ABSTRACT

The performance of a container terminal depends on many aspects of operations. This paper focuses on the optimal sequencing of a yard crane (or YC for short) for serving a fleet of vehicles for delivery and pickup jobs. The objective is to minimize the average vehicle waiting time. While heuristic algorithms could not guarantee an optimal solution, a conventional mathematical formulation such as mixed integer program would require too much computing time. We present two new algorithms to efficiently compute YC dispatching sequences that are provably optimal within the planning window. The first algorithm is based on the well-known A* search along with an admissible heuristics. We also incorporate this heuristics into a second backtracking algorithm which uses a prioritized search order to accelerate the computation. Experimental results show that both new algorithms perform very well for realistic YC jobs. Specifically, both are able to find within seconds optimal solutions for heavy workload scenarios with over $2.4 \times 10^{18}$ possible dispatching sequences. Moreover, even when the vehicle arrival times are not accurately forecasted, the new algorithms are still robust enough to produce optimal or near-optimal sequences, and they consistently outperform all the other algorithms evaluated.

KEYWORDS

Decision-making, Optimization, Algorithms, Container Terminal
1 INTRODUCTION

Containerization has revolutionized cargo shipping and resulted in increased global cargo flow. Container terminals serve as crucial hubs in the transportation chain of trade flows. In providing efficient services, minimizing vessel turnaround time is a common and key performance goal of terminal operations. The vessel turnaround time is calculated by vessel unberthing time minus vessel berthing time. When a vessel berths at a terminal, a number of Quay Cranes (QCs) are allocated to serve the vessel. QCs first unload containers from the vessel onto in-terminal vehicles for transferring them to the container storage yard. A vehicle would arrive at a specific job location at a yard block and be served by a Yard Crane (YC) to pick up the container and to temporarily store it in the yard block. The operation of loading containers onto a vessel is carried out in the reverse order. Previous studies on YC dispatching/scheduling have pointed out that YC operations are of great importance and likely to be a potential bottleneck to the overall terminal performance [1].

One common way to organize the yard operations is to partition a container storage yard into a number of zones for each planning window. Each zone is handled by one or two YCs according to the predicted number of loading and unloading operations in the next planning window, e.g. 1 or 2 Hours. In most cases, the partition is done in units of yard blocks. Because YC gantry to a different row may take around half an hour or more, yard blocks in different rows usually will not be grouped into the same zone. Figure 1 shows a possible partition of a storage yard in a typical container terminal where a zone may not be whole blocks.

In a yard block, containers are arranged in a number of rows and slots as shown in Figure 2. A number of adjacent rows across a few slot locations form a cluster. Vehicles travel along lanes for container jobs.

Figure 1. Storage yard partitioned into zones
When a vessel is unloading, vehicles carry containers to different yard clusters. When multiple vessels are loading and unloading at the same time, vehicles will arrive at different slot locations. Local trucks carrying export containers may also arrive at any time to unload at certain designated slot locations. As a result, YCs need to move between different slot locations in their assigned zones to serve vehicle jobs. When an YC is busy serving other vehicle(s), a vehicle needs to wait for it. A vehicle may also need to wait for the YC to move to its job location.

The overall objective of terminal operations is to reduce vessel turnaround time [8]. While trying to minimize vessel turnaround time, the terminal also gets the benefits of improved berth utilization and higher productivity. This also means lower manpower costs. Therefore at the yard side, YCs need to service coming vehicle jobs as fast as possible to minimize vehicle delays. This helps in supporting unblocked vehicle flows to the quay side to achieve overall objective of the terminal.

YC management involves two types of problems: YC deployment and YC dispatching. YC deployment is the problem of deploying YCs to various parts of the yard to serve in a zone for a planning period. YC dispatching is the problem of dispatching YCs to serve various vehicle jobs in its assigned zone within a planning period. The main focus of this paper is the YC dispatching problem. The YC dispatching algorithm proposed is part of the hierarchical YC management scheme for container terminals that we are working on. To help understand the context, the hierarchical YC management scheme is described in Section 3.

The YC dispatching problem is sometimes referred to as the YC routing problem in literature because YCs are routed among the various job locations. It is a complicated problem because: (1) both the job arrival times and the job locations affect the dispatching sequence. For example, a job arriving later but nearer to the current YC location could possibly be a better choice as next job than a job arriving earlier but at a location...
further away. It also implies that the choice of the next job is affected by the YC’s location and finishing time of the current job. Therefore the problem is sequence-dependent and could not be easily broken into simpler subproblems. This makes techniques like dynamic programming not quite applicable. (2) An idle YC can move to the chosen next job location before the actual job arrival to shorten vehicle waiting time. This is possible if the next several job arrivals can be predicted. So the time spent on YC gantry may or may not contribute to vehicle waiting times.

The main difficulty of the YC dispatching problem is possibly its computation complexity. The problem of Single YC dispatching is NP-Hard [3], which means the time it takes to find the optimal dispatching solution is likely to increase exponentially with the problem size. An YC with 10 job requests would have over 3.6 million possible dispatching solutions. Even when the planning window is small, it could still be too time consuming to find the optimal solution. The difficulty is aggravated as multiple YCs need to be dispatched in the yard. And in real world applications, operations are carried out continuously over time so computation for a solution has to meet the real time constraint. Conventional approaches include Mixed Integer Programming (MIP) and heuristic methods. MIPs (e.g. [4]) are computationally intensive and only suitable for small-size problems while heuristic methods reduce computation time but sacrifice solution optimality (e.g. [5], [12]).

In handling the YC dispatching problem, we propose two optimal algorithms for an YC to handle the jobs in its assigned zone within a planning window efficiently. The algorithms take the predicted job arrival times as input. The two algorithms find the best dispatching sequence with reasonable computational time in solving problems of practical sizes. This is achieved by using domain-specific knowledge beyond the definition of the searching problem itself. The first algorithm is a modified A* search algorithm. Experiments show that this algorithm could find the optimum out of $2.4 \times 10^{18}$ possible dispatching sequences in about 2 to 4 seconds. We further hybrid the domain-specific knowledge with a Recursive Backtracking algorithm (RBA*) to improve the memory usage limitation. The RBA* algorithm is an anytime algorithm.

One concern of YC dispatching based on the predicted vehicle arrivals is the dynamics of these arrivals which make accurate predictions difficult. The Advanced traffic information systems (ATIS) that can provide road-users and traffic managers with accurate and reliable real-time traffic information have been widely studied in recent years. A surge of research into reliable and accurate traffic and travel time prediction mod-
els have been developed in the past decades [2]. With real-time tracking of terminal assets, vehicle job arrivals in near future could be predicted. Predicting vehicles’ arrival times in container terminals is relatively simpler than other problems because vehicle tracks in terminals are simpler than road networks in cities and vehicle speeds in terminals are more uniform than those in public roads. However it is possible that sometimes actual vehicle job arrival times deviate from the predictions. We evaluate the performance of the proposed algorithms under noisy conditions and show that our algorithms are robust enough to outperform the other heuristics.

The rest of the paper is structured as follows. In Section 2 we review the related works. Problems in YC management are discussed and a hierarchical YC management scheme is described in Section 3. A formal description of the YC dispatching problem is given and the problem is reduced to a TSP with dynamic edge weights in Section 4. Our proposed algorithms are presented in Section 5. Section 6 is dedicated to the experimental evaluation of the performance of the proposed algorithms. Conclusion is drawn in Section 6.

2 RELATED WORK

The problems of scheduling and dispatching resources in container terminals have been widely studied in recent years. Surveys on researches of various container terminal operations have been done by Vis and de Koster [7], Steenken et al. [8] and Stahlbock and Voss [9].

