with the goal of iteratively finding better local optima with each iteration. The goal of the local search phase is to find a local optimum as efficiently as possible, and the perturbation phase is used to escape from the local optimum. This sequence is then repeated until a termination condition has been met, which is generally either when a solution of acceptable quality has been found or a certain number of iterations have been performed.

Algorithm 1 shows the structure of our ILS for the CPMP. A difference between our approach and traditional ILS is that there is no initial solution generation phase; we always start with the initial bay configuration L. For our algorithm, a lower bound is used as the cost function (described subsequently) to be minimised in each iteration. The algorithm will return a solution, if found, in the form of the sequence of moves m_{seq} that will sort the bay layout L

Our acceptance criterion is if the lower bound value of the newly generated L' layout is not more than 1 worse than the lower bound value of L. We do this so that moves that are considered losing moves will be accepted. As noted in the following section, there is no guarantee that all better bay layouts have a lower lower-bound value, which is why we allow some leeway in our acceptance criterion.

Algorithm 1 CPMP ILS algorithm

$m_{seq}, L^* \leftarrow$ Local-Search(L)
while Has-Mis-Overlaid-Containers(L^*) **do**
  $m^1_{seq}, L' \leftarrow$ Perturbation(L^*)
  $m^2_{seq}, L' \leftarrow$ Local-Search(L')
  if Lower-Bound(L') $\leq$ Lower-Bound(L^*) $+1$ **then**
    $L^* \leftarrow L'$
    $m_{seq} \leftarrow m_{seq} + m^1_{seq} + m^2_{seq}$
  end if
end while
return m_{seq}

A. Lower Bound as Cost Function

We use the lower bound from [9] as the cost function for our Iterated Local Search. As the lower bound gives an indication of the minimum number of moves left to sort a bay layout, our goal will be to reduce this value with each move that is made in our algorithms. One drawback of this approach is that there is no guarantee that the lower bound function will return a lower value for a move which does indeed bring the bay closer to being sorted. Likewise, there is no guarantee that it will return a higher value for a move which makes the bay more difficult to sort.

B. Local Search

For the local search element of our ILS (outlined in Algorithm 2), we implement a greedy hill climbing search algorithm, with the goal of minimising our cost function. Our search tree is made up of vertices that represent bay layouts, where an edge between vertices represents the container move that converts one bay layout to another.

Starting with bay layout L, we generate all valid n moves possible from that layout, and generate $n * L'$ layouts. We iterate over all $n * L'$ layouts generated to find the one that yields the lowest cost. If the cost of L' is lower than the cost of L, we repeat the process, substituting L' for L. On the other hand, when the cost is equal or higher, it means we have reached a local optimum and will stop the search. A dedicated tie-breaking strategy (explained in the next section) is employed if multiple L' layouts have the same lower cost. Finally, a cost of 0 is an additional stopping criterion as we have found a layout with no mis-overlaid containers.

The configuration of our neighbourhood uses an "iterative deepening" approach. We first look at all bay layouts that are exactly one move away from L. If we don't find any neighbours with a lower cost, we look at bay layouts that are two moves away from L.

Algorithm 2 CPMP Local Search algorithm

$isLocalOptimum \leftarrow false$
$L' \leftarrow L$
$cost =$ Lower-Bound(L)
$m_{seq} \leftarrow []$
$nMoves_{min} \leftarrow 1$
$nMoves_{max} \leftarrow 2$
$nMoves \leftarrow nMoves_{min}$
while not $isLocalOptimum$ **&** $cost > 0$ **do**
  $cost', m'_{seq}, L' \leftarrow$ Find-Best-Neighbour($L', nMoves$)
  if $cost' < cost$ **then**
    $m_{seq} \leftarrow m_{seq} + m'_{seq}$
    $cost \leftarrow cost'$
  else if $nMoves < nMoves_{max}$ **then**
    $nMoves \leftarrow nMoves + 1$
  else
    $isLocalOptimum \leftarrow true$
  end if
end while
return m_{seq}, L'

1) Tie-breaking Layouts with the Same Cost: When performing the local search, many neighbours may end up having the same cost. Our cost function is only an approximation via the lower bound, so multiple neighbours having the same cost does not mean they will all equally be as good. We experiment with the following two different tie-breaking strategies, as well as random, in an effort to select the "best" neighbour:

- **Highest_Value_Container**: Select the neighbour which moves the container with the lowest priority.
- **Smallest_Container_Difference**: Select the neighbour which moves a container on top of another container that is closest in priority.