Some early efforts in YC dispatching started by addressing the single YC loading only problem. The loading plan of a QC and the containers stock at each bay (clustered slots) are known beforehand. The solutions focus on the sequence of bay visits and the number of containers to be picked up at each bay while the individual container pick-up sequence within a specific bay is left to the crane operator. Kim and Kim [4] addressed this loading only problem using Mixed Integer Programming (MIP) for routing single YC to support the loading operations of a vessel. A beam search heuristic was proposed later and compared with a Genetic Algorithm (GA) [5] [10]. However, for large problems, the MIP model has limited applicability due to the excessive computational time while heuristics will not guarantee to generate optimal solutions. In addition, the assumption of having dedicated yard cranes just to support vessel loading operations in these works is not always the best for terminals with many berths and more yard blocks than cranes.
Several works studied the problem with 2 or more YCs. Due to the problem complexity, MIP models were commonly employed just to formulate yard related problems while heuristic methods were proposed to find near-optimal solutions. Jung and Kim [11] considered 2 YCs working in one shared zone to support vessels loadings with a GA and a Simulated Annealing (SA) algorithm. Cao et al. [12] considered Double-rail-mounted gantry (DRMG) crane systems where two YCs can pass through each other along a row of blocks with a combined greedy and SA algorithm. Ng [13] studied the problem of scheduling multiple YCs to handle jobs with different ready times within a yard zone with MIP and heuristics.

The combination of MIP models and heuristics methods has been employed for the YC deployment problem [14 - 17]. Decisions at this level deploy YCs among the yard blocks for a few times in a day based on forecasted workload typically in terms of the number of container moves or vehicle jobs. However, the number of container moves is not an accurate estimate of the YC workload since the time spent on YC gantry can vary significantly depending on the service sequence of jobs. Other studies extended to a broader scope by including several related subproblems in one integrated model. Handling equipments of YCs, QCs and in-terminal vehicles were all considered in Chen et al. [18], Lao and Zhao [19], Zeng and Yang [20]. Froyland et al [21] considered the problem of operating a landside container exchange area including YC dispatching, short-term container stacking and allocation of vehicle delivery locations.

The research work today have helped much in finding better ways to manage container terminals. However, the hardcore single YC dispatching problem with detailed job serving sequence is not satisfactorily solved. Migration into real world applications still appears unlikely as MIP models require too much time for the NP-hard problem while using heuristics cannot guarantee optimal YC performance.

The consequence of time-consuming or suboptimal solutions of this hardcore problem is serious. As terminals are operating continuously with multiple YCs, small performance downgrades for a single YC over a period of time might result in serious overall YC performance downgrades in a long term. As YC subsystem is tightly coupled with in-terminal vehicle subsystem and QC subsystem, the poorly performing YC subsystem would have significant negative impact on them. This will certainly lead to low efficiency and productivity of the whole terminal. Therefore, a better solution to the hardcore single YC dispatching prob-
lem is demanded. In this paper, we propose new algorithms to incorporate the key features of the A* search approach to guarantee solution optimality in the YC service zone within a planning period.

3 YARD CRANE OPERATION MANAGEMENT

The conventional YC management follows a top-down approach to make decisions at higher levels without considering decisions at lower levels. After equipment ordering at the beginning of a shift, at the higher level, a common practice is to initially assign YCs to various yard blocks to work. Then a re-distribution of YCs among the yard blocks is done from time to time to match the dynamically changing workload. The workload in a future time interval is often simply estimated by the number of jobs expected [14, 15] and the frequency of YC re-distribution is usually picked based on past experience. At the lower level, job serving sequence for the YC(s) in charge of a block may be left to the crane operator(s) to decide or solved by methods in [5], [10-13]. Using this approach, it is difficult to achieve the best possible balance of workloads among YCs to minimize the delays of vehicles at the yard side. The reasons are as follows.

a) Block-based Workload Partition

Since the number of YCs is often not the same as the number of yard blocks, partitioning in units of blocks may lead to one of the following scenarios: (1) some blocks do not have any YCs in charge even though there are jobs arriving; or (2) some YCs are in charge of more than one block; (3) some blocks have more than one YC. In the first situation it will result in long vehicle waiting times. In the second scenario, an YC in charge of 2 blocks (or more) may have too many jobs to handle. The third scenario needs carefully synchronized YC operations to avoid YC clashes in order to avoid long YC waiting times.

b) Workload Prediction

Obviously, a successful workload balance among multiple YCs highly depends on the accuracy of the future workload estimation. However, estimations based on the number of jobs may widely deviate from the true “workload” as 1) For the same number of jobs, a YC will incur different gantry times due to the different job locations or different job serving sequences; 2) For the same set of jobs, different arrival sequences and arrival times will result in different job serving sequences and therefore different gantry times. So workload
prediction by the number of jobs is not accurate which may make YC allocation algorithms less effective in balancing workload.

c) **Re-distribution/Re-partition Frequency**

Since the yard operation is a complex system running 24 hours continuously, the management is typically based on rolling time periods. In general, the trade-off issue is that a high frequency of re-distribution/re-partition leads to flexibility in coping with dynamic workload but with possible performance loss due to local optimization for short planning windows. It incurs higher computational costs. In addition, the movement of a Rubber Tired Gantry Crane to a different row involves two 90 degree turns which take much longer time than linear gantry. Therefore, frequent re-distribution of YCs in this way may delay the vehicle movements by blocking the lanes. It is often very hard to decide on the frequency that offers the best trade-off.

To overcome the difficulties above, we propose a hierarchical scheme for YC operation management which is organized into three levels as shown in Figure 3. Suppose a suitable number of YCs has been assigned to work for the current shift.

<table>
<thead>
<tr>
<th>Level 1</th>
<th>Deployment of YCs to different Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2</td>
<td>Time Partition into planning windows</td>
</tr>
<tr>
<td></td>
<td>Space Partition into YC working zones</td>
</tr>
<tr>
<td>Level 3</td>
<td>YC dispatching in individual zones</td>
</tr>
</tbody>
</table>

Figure 3. Hierarchical YC operation management scheme

Level 1 deploys YCs among different rows at suitable times based on predicted future workload. This is done a few times during a shift of 8 hours. The objective at this level is to minimize the mismatch of YC assignments among the rows of yard blocks and to reduce the number of unnecessary time-consuming cross gantries. Cross gantries only occur at times necessary to balance the workload among different rows.

Level 2 serves two purposes. First is to partition the time interval T between two rounds of YC deployment at Level 1 into a number of planning windows of variable lengths. The second is to partition a row of yard blocks into a number of zones for each planning window. A zone can be part of a yard block, one block or span more than one block. Each zone is assigned one YC. Observing the safety constraints to avoid YC
collision, YC zones are typically non-overlapping for the same planning window to guarantee that even in case of noises in job arrival times, no potential collision would occur. We follow the industrial concern in our scheme. The search for the suitable partition of both the time and space will be an iterative process. Starting with a large time window, Level 2 tries to find the space partition. Sometimes the space partition for the proposed time window does not have a satisfactory solution. The reasons may be a lack of sufficient partition points which are areas of at least 8-slots separation between adjacent jobs to achieve a well balanced partition plan. Then the time window will be reduced gradually until a balanced space partition is found. The process continues till the current time interval $T$ to be managed by Level 2 is completely partitioned.

For each zone in a proposed space partition of a planning window in Level 2, Level 3 helps in two ways. First is to determine the serving sequences of vehicle jobs (this is the YC’s dispatching sequence). The second is to provide an estimation of the YC’s performance of the zone in supporting decision makings at Level 2.

The hierarchical scheme aims to achieve the best possible balance of workloads among YCs through flexible time and space partitioning of workload to cope with dynamic workload changes. Instead of making estimation of workload at Level 2 by the number of jobs, the scheme considers a pool of candidate partition solutions. These candidate partition solutions are evaluated with the support of Level 3, which computes the YC dispatching sequences and returns the expected YC performance of these candidate solutions. It also means that in handling such inter-related complex problems, the efficiency of the lower level support is essential. Here the single YC dispatching algorithm at level 3 may need to be called hundreds or even thousands of times to support decision-making at higher levels. The computation of the dispatching solution must be extremely time-efficient and the quality of the dispatching solution must be guaranteed.

In the rest of this paper, we focus on the single YC dispatching problem at Level 3.

4 PROBLEM FORMULATION

4.1 General Description

The following assumptions are made in the YC dispatching model:
Each vehicle job involves one container only. If a vehicle carries two containers stacked to different locations, it will go to the delivery location of the top container first. A vehicle carrying multiple non-stacking containers to the same destination slot location is modeled as multiple container jobs as explained later.

- The slot locations of jobs are pre-determined and the vehicle arrival times can be predicted for a relatively short planning window, e.g. 30 minutes.
- The YC process time of each job can be accurately predicted.
- YC gantry time between two job positions could be predicted with high accuracy as gantry speed is usually quite consistent.
- A terminal environment where vehicle dispatching is done in the just-in-time manner so both delivery and retrieval jobs are to be done as soon as possible. This will help the vehicle productivity.

Usually, predictions of arrivals are only made for jobs in the near future. Also, van Hinsbergen et al. [22] show that certain groups of approaches nowadays are able to predict point to point traffic conditions accurately. Therefore, the assumptions of reliable predictions for vehicle job arrivals in the near-future as inputs for the YC dispatching models are realistic with advanced technologies and recent developments in transportation studies.

In our formulation, the following notations are used:

\[
\begin{align*}
J & = \{1, 2, \ldots, n\} \quad \text{the set of job IDs for the planning window} \\
Pr & = \{0, 1, 2, \ldots, n\} \quad \text{the set of possible immediate predecessors of a job} \\
Su & = \{1, 2, \ldots, n, n + 1\} \quad \text{the set of possible immediate successors of a job} \\
A_i & \quad \text{the arrival time of job } i. \\
P_i & \quad \text{the process time of job } i \text{ by an YC;} \\
M_{ij} & \quad \text{the time for YC gantry from the position of job } i \text{ to that of job } j. \\
S_i & \quad \text{the time a YC starts processing job } i. \\
\end{align*}
\]

J is the set of jobs to be sequenced. Let the start status of an YC be a virtual job 0 and the final status of an YC be a virtual job n+1. When considering the continuous YC work flow, job 0 will be the last job in the previous planning window and job n+1 is the first job in the next planning window as shown in Figure 4. \(m_{0j}\) is the YC gantry time from its position at the start of the time window to the position of job j.
The objective of an YC is to serve every vehicle as quickly as possible so that the vehicles can continuously feed the quay cranes so as to reduce the vessel turnaround time. So, the YC dispatching problem is: For a set $J$ of $n$ vehicle jobs with predicted job arrival times $A_i (i = 1, 2, \ldots, n)$, job processing times $P_i (i = 1, 2, \ldots, n)$ and YC gantry times $M_{ij} (i = 1, 2, \ldots, n; i = 1, 2, \ldots, n)$, find a job sequence for the YC so as to

$$
\text{Minimize } \frac{1}{n} \sum_{i \in J} (S_i - A_i).
$$

$S_i$, the start time of each job $i$, has to be after $A_i$, the arrival time of the vehicle and after the YC moves from its previous job location to the current job location. The YC may immediately start moving towards its next job location after the completion of the processing of the current job.

The YC dispatching model is flexible to include operation conditions where some vehicles carry more than one container. For situations that containers are not stacked on top of each other at the same slot location, they could be simply modeled as several container jobs with the same arrival time. For situations where several containers are to be loaded/unloaded at the same slot one after another, they will be individual jobs with their own vehicle arrival times, possibly one after another. In both situations, the YC dispatching algorithm will find a job serving sequence which returns the minimum total waiting time for all the jobs.

When vehicle arrivals cannot be predicted, an YC can only start to move towards the next job location after the actual job arrival. Job starting time in this case could be derived as in Equation (1) where job $i$ is the current job and job $j$ is the next job. If vehicle arrivals can be predicted and the next job is decided, an YC is able to start moving towards the next job location before the actual vehicle arrival. This is referred to as the pre-gantry ability. Job starting time with pre-gantry ability is shown in Equation (2).

$$
S_j = \max\{S_i + P_i, A_j\} + M_{ij}
$$

\text{(1)}
The advantage of the pre-gantry ability is the possibility of utilizing YC idle time between jobs to transfer between different job locations. In Figure 5a, when the next job is not known in advance, the YC would be idle at the current job location and only start gantry after the arrival of the next job. When job arrivals are known in advance, the YC could start gantry towards the next job location immediately after the completion of the current job. The pre-gantry ability saves partial or the whole gantry time as shown in Figure 5b.

\[ S_j = \max\{S_i + P_i + M_{ij}, A_j\} \]  

(2)

Using the well known three-field classification of scheduling problems by Graham et al. [23], the single YC dispatching problem with the objective of minimizing average (total) job waiting time may be described as \( 1/(r, d)/T \). The first field means single machine. The middle field states the job characteristic of arbitrary job release times \( r_i \) and compatible release times with due times \( d_i \). Compatible release times with due times means for any two jobs \( i \) and \( j \), if \( r_i \leq r_j \), then \( d_i \leq d_j \). The third field specifies the objective function.

In the case of compatible release times with due times, the objective of minimizing total tardiness is equivalent to minimizing total job waiting time. In our problem, \( r = d \).

The \( 1/(r, d)/T \) problem has been shown to be strongly NP-Hard by Koulamas and Kyparisis [24] with zero setup times. Our problem of single YC dispatching has the additional features of sequence-dependent setup times. And the setup times are anticipatory as defined by Allahverdi et al. [25] since YC may pre-
gantry (if possible) to shorten the job waiting times. Our problem could be considered as a $T/(r,d)/\bar{T}$ problem with anticipatory sequence dependent setup times which is at least as hard as or even harder than the original problem. Approaches developed for the normal single machine job scheduling problem with the objective of minimizing total tardiness are not applicable to our problem since that problem usually assumes zero job release times.

4.2 YC Dispatching Reduced to A Tree-search Problem

We believe that solving the YC dispatching problem as described in the last section by integer programming approach will not be feasible for practical applications. Given an YC dispatching problem of n jobs, there are $n!$ possible dispatching solutions. The solution space can be transformed into a tree-search problem. Each job is a node and the virtual job 0 is the starting node. The edge weight from node i to node j has a value equal to the vehicle waiting time for job j if the YC is to do job j immediately after finishing job i. This edge weight is given by

$$W_i = \max\{S_i + P_i + M_{ij} - A_j, 0\}$$

(3)

Note that $S_0=P_0=0$. The task is to find a path of minimum total distance from the start node to a leaf node that visits each job exactly once. From (3), it can be seen that the edge weights are not pre-defined and depend on $S_i$.

![Figure 6. Search Space of the Problem](image)

Figure 6 shows the search space. For a dispatching problem of n jobs, each path from the start node to a leaf node in the tree represents a complete dispatching sequence of height n and there are in total $n!$ such
paths. The objective of the problem is now transformed into finding a path of minimum total cost (i.e. total weights / total job waiting time) from the start node to a leaf node.

5 OPTIMAL SEARCH ALGORITHMS

In finding the least-cost path of the YC dispatching problem, we need to traverse the tree for searching the optimal solution. As the problem is NP-hard, exhaustive search would be time-consuming to perform. Here we propose to use modified A* search to reduce search time and yet to guarantee optimality. We derive a heuristic function using domain knowledge for the modified A* search.

5.1 Modified A* Search

A* search is a common method for graph search to find the optimal path from an initial node to one goal node. It keeps an open list of nodes to be expanded and always tries to select the most promising node first based on an evaluation function \( f(x) = g(x) + h(x) \) where \( g(x) \) is the cost from the start node to \( x \) and \( h(x) \) is the estimated lowest cost from \( x \) to the goal node. If \( h(x) \) never overestimates the true cost \( h^*(x) \), i.e. \( h(x) \leq h^*(x) \), the heuristic \( h(x) \) is admissible. When coupled with an admissible heuristic \( h(x) \), A* search is optimally effective. In other words, no other algorithm will expand less nodes than A* by employing the same \( h(x) \) [26].