C. Perturbations

The perturbation step in ILS is used to escape local optima reached by the Local Search. For the CPMP, our perturbation steps involve generating a number of moves for the bay layout, either randomly or heuristically, and applying those moves. This new layout is then fed into the Local Search to find new, and hopefully better, neighbours.

Our perturbations focus on selecting a stack and clearing it by moving all its containers to other stacks. The reasoning is that containers with the lowest priority must be placed at the bottom of stacks or on top of containers of the same priority group, so it is frequently necessary to clear a stack such that these lowest priority containers can be well placed. Clearing stacks is a fundamental part of the approach in [7], which returns excellent solutions for CPMP problem instances.

The combination of greediness and randomness, like in many metaheuristic approaches, is a key element of our approach. While [9] and [14] discuss that greediness is a necessary part in any CPMP solution technique, it has been shown in [15] that performance of existing greedy heuristic algorithms is generally improved by adding a certain level of randomisation and that some bay configurations cannot be solved by purely greedy algorithms.

1) Stack Selection Strategies: There are different ways in which we can select a stack to be cleared:

- **Random**: A stack is selected at random.
- **Lowest_Stack**: The stack with the fewest containers in it is selected. If multiple stacks have the same number of containers, one is chosen at random. Note that empty stacks are included in the set of eligible stacks to select from in this perturbation.
- **Lowest_Mis-overlaid_Stack**: We first find the set of stacks where at least one container is mis-overlaid and then select the lowest stack from this set.

2) Clearing Stack Strategies: When clearing a stack of containers, a decision must be taken for each container as to where it should be moved to. We tested two strategies:

- **Random**: Containers are moved to any other random non-full stack.
- **Clear_To_Smallest_Difference**: Containers are moved to stacks where they will not be mis-overlaid, and that there is the smallest difference in priority between the container being moved and the container it will be placed above. The hypothesis is that containers cleared in this way might not need to be moved again if they are well placed as part of the clearing.

IV. Evaluation & Results

A. Experimental Setup

The implementation for the proposed iterated local search was performed using Java 8 [2]. All experiments were run on a Lenovo Thinkpad X1 2018 with 8GB of Memory and an Intel 2.3 GHz Quad-Core i5 CPU using Ubuntu 20.04 LTS. Each experiment was executed using a single thread, and allowed a maximum of 1GB of memory.

We use the BF dataset for our experiments [9]. There are 32 categories of problems in the dataset, with 20 problem instances in each category, so 640 instances in total. The difficulty of a CPMP problem instance is mostly dependent on five factors [9]:

- The number of stacks in the bay layout
- The maximum height of the stacks
- The number of different priority groups of the containers
- The percentage of locations occupied by containers in the bay layout
- The percentage of mis-overlaid containers in the initial bay layout.

In general, an increase in the value of a factor is expected to result in a more difficult instance. Each BF category increases one of these factors while the others remain the same.

The proposed ILS algorithm was run 50 times on each problem instance. The results reported in the following include the average solution found per run *ILS(Avg)*, the average of the best solution found across runs *ILS(Best)*, and the overall average runtime in seconds.

[2]https://github.com/johnnyleitrim/cpmp-ls-ga

	Lower bound	TGH	FBH	BS-B	BRKGA		ILS	
					Avg	Best	Avg	Best
Mean	57.09	72.75	68.44	60.66	60.78	60.06	66.87	63.49
Time (s)		<0.1	<0.1	15.3	97		0.39	

TABLE I
COMPARISON TO STATE-OF-THE-ART ON BF DATASET

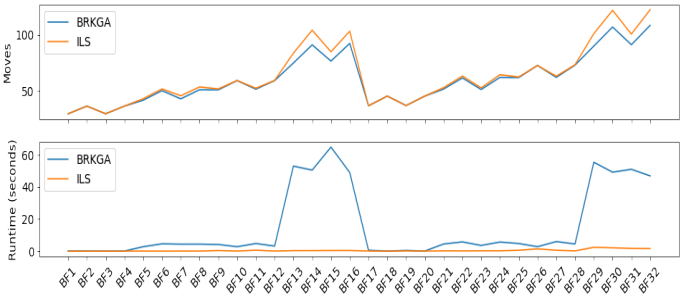


Fig. 3. Solution quality and runtime comparison with BRKGA per instance

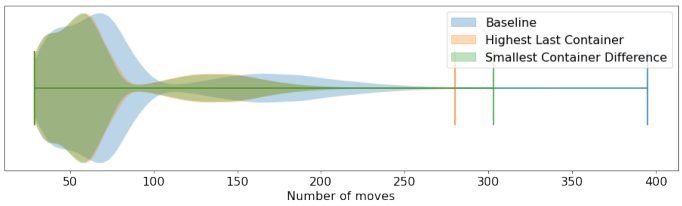


Fig. 4. Comparison of Neighbour Tie-breaking strategies

B. Comparison to the state-of-the-art

In initial experiments, the following configuration was found to perform best:

- Best Neighbour Tie Breaking Strategy: **Smallest_Container_Difference**
- Stack Selection Strategy: **Lowest_Stack**
- Clear Stack Strategy: **Clear_To_Smallest_Difference**

Table I summarizes the results of our approach and the comparison methods over all 32 BF problem categories. BRKGA shows the best performance, having the best solutions for most BF categories, followed by BS-B. However, both of these come at a computational cost with BS-B taking up to 10 minutes for the largest instances, and BRKGA taking over a minute which suggests scalability could be a factor for much larger instances (the largest here involves 128 containers and 20 stacks). Our approach did not on average find solutions of the same quality, but appears more scalable. On the other hand, the constructive approaches of TGH and FBH were marginally faster but was on average 4 moves or more worse in solution quality. The authors note that DLTS from [10] reports that it improves upon solutions found by BRKGA, but it does not include a comparison with the BF dataset.

We compare our algorithm's performance to four other state-of-the-art heuristic approaches used for solving the CPMP. All results were taken directly from the cited papers. Three of the four are deterministic so unless otherwise stated they involved 1 run per instance, and the *Moves* value reported is the average of the solution moves found for each instance in the BF category.

- The target-guided heuristic (**TGH**) and beam search with baby moves (**BS-B**) from [12]. Their experiments were run on an Intel 3.4GHz Core i7 CPU using Windows 7.
- The feasibility-based heuristic (**FBH**) from [13]. All the algorithms were written in Java, and experiments were run on an Intel 3.4GHz Core i7 CPU using Windows 7.
- The biased random-key genetic algorithm (**BRKGA**) from [7]. It was implemented in C# and executed using the Mono framework v4.0.0. All experiments were run on Intel Xeon E5–2670 version 0 CPUs, with 600MB of RAM per execution and used only a single thread. The BRKGA was run on each problem instance 10 times, so we report both the best and average moves for each BF instance (labelled "BKRGA (min)" in their paper).

Figure 3 presents results directly comparing BRKGA and our proposed ILS, in terms of average best moves and average runtime. Overall we see that our approach follows BRKGA, while runtime for the latter is significantly greater for the 8 instances with over 100 containers (BF13-16,BF29-32). In terms of CPU power, our machine was faster than those used for BRKGA and BS-B, but by less than a factor of 2 which doesn't account for the orders of magnitude difference in some instances.