Unlike the original A* search, we are not actually looking for one “goal” node in the YC dispatching problem. The search tree is formed such that each path from the root to a leaf node is of height \( n \) representing a complete dispatching sequence of \( n \) jobs. The objective of our tree search is to find a path from the start node to a leaf node (a dispatching sequence of \( n \) jobs) with minimum \( f(x) \) (i.e. total job waiting time). Once a leaf node is reached, the cost \( f(x) \) of this complete dispatching sequence is kept.

We still use the A* cost function \( f(x) \) to select the most promising node to expand, but the criterion for stopping the search is modified accordingly. In the original A* search, the search will stop when a goal node is reached or when the open list is empty. In the modified A* search, the search will stop when we reach a leaf node with a cost \( f(x) \) lower than that of any path in the open list or when the open list is empty. In this way, we formed our definition of the “goal state” in the stopping criteria of A* search while maintaining the
characteristics of A*search: admissible, complete and optimally effective. Pseudocode of the Modified A* is shown in Figure 7. Let

\[ g(x) = \text{cumulated job waiting time from start to node } x \text{ with edge weight } W_y \text{ computed by Equation (3).} \]

\[ h(x) = \text{estimated lower bound cost from node } x \text{ to a leaf node.} \]

\[ N = \text{set of all jobs, } |N| = n. \]

\[ P(x) = \text{set of jobs already in the (partial) path from the root of the tree (start) to node } x. \]

\[ U(x) = N - P(x). \]

**Modified A* Search**  // the cost function \( f(x) \) determines the priority of node \( x \)

Create a node containing start status as \( \text{node}_\text{start} \)

Create a node containing infinite cost as \( \text{node}_\text{best} \)

Put \( \text{node}_\text{start} \) in the priority queue \( q\_open \)

**WHILE** \( q\_open \) is not empty

\[ \text{node}_\text{cur} = \text{remove}_\text{first}(q\_open) \]

**IF** \( f(\text{node}_\text{cur}) \) is no better than \( f(\text{node}_\text{best}) \)

Return \( \text{node}_\text{best} \)

**IF** \( \text{node}_\text{cur} \) is a leaf node

Update \( \text{node}_\text{best} \) as \( \text{node}_\text{cur} \)

**ELSE**

Generate each node_successor of \( \text{node}_\text{cur} \)

**FOR** each node_successor \( x \)

Get true cost \( g(x) \) from start to this node, equation (3)

Estimate lower bound cost \( h(x) \) from this node, equation (7)

Get cost function \( f(x) = g(x) + h(x) \)

Add \( x \) node to priority queue \( q\_open \)

---

**Figure 7. Pseudocode of Modified A* Search**

### 5.1.1 An admissible heuristic \( h(x) \)

A* is optimal in tree-search if \( h(x) \) is an admissible heuristic, that is, \( h(x) \) never overestimates the cost from \( \text{node}_\text{cur} \) to the goal. In this problem, an admissible heuristic means \( h(x) \) never overestimates the total job waiting time to handle the remaining unordered jobs.
Now, the problem is how to evaluate \( h(x) \) where \( x \) is a successor node of \( \text{node}_\text{cur} \) in the algorithm presented in Figure 7. As shown in Figure 8, all successors of \( \text{node}_\text{cur} \), i.e. \( U(\text{node}_\text{cur}) \), are re-indexed in a list \( L \), where \( A_1 \leq A_{i+1} \), for \( i = 1, 2, \ldots \), \( M-1=M\left|U(\text{node}_\text{cur})\right| \).

Consider a node \( x = J_i \). To evaluate \( h(x) \), the cost from \( J_i \) to a leaf node, we need to estimate the minimal total job waiting time for the remaining \( M-1 \) jobs, that is, \( U(x) = U(\text{node}_\text{cur}) - \{J_i\} \). \( U(x) \) can be partitioned into two groups:

\[
U(x = J_i) = C_{\text{earlier}} \cup C_{\text{later}}
\]

\( C_{\text{earlier}} = \{J_j\}, \text{ where } A_j \leq F_i \text{ and } j \neq i \); And \( C_{\text{later}} = \{J_k\}, \text{ where } A_j > F_i \)

As shown in Figure 9, \( F_i \) is the finish time of Job \( J_i \). Jobs in \( C_{\text{earlier}} \) and in \( C_{\text{later}} \) are the ones that arrive earlier or later than \( F_i \) respectively.

For each job in \( C_{\text{earlier}} \), it definitely needs to wait for the time period from its own arrival to the finish time of Job \( J_i \) plus the time period of the YC’s gantry from \( J_i \) to its own job location. The minimum total job waiting time for jobs in \( C_{\text{earlier}} \) is shown in Expression (4).

\[
\sum (F_i - A_j + M_{ij}), \text{ where } j \in C_{\text{earlier}}
\]

We assume additional reshuffling of containers if required is separately done from the vessel loading and unloading process. Therefore, containers to be retrieved from the yard during loading are on top of the stor-
age yard, and containers to be stored in the yard during unloading are just placed on top of the proper slot. In this case, process time differences among jobs are minor and we approximate the crane process time for all jobs by averaged YC process time, $T_s$ time units.

In addition to the waiting time described in expression (4), the second job to be handled in $C_{earlier}$ has to wait at least one unit of $T_s$, the processing time of the first job handled in $C_{earlier}$. The third job to be handled in $C_{earlier}$ has to wait for at least $2*T_s$, the sum of job process times of the first and the second job handled in $C_{earlier}$. Likewise, the last job to be handled in $C_{earlier}$ will have to wait for at least $\lvert C_{earlier} \rvert *1$ units, the sum of the job process times for the previous $\lvert C_{earlier} \rvert - 1$ jobs in $C_{earlier}$.

Thus, combining expression (4) and the waiting times mentioned above, the minimum total waiting time for jobs in $C_{earlier}$ is as follows.

\[
[1 + 2 + \cdots + (\lvert C_{earlier} \rvert - 1)] * T_s + \sum (F_i - A_j + M_{ij}), \text{where } j \in C_{earlier}
\]  

(5)

Some of the jobs in $C_{later}$ may arrive earlier than the earliest reachable time of the YC after it finishes job $J_i$. The minimum total waiting time for jobs in $C_{later}$ is described in Expression (6)

\[
\sum \max \{0, F_i - A_k + M_{ik} \}, \text{where } k \in C_{later}
\]  

(6)

The estimated minimum cost from node $J_i$ to a leaf node $h(x = J_i)$ is now labeled as $h(i)$, where $J_i$ is the $i$th job arrival among all the successors of $node\_cur$. The heuristic $h(i)$, which describes the minimum total job waiting time for the remaining $M-1$ jobs, is then the sum of the two components in (5) and (6):

\[
h(i) = [1 + 2 + \ldots + (\lvert C_{earlier} \rvert - 1)] * T_s + \sum_{j \in C_{earlier}} (F_i - A_j + M_{ij}) + \sum_{k \in C_{later}} \max \{0, F_i - A_k + M_{ik} \}
\]  

(7)

**Proof:** $h(i)$ is admissible

1. The waiting time as expressed by (5) is the very minimum for jobs in $C_{earlier}$ as if each job were the first job to be served after Job $J_i$ and no job in $C_{later}$ would be served before any job in $C_{earlier}$.

2. The waiting time as expressed by (6) is the very minimum for each of the jobs in $C_{later}$ as if it were the first job to be served after job $J_i$.