C. Component evaluation

We further performed experiments on the individual components of our algorithm, to assess their contribution to the overall effectiveness of the approach. The baseline configuration for comparing the impact consisted of:

- Best Neighbour Tie Breaking Strategy: **Random**
- Stack Selection Strategy: **Random**
- Clear Stack Strategy: **Random**

1) Neighbour Tie-breaking Strategy: Figure 4 shows the effect of different neighbour tie-breaking strategies. Both *Highest_Last_Container* and *Smallest_Container_Difference* show improvement over the baseline, but the difference between these strategies is negligible.

The results confirm our hypothesis that using the lower bound cost function alone is not enough to determine the best neighbour. Using one of the additional tie-breaking strategies consistently resulted in better solutions. Both strategies focus on selecting containers that, when moved, are less likely to need to be moved again. Moving high value containers is advantageous as they are generally harder to place correctly,

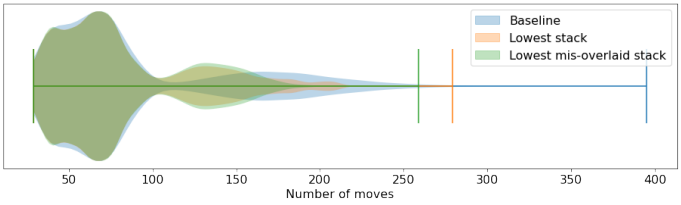


Fig. 5. Comparison of Stack Selection strategies

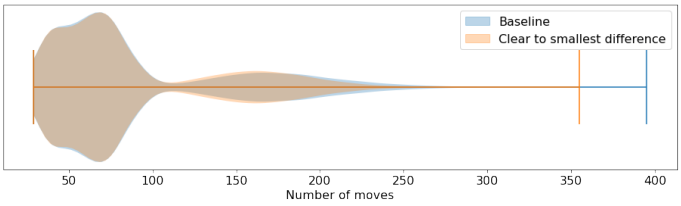


Fig. 6. Comparison of Clear Stack strategies

so should be moved whenever possible. Likewise, moving a container to one with the smallest difference means that we will be packing containers which are closer in priority together, so they are less likely to need to be moved in the future. The distribution of the violin graph shows that the strategies reduce the number of moves across all BF categories, both simple (that require less moves to solve) and difficult.

2) Stack Selection Strategy: Figure 5 compares *Lowest_Stack* and *Lowest_Mis-overlaid_Stack* strategies. Both show significant improvement over the baseline, but the difference between these two strategies is negligible.

Selecting lower stacks to clear means less containers to move to clear them. Also, moving less containers means that they are less likely to cause mis-overlays in other stacks. The graph shows that the improvement is mainly for more difficult BF categories .This is likely due to the fact that these more difficult BF categories have more stacks in the bay and higher proportion of mis-overlays, meaning that the decision of which stack to clear is more important.

3) Clear Stack Strategy: Figure 6 surprisingly showed only a minor improvement using the *Clear_To_Smallest_Difference* strategy compared to the baseline. The lack of real difference between clearing containers to random stacks versus moving them to ones where they will be closer in priority is surprising. Minimizing the difference between container priorities was a successful strategy when breaking ties for neighbour selection, but did not have the same impact here.

V. Conclusions

We presented a novel ILS approach for the CPMP that performs well even on large problem instances. Analysis of the components of our approach showed that interestingly selecting which stack to clear, as part of the perturbations, is more important than where the containers are cleared to. Furthermore we showed that, due to approximating the cost using the lower bound, a dedicated tie-breaking strategy can give large benefits.

Overall our results showed a good tradeoff between runtime and solution quality, with even the largest instances (requiring between one and ten minutes for the state-of-the-art search approaches) taking less than two seconds. This demonstrates the scalability of our approach, which would be an important consideration for real world port scenarios.