Since components (5) and (6) do not overlap with each other, $h(i)$ can never overestimate the total job waiting time for the remaining $M-1$ jobs. Thus $h(i)$ can never overestimate the cost to the leaf node and is hence admissible.
5.2 Prioritized Recursive Backtracking with Heuristics

A* algorithm is optimally efficient for any given heuristic function \( h(x) \). However, it has two limitations in real-world applications. First one is memory usage. Because A* keeps all generated nodes in memory and the number of nodes is increasing exponentially with problem size increases, it may run out of memory space. This limits its usage in large-scale problems. The second limitation is that real-time dispatching often requires anytime algorithms. This means an algorithm which could work out a solution quickly and keeps improving it until the optimal is reached or interrupted by real-time constraints to provide the current “best” solution. A* search has actually a breadth-first searching feature and could not quickly obtain a solution (i.e., not anytime).

To overcome these limitations, we propose a Recursive Backtracking (RB) based algorithm. RB is complete and optimal if the depth of a search tree is finite and there is no time constraint. It greatly reduces memory usage by recursively calling sub-problems and keeping only the nodes on the current path. RB could quickly find the first solution by visiting only \( N \) nodes in a problem of size \( N \). It would then backtrack to examine other solutions. If a better solution is encountered during the backtracking process, the knowledge of the current best solution will be updated accordingly. Therefore, RB-based algorithm is anytime such that a current best solution is provided even though the entire search is not finished.

5.2.1 Combination of RB and heuristics \( h(x) \), RBA*

We propose to employ the heuristic which embedded the domain-specific knowledge as in equation (7) to trim the search space of RB. Every time we expand a new node along a search path, the job waiting time could be obtained and the total job waiting time of all planned jobs from the start to the current node \( g(x) \) is updated. The pruning of the search space could be more effective if the lowest cost from the current node \( x \) to a goal node can be computed. Let \( h(x) \) represents the estimated minimum total job waiting time for all unplanned jobs from the current job node. If \( h(x) \) is admissible, in other words, \( h(x) \) never overestimates the cost from the current node to a goal node, the heuristic is by nature optimal because it thinks the cost of solving the problem is less than it actually is [27]. Since \( g(x) \) is the exact cost from start node to the current node \( x \), this implies that \( g(x)+h(x) \) never overestimates the true cost of a solution through node \( x \). The decision to stop searching further the sub-trees of the current node could use the evaluation of \( g(x)+h(x) \). This evaluation function has been proposed in Section 5.1.1. By combining this heuristic into the RB algorithm, we propose
the RBA* which will not miss the optimal solution and will greatly reduce the computation time of RB. Pseudocode is shown in Figure 10.

In Figure 10, optimalJ is created to record the current best complete dispatching sequence. In each iteration of the FOR loop in RBA*, an unplanned job J, will be considered as the current job to be served by the YC. Dynamic data driven simulation of the YC gantry and loading/unloading operation enables the prediction of the cumulated job waiting time $g(J_i)$ from the first job till job J, that is the partially built dispatching sequence in newJ. The minimum total job waiting time for jobs not planned yet $h(J_i)$, is computed according to Equation (7). The sum $g(J_i)+h(J_i)$ represents the estimated minimal total job waiting time for any dispatching sequence which has the partially built dispatching sequence as in newJ. If this sum is smaller than that of the current best dispatching sequence, the function RBA* will be called to explore further the current path in the search tree. Otherwise, the current path is terminated and another unplanned job will be considered as the current job. If the current job is a leaf node, the algorithm has found a complete dispatching sequence and the current best solution will be updated accordingly.

\[
J = \{J_1, J_2, \ldots, J_n\} \quad // \text{Unfinished Job List}
\]

\[
\text{YCDispatching}(J) \{ \quad // \text{This function is triggered when YC starts to serve the last job in the previous planning window}
\]

\[
\text{newJ} = \emptyset; \quad \text{optimalJ} = \emptyset; \quad \text{CurSmallest} = \infty;
\]

\[
\text{RBA*}(J, \text{newJ});
\]

\[
\}
\]

\[
\text{RBA*}(J, \text{newJ}) \{
\quad \text{FOR each job } J_i \in J
\]

\[
\quad // \text{Select } J_i \text{ as the job to serve after jobs in newJ;}
\]

\[
\quad \text{Remove } J_i \text{ from } J \text{ and Append } J_i \text{ to newJ;}
\]

\[
\quad \text{Get } g(J_i) \text{ for jobs in newJ by (3)}
\]

\[
\quad \text{Estimate lower bound cost } h(J_i) \text{ for jobs in } J \text{ by (7)}
\]

\[
\quad \text{IF } g(J_i)+h(J_i) \text{ is smaller than CurSmallest}
\]

\[
\quad \text{IF } J \text{ is not empty}
\]

\[
\quad \text{RBA*}(J, \text{newJ});
\]

\[
\quad \text{ELSE} \quad // \text{TotalWaitingT is smaller than CurSmallest}
\]

\[
\quad \text{Update CurSmallest as } g(J_i)+h(J_i);
\]

\[
\quad \text{Update optimalJ = newJ}; \quad //\text{Store Optimal List}
\]

Figure 10. Pseudocode of RBA*
Guo, Huang, Hsu and Low

In each iteration, after picking job \( J_i \) as the current job, it results in a partial built dispatching sequence \( P(J_i) = \text{new}J \), and jobs left unplanned \( U(J_i) = J \). There are \( \text{factorial}(\mid U(J_i)\mid) \) dispatching sequences with the partially built sequence \( \text{new}J \) as the prefix. Whenever \( g(J_i) + h(J_i) \) is found to be no better than the current best solution, a total of \( \text{factorial}(\mid U(J_i)\mid) \) dispatching sequences are pruned from the search space.

5.2.2 Acceleration with prioritized search order

In the algorithm of RBA*, unnecessary exploration to sub-trees is pruned according to the knowledge of the current best solution. In the worst case, the optimal solution could be reached last and we could still be forced to explore the entire solution space. Therefore, the discovery of a good path which has near-optimal total job waiting time at the early stage of the tree search is crucial to the performance of the RBA*.

As in Figure 10, the recursive algorithm RBA* takes a list of unplanned jobs \( J \) as input. For each job \( J_i \) in list \( J \), it will be considered as a next job and followed by a function call for the remaining jobs \( (J - \{J_i\}) \). To accelerate the search process, we proposed a technique called prioritized search order which is more likely to discover a good dispatching sequence early in the planning process, instead of choosing the next job randomly. Two ordering methods are used and compared:

1. The first method is to use job arrival times as the priority in the search order. The rationale is that considering the earliest arrival job first among all unplanned jobs is more likely to minimize its own waiting time and substantially lead to an optimal or near-optimal path at an early stage and will help in pruning more sub-trees in the subsequent searches.

2. Another method is to sort the jobs by the reachable time (the earliest possible service start time) as described in Equation (8).

\[
T_{\text{Reachable}_{i+1}} = \max\{A_{i+1}, F_i + M_{ij}\}
\]  

(8)

If a vehicle job arrives earlier than the time the YC can gantry to its job location, the reachable time of the job will be the YC’s arrival time to the job location. If the YC can pre-gantry to the location of a job first, the reachable time will be the vehicle’s arrival time. The earliest reachable unplanned job may not be the one with the earliest arrival time. The earliest reachable job may not be the one nearest to the pre-
vious job. The rationale behind this approach is that giving priority to the job that is earliest reachable improves the crane’s throughput and therefore is likely to lead to an optimal or near-optimal solution.

These two methods will be evaluated by simulation experiments.

6 PERFORMANCE EVALUATION

6.1 Experimental Design

To evaluate the performance of the proposed YC dispatching algorithms, simulation experiments were carried out. Parameters used in the simulation experiments, like YC gantry speed and yard block sizes were set according to the ones in a joint project with an industrial partner [6]. Similar settings were also seen in [33] where case studies of a local terminal in Singapore, a regional terminal in South East Asia and a major terminal on the drawing board were carried out. The linear gantry speed of an YC is 7.8km/hour. As reshuffling operations of containers in the storage yard are usually done separately during the lull periods of yard operations, we assume containers to be retrieved are already on the top of the slots and containers to be stored in yard will be placed on top of their slot locations. The YC processing time is thus assumed to be 120s for each container job. Other related recent studies using constant YC process time include Cao et al. [12], Jung and Kim [11] and Lee et al. [28]. The simulation model is programmed using C++ language under Visual C++ 6.0 compiler on Pentium Core2 Quad CPU Q9450 and 3GB RAM.

<table>
<thead>
<tr>
<th></th>
<th>Heavy Load</th>
<th>Normal Load</th>
<th>Light Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone = 1 BLK</td>
<td>180s</td>
<td>240s</td>
<td>270s</td>
</tr>
<tr>
<td>Zone = 1.5BLKs</td>
<td>225s</td>
<td>270s</td>
<td>300s</td>
</tr>
<tr>
<td>Zone = 2 BLKs</td>
<td>255s</td>
<td>300s</td>
<td>330s</td>
</tr>
</tbody>
</table>

Three sets of experiments were conducted to simulate an YC assigned to a zone of 1, 1.5 and 2 yard blocks respectively. Each block has a size of 37 slots. The distance between two neighboring blocks is equivalent to 8 slots in terms of YC gantry time. For each set of experiments, three scenarios of different vehicle arriving rates were tested. The means of the exponential distributions of inter-arrival times are listed in Table 1 in unit of seconds. Heavy load means that job arrival rate is slightly lower than the YC servicing rate (de-
dependent on crane process time + average inter job gantry time). For the same category of workload, a larger service zone is given a larger mean inter-arrival time because YCs need to gantry longer distances. For each experimental setting, 20 independent runs were performed. The slot locations of the jobs are generated randomly within the zone. Other recent studies using randomized container locations include [20], [29], and [30].

Two groups of algorithms are tested as listed in Table 2. One group is the optimal algorithms. As the solution tree of this problem is finite and all branches have the same depth, all algorithms in this group are able to find the optimal dispatching sequence given enough time and/or memory space. In Section 6.2, we compare the computational time of the optimal algorithms. The other group is the heuristic algorithms. Algorithms in this group could find a solution fast but the optimal solution is not guaranteed. SCJF is a heuristic algorithm by Ng [13]. In Section 6.3, we compare the performance between the heuristic algorithms and the most efficient optimal algorithms under noisy conditions. This is to evaluate how they perform when the prediction of job arrival times are not 100% accurate.

<table>
<thead>
<tr>
<th>Table 2. Algorithms used in the experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimal Algorithms</strong></td>
</tr>
<tr>
<td>ExS</td>
</tr>
<tr>
<td>PGS</td>
</tr>
<tr>
<td>A*</td>
</tr>
<tr>
<td>RBA*_A</td>
</tr>
<tr>
<td>RBA*_R</td>
</tr>
<tr>
<td>RBA*_ran</td>
</tr>
<tr>
<td><strong>Heuristic Algorithms</strong></td>
</tr>
<tr>
<td>FCFS</td>
</tr>
<tr>
<td>NJF</td>
</tr>
<tr>
<td>SCJF</td>
</tr>
</tbody>
</table>

### 6.2 Computational Time of Optimal Algorithms

Computational times for planning windows of 12 jobs and 20 jobs are presented in Sections 6.2.1 and 6.2.2 respectively. These are examples of possible window sizes for an YC to be in charge of an assigned zone. Computational costs in terms of the number of nodes explored and the effective branching factors of 10 to 24 jobs are presented in Section 6.2.3. For each experimental setting, 20 independent runs were performed.
6.2.1 Planning window of 12 jobs

Computational time is the factor of interest since all algorithms return the optimal dispatching sequence. To evaluate the mean computational time $\mu$, we obtain the average $\bar{x}$ of 20 runs. $t$-distribution is employed to generate $t$-Confidence Interval (CI) on $\mu$: $\bar{x} - t_{a/2,n-1}s/\sqrt{n} \leq \mu \leq \bar{x} + t_{a/2,n-1}s/\sqrt{n}$. In this study, we generate 95% CI where $t_{0.05/2.20-1} = 2.09$. Computational times (seconds) for dispatching 12 jobs in the work zone of 1 yard block (BLK) are shown in Table 3 where the Upper is calculated by $\bar{x} + t_{a/2,n-1}s/\sqrt{n}$ and the Lower by $\bar{x} - t_{a/2,n-1}s/\sqrt{n}$.

The data in Table 3 show that the Exhaustive Search (ExS) method is able to explore the entire search tree and find the optimal solution in about 3.5 hours when the service zone is 1 block under all three workload cases. PGS performs much better than ExS. PGS always expands the node whose current cumulated waiting time is minimum using $f(x) = g(x)$. It greatly reduces the search space in the problem tree. With the domain specific knowledge embedded in $h(x)$ by equation (7) in section 5.1.1, the proposed A* search further reduces the computational time to less than 1 second. It shows that the proposed $h(x)$ is able to prune the unworthy partial paths effectively.

<table>
<thead>
<tr>
<th></th>
<th>(seconds)</th>
<th>ExS</th>
<th>PGS</th>
<th>A*</th>
<th>RBA*_ran</th>
<th>RBA*_R</th>
<th>RBA*_A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heavy load</strong></td>
<td>Mean</td>
<td>12666</td>
<td>60.2</td>
<td>0.07</td>
<td>52.7</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>12673</td>
<td>97.2</td>
<td>0.11</td>
<td>95.3</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>12659</td>
<td>23.2</td>
<td>0.04</td>
<td>10.1</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td><strong>Normal load</strong></td>
<td>Mean</td>
<td>12662</td>
<td>30.5</td>
<td>0.05</td>
<td>31.5</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>12670</td>
<td>51.7</td>
<td>0.07</td>
<td>58.0</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>12654</td>
<td>9.3</td>
<td>0.02</td>
<td>5.0</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td><strong>Light load</strong></td>
<td>Mean</td>
<td>12670</td>
<td>18.8</td>
<td>0.03</td>
<td>19.8</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>12676</td>
<td>33.9</td>
<td>0.05</td>
<td>36.2</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>12663</td>
<td>3.8</td>
<td>0.02</td>
<td>3.3</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Guo, Huang, Hsu and Low

Table 4. Computational Time of Zone = 1.5BLKs

<table>
<thead>
<tr>
<th></th>
<th>ExS</th>
<th>PGS</th>
<th>A*</th>
<th>RBA*_ran</th>
<th>RBA*_R</th>
<th>RBA*_A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heavy load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>12662</td>
<td>35.5</td>
<td>0.05</td>
<td>27.5</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Upper</td>
<td>12668</td>
<td>57.4</td>
<td>0.07</td>
<td>52.5</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>Lower</td>
<td>12657</td>
<td>13.6</td>
<td>0.02</td>
<td>2.5</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td><strong>Normal load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>12664</td>
<td>15.0</td>
<td>0.03</td>
<td>17.9</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Upper</td>
<td>12681</td>
<td>27.3</td>
<td>0.05</td>
<td>33.2</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Lower</td>
<td>12647</td>
<td>2.8</td>
<td>0.01</td>
<td>2.6</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td><strong>Light load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>12668</td>
<td>9.6</td>
<td>0.02</td>
<td>15.3</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Upper</td>
<td>12673</td>
<td>17.5</td>
<td>0.03</td>
<td>28.0</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Lower</td>
<td>12662</td>
<td>1.7</td>
<td>0.01</td>
<td>2.7</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 5. Computational Time of Zone = 2BLKs