For future work, there are several possible avenues. There are a number of improvements that could be investigated, such as adding a tabu list to ensure moves are not undone. Adding the unrelated move symmetry and transitive move avoidance from [5] to reduce the number of nodes in the search tree might make it possible to use more moves in the neighbourhood. Also, using the more recent improved lower bounds from Tierney et al. (2017) [5] and Tanaka et al. (2019) [16] as the cost function may yield better solutions.

References

[1] C. Bierwirth and F. Meisel, "A survey of berth allocation and quay crane scheduling problems in container terminals," *European Journal of Operational Research*, vol. 202, no. 3, pp. 615–627, 2010.

[2] M. Veloqui, I. Turias, M. Cerbán, M. González, G. Buiza, and J. Beltrán, "Simulating the landside congestion in a container terminal. the experience of the port of naples (italy)," *Procedia-Social and Behavioral Sciences*, vol. 160, pp. 615–624, 2014.

[3] Y. Han, L. H. Lee, E. P. Chew, and K. C. Tan, "A yard storage strategy for minimizing traffic congestion in a marine container transshipment hub," *OR spectrum*, vol. 30, no. 4, pp. 697–720, 2008.

[4] T. Wang, X. Tian, and Y. Wang, "Container slot allocation and dynamic pricing of time-sensitive cargoes considering port congestion and uncertain demand," *Transportation Research Part E: Logistics and Transportation Review*, vol. 144, p. 102149, 2020.

[5] K. Tierney, D. Pacino, and S. Voß, "Solving the pre-marshalling problem to optimality with a* and ida," *Flexible Services and Manufacturing Journal*, vol. 29, no. 2, pp. 223–259, 2017.

[6] Y. Lee and N.-Y. Hsu, "An optimization model for the container pre-marshalling problem," *Computers & operations research*, vol. 34, no. 11, pp. 3295–3313, 2007.

[7] A. Hottung and K. Tierney, "A biased random-key genetic algorithm for the container pre-marshalling problem," *Computers & Operations Research*, vol. 75, pp. 83–102, 2016.

[8] K. H. Kim and G.-P. Hong, "A heuristic rule for relocating blocks," *Computers & Operations Research*, vol. 33, no. 4, pp. 940–954, 2006.

[9] A. Bortfeldt and F. Forster, "A tree search procedure for the container pre-marshalling problem," *European Journal of Operational Research*, vol. 217, no. 3, pp. 531–540, 2012.

[10] A. Hottung, S. Tanaka, and K. Tierney, "Deep learning assisted heuristic tree search for the container pre-marshalling problem," *Computers & Operations Research*, vol. 113, p. 104781, 2020.

[11] M. Caserta and S. Voß, "A corridor method-based algorithm for the pre-marshalling problem," in *Workshops on Applications of Evolutionary Computation*. Springer, 2009, pp. 788–797.

[12] N. Wang, B. Jin, and A. Lim, "Target-guided algorithms for the container pre-marshalling problem," *Omega*, vol. 53, pp. 67–77, 2015.

[13] N. Wang, B. Jin, Z. Zhang, and A. Lim, "A feasibility-based heuristic for the container pre-marshalling problem," *European Journal of Operational Research*, vol. 256, no. 1, pp. 90–101, 2017.

[14] C. Expósito-Izquierdo, B. Melián-Batista, and M. Moreno-Vega, "Pre-marshalling problem: Heuristic solution method and instances generator," *Expert Systems with Applications*, vol. 39, no. 9, pp. 8337–8349, 2012.

[15] R. Jovanovic, M. Tuba, and S. Voß, "A multi-heuristic approach for solving the pre-marshalling problem," *Central European Journal of Operations Research*, vol. 25, no. 1, pp. 1–28, 2017.

[16] S. Tanaka, K. Tierney, C. Parreño-Torres, R. Alvarez-Valdes, and R. Ruiz, "A branch and bound approach for large pre-marshalling problems," *European Journal of Operational Research*, vol. 278, no. 1, pp. 211–225, 2019.