<table>
<thead>
<tr>
<th></th>
<th>ExS</th>
<th>PGS</th>
<th>A*</th>
<th>RBA*_ran</th>
<th>RBA*_R</th>
<th>RBA*_A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heavy load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>12666</td>
<td>23.4</td>
<td>0.04</td>
<td>16.4</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Upper</td>
<td>12676</td>
<td>40.0</td>
<td>0.06</td>
<td>28.0</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>Lower</td>
<td>12655</td>
<td>6.8</td>
<td>0.02</td>
<td>4.7</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td><strong>Normal load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>12670</td>
<td>11.8</td>
<td>0.02</td>
<td>14.9</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Upper</td>
<td>12684</td>
<td>21.0</td>
<td>0.04</td>
<td>27.3</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>Lower</td>
<td>12655</td>
<td>2.7</td>
<td>0.01</td>
<td>2.5</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td><strong>Light load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>12668</td>
<td>7.8</td>
<td>0.02</td>
<td>13.7</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Upper</td>
<td>12682</td>
<td>13.7</td>
<td>0.04</td>
<td>25.6</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>Lower</td>
<td>12654</td>
<td>2.0</td>
<td>0.01</td>
<td>1.8</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Among the three RRA* algorithms, both RBA*_A and RBA*_R perform better than RBA*_ran by finding the optimal solution in about 0.02 to 0.04 seconds. We conclude that the two acceleration techniques of prioritized search order help to find a good solution first and work really well with the heuristic to shorten the computational time. These 3 proposed algorithms (A*, RBA*_A and RBA*_R) perform very well to find the optimal solution in less than 1 second when the service zone of the YC is one yard block. Their computational time difference is not obvious in terms of 95% CI for a planning window of 12 jobs.

Similar results have been obtained for service zones of 1.5 blocks and 2 blocks as shown in Table 4 and Table 5 respectively. This indicates that the algorithms are also effective when the YC is in charge of a larger zone with longer gantry distances.
6.2.2 Planning window of 20 jobs

The computational times of the group of optimal algorithms are further evaluated for a longer planning window of 20 jobs.

Table 6 shows the computational time of Modified A* search and RBA* with two prioritized search orders when the YC service zone is one yard block. All three algorithms work efficiently to find the optimal dispatching sequence in around 2 seconds in terms of sample average from 20 independent runs. Although the two RBA* algorithms explore more nodes in the search tree than A* method, they outperform the A* methods because they do not have the overhead to maintain the open_list queue for deciding which node to visit next. RBA* prioritized by arrival of jobs (RBA*_A) performs slightly better in terms of averaged computational time than RBA* prioritized by reachable time of YCs (RBA*_R). We believe this is because RBA*_R may favor dispatching solutions where some jobs are starved leading to poor initial solutions. These poor solutions are not very effective in pruning later searches for the optimal solution.

These results show that the proposed A* algorithm can produce optimal YC dispatching sequence with very low computational costs. And the proposed combination of RB and A* heuristics with the prioritized search order further improves the performance. Even in the heavy load case, we have 95% confidence that RBA* can find the optimal solution out of over $2.4 \times 10^{10}$ possible dispatching sequences within 2.02 seconds, which is the upper bound in the 95% confidence interval. This is considered good enough to be used in all real time environments.

As a benchmark for our A* search algorithm, PGS runs out of memory in all cases tested for a list of 20 jobs. Employing the evaluation function $f(x) = g(x)$, PGS runs out of memory because it needs to keep a large number of partially evaluated dispatching sequences in the memory. As this algorithm is equivalent to an A* search algorithm whose $h(x)$ is always equal to zero, it also shows that a good admissible heuristic $h(x)$ close to the actual cost $h'(x)$ is important for the success of A* methods.

When the computational time of a search algorithm to get the optimal solution is longer than the planning time window, we interpret it as running out of time. In all tested cases, exhaustive search for a list of 20 jobs would run out of time. Out of 20 independent runs, some runs for RBA*_ran have the same problem.
This means that when the search order is randomized, the initial solutions obtained can be very bad and is not very useful to prune other partial dispatching sequences. It also confirms the importance of prioritized search order in RBA*.

When the workload is heavy, all 3 algorithms listed in Table 6 need to take more time to work out the optimal solution. This can be explained using queuing theory: the closer the job arrival rate is to the process rate (YC handling rate), the higher probability there is to see queuing jobs. When several vehicle jobs wait for the YC service, their relative job locations would be an important factor in determining the optimal service order. The algorithms may need to expand and evaluate more alternative job sequences in finding the optimal, resulting in longer computational times. However, a few seconds are all that is needed.

### Table 6. Computational Time of Zone = 1BLK
(Note: PGS runs out of memory and exhaustive search, RBA*_ran run out of time)

<table>
<thead>
<tr>
<th>(seconds)</th>
<th>A*</th>
<th>RBA*_R</th>
<th>RBA*_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heavy load</td>
<td>Mean</td>
<td>1.91</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>2.65</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>1.17</td>
<td>1.08</td>
</tr>
<tr>
<td>Normal load</td>
<td>Mean</td>
<td>0.43</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>0.62</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>0.23</td>
<td>0.18</td>
</tr>
<tr>
<td>Light load</td>
<td>Mean</td>
<td>0.31</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>0.13</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>0.31</td>
<td>0.26</td>
</tr>
</tbody>
</table>

### Table 7. Computational Time of Zone = 1.5BLKs
(Note: PGS runs out of memory and exhaustive search, RBA*_ran runs out of time)

<table>
<thead>
<tr>
<th>(seconds)</th>
<th>A*</th>
<th>RBA*_R</th>
<th>RBA*_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heavy load</td>
<td>Mean</td>
<td>3.32</td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>4.81</td>
<td>3.52</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>1.82</td>
<td>1.54</td>
</tr>
<tr>
<td>Normal load</td>
<td>Mean</td>
<td>1.30</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>1.95</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>0.66</td>
<td>0.52</td>
</tr>
<tr>
<td>Light load</td>
<td>Mean</td>
<td>0.61</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>0.82</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>0.41</td>
<td>0.33</td>
</tr>
</tbody>
</table>
Table 8. Computational Time of Zone = 2BLKs
(Note: PGS runs out of memory and exhaustive search, RBA*_ran runs out of time)

<table>
<thead>
<tr>
<th>(seconds)</th>
<th>A*</th>
<th>RBA*_R</th>
<th>RBA*_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heavy load</td>
<td>Mean</td>
<td>2.99</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>4.89</td>
<td>3.53</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>1.08</td>
<td>1.07</td>
</tr>
<tr>
<td>Normal load</td>
<td>Mean</td>
<td>0.76</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>1.14</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>Light load</td>
<td>Mean</td>
<td>0.43</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>Upper</td>
<td>0.62</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>Lower</td>
<td>0.24</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Similar results have been obtained for service zone of 1.5 blocks and 2 blocks as shown in Table 7 and Table 8 respectively. Comparing results across three zone sizes, we find RBA*_A always performs the best in terms of mean computational time. RBA*_A is fast enough to compute a dispatching sequence in real time environment while at the same time guarantees to find the optimal solution. It is also fast enough when being called by our Level 2 YC management scheme to find dispatching solutions and expected YC performance for various proposed zones of partition plans and planning windows.

6.2.3 Effective branching factor

One way to characterize the quality of a search algorithm is the Effective Branching Factor (EBF), b*. It is an indicator of how fast an algorithm could find the optimal solution in the solution space. If the total number of nodes explored to find the optimal solution for a particular problem is N, and the solution depth is d, then b* is the branching factor that a uniform tree of depth d would have in order to contain N+1 nodes.

\[ N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d \]

The effective branching factor of an algorithm can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. Therefore, experimental measurements of b* on a small set of problems can provide a good guide to the algorithm’s overall usefulness (Russell and Norvig 2003). Comparison of the search cost and effective branching factors for PGS, Modified A*, RBA* with prioritized search order
by job arrival time (RBA*_A) and RBA* with prioritized search order by reachable time (RBA*_R) from 10 to 24 jobs in incremental steps of 2 jobs are shown in Table 9.

A smaller effective branching factor means less number of nodes explored by an algorithm to find the optimal route. The results show that PGS will face the problem of running out of memory in the current experimental settings when the number of jobs in the planning window reaches 16. It also shows the exponential growth of the number of nodes explored by PGS. The two RBA* methods are slightly inferior to the optimally effective A* method in terms of node expansion. However, they end in smaller computational time because they do not have the overhead to maintain the open_list queue for deciding which node to visit next as in A* search.

<table>
<thead>
<tr>
<th>d</th>
<th>Search Cost (N)</th>
<th>Effective Branching Factor (b*)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PGS</td>
<td>A*</td>
</tr>
<tr>
<td>10</td>
<td>4806</td>
<td>51</td>
</tr>
<tr>
<td>12</td>
<td>35559</td>
<td>94</td>
</tr>
<tr>
<td>14</td>
<td>244077</td>
<td>173</td>
</tr>
<tr>
<td>16</td>
<td>---</td>
<td>333</td>
</tr>
<tr>
<td>18</td>
<td>---</td>
<td>717</td>
</tr>
<tr>
<td>20</td>
<td>---</td>
<td>1487</td>
</tr>
<tr>
<td>22</td>
<td>---</td>
<td>3,947</td>
</tr>
<tr>
<td>24</td>
<td>---</td>
<td>6,305</td>
</tr>
</tbody>
</table>

### 6.3 YC Performance under Noise

It may happen that the predicted job arrival times are not 100% accurate. The actual arrival times may deviate from predicted arrival times because of unexpected events. In these situations, there may be performance loss in planning with predictions. In this section, the YC performance under noisy conditions is evaluated.

Assume Noise follows normal distributions. The prediction error is assumed as follows. If a job is predicted to arrive within 20 minutes, there is 95% confidence that the real arrival time falls in the range ±30 seconds from the predicted arrival. If a job is predicted to arrive in the period from 20 minutes to 40 minutes,
there is 95% confidence that the real arrival time falls in the range ±60 seconds from the predicted arrival. If a job is predicted to arrive at least 40 minutes from the planning time, there is 95% confidence that the real arrival time falls in the range ±120 seconds from the predicted arrival.

Four algorithms are evaluated in the scenarios with prediction noise. FCFS and NJF are two algorithms that do not use prediction information. For algorithms SCJF and A*/RBA*, the dispatching sequence picked using prediction information is evaluated under actual arrival times. R_Opt represents the performance of the optimal dispatching sequence based on the actual arrival times. Root Mean Square Deviation (RMSD) is calculated by comparing performance of selected algorithm with R_Opt for 20 individual runs:

$$\sqrt{\frac{\sum(Selected \ \text{Algorithm} - R_{Opt})^2}{N}}.$$  

The data in Table 10 and Table 11 show the performance of dispatching 12 jobs and 20 jobs under the assumed noise respectively. A*/RBA* outperforms three other algorithms (FCFS, NJF, SCJF) in all tested scenarios. With prediction error, although the dispatching sequence selected by A*/RBA* using prediction information may not be the optimal sequence under actual arrivals, it still perform quite good and much better than pure real time dispatching algorithm (FCFS & NJF). With the same level of noise interference, the A*/RBA* algorithms outperform the SCJF algorithm. When planning a window of 20 jobs, the performance loss is bigger than planning for 12 jobs. Planning for a longer window means handling more jobs with bigger prediction error. It shows the performance of using prediction information will get worse when the prediction error increases.

The experimental results show that using prediction information which is not 100% accurate could still improve terminal performance from purely real time algorithms like FCFS or NJF which does not use prediction information at all. In real world terminal operations, the deviation of predicted arrivals from real arrivals could be monitored. And rolling planning window could be applied to update predictions and schedule proper re-planning when the prediction error is large. More details about rolling window planning could be found in [32].

Table 10. Performance under noise among various algorithms (N = 12 jobs)
<table>
<thead>
<tr>
<th></th>
<th>Mean Average Job Waiting Time</th>
<th>RMSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FCFS</td>
<td>NJF</td>
</tr>
<tr>
<td>1BLK</td>
<td>HeavyL</td>
<td>208.0</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>134.5</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>120.5</td>
</tr>
<tr>
<td>NormalL</td>
<td>HeavyL</td>
<td>256.8</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>204.4</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>175.1</td>
</tr>
<tr>
<td>1.5BLK</td>
<td>HeavyL</td>
<td>283.6</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>239.5</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>215.0</td>
</tr>
<tr>
<td>2BLK</td>
<td>HeavyL</td>
<td>283.6</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>239.5</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>215.0</td>
</tr>
</tbody>
</table>

Table 11. Performance under noise among various algorithms (N = 20 jobs)

<table>
<thead>
<tr>
<th></th>
<th>Mean Average Job Waiting Time</th>
<th>RMSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FCFS</td>
<td>NJF</td>
</tr>
<tr>
<td>1BLK</td>
<td>HeavyL</td>
<td>231.4</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>135.5</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>116.9</td>
</tr>
<tr>
<td>NormalL</td>
<td>HeavyL</td>
<td>335.4</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>255.5</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>208.0</td>
</tr>
<tr>
<td>1.5BLK</td>
<td>HeavyL</td>
<td>362.6</td>
</tr>
<tr>
<td></td>
<td>NormalL</td>
<td>289.1</td>
</tr>
<tr>
<td></td>
<td>LightL</td>
<td>251.5</td>
</tr>
</tbody>
</table>

7 CONCLUSION

We have proposed an YC management scheme for container terminals and proposed algorithms to solve the problem of YC dispatching as part of it. A modified A* search algorithms with an admissible heuristic is proposed to compute optimal dispatching solutions. To overcome the large memory usage limitation of the A* search, we further proposed an RBA* algorithm that combine the advantages of A*search and Backtracking with prioritized search order. Experiments were carried out to evaluate the algorithms under three various workload cases in three different sized yard zones: 1 block, 1.5 blocks and 2 blocks. Results show that the
proposed algorithms consistently perform very well over all tested cases. Our RBA* algorithm with prioritized search order is able to find the optimal solution over $2.4 \times 10^{18}$ possible dispatching sequences within three seconds under heavy workload.

The experiments under noise show that the proposed algorithms perform well even when predictions of arrivals are not 100% accurate. The efficiency of our algorithms, suggests that even when there is heavy noise like large unexpected change in job arrivals, the dispatching sequence may be re-computed efficiently without delaying the YC operations in rolling planning windows.

In our current definition of the YC dispatching problem, predicted vehicle arrival times are input to the algorithm to minimize the average vehicle waiting time. This means that given the number of jobs, we are minimizing the total job waiting time. This works well when all jobs should be finished as soon as they arrive. In practice, some terminal operators would guarantee the external truck drivers carrying import/export containers an upper limit of their waiting times. Internal vehicles for loading and unloading vessels will have higher priority than external trucks. Our YC dispatching algorithms will be able to handle this situation if the input to the algorithm is changed to include vehicle job deadlines in addition to job arrival times. Job arrival times are used in the computation of job finish time in a dispatching sequence but job deadlines are used in the minimization of total job waiting time (job waiting time is zero if it is done by the deadline). We will be doing further studies on this.

In the hierarchical YC management scheme, higher-level planning to determine the workload partition for multiple YCs involves estimating and balancing workloads of zones in various possible time and space partitions. Our proposed algorithms will be a very effective tool to return useful information such as crane workload, total job waiting time, etc, in each zone of each candidate partition plan. This would greatly improve decision makings of choosing a balanced partition at level 2. A preliminary study of workload partition involving space partition was done in Guo et al. [31].

The “hybrid” of modified A* search heuristic $h(x)$ and recursive backtracking with prioritized search order can be applied to other applications where the solution space could be transformed into a search graph. However, the $h(x)$, the estimation of the cost from current node to a goal node, need to be modified to embed the application-specific knowledge accordingly. And the prioritized search order in the recursive backtrack-
ing need to match problem-specific objectives. With proper utilization of application-specific knowledge, the search efficiency in other application could be improved also.

REFERENCE


Guo, Huang, Hsu and Low


