Available online at www.sciencedirect.com
SCIENCE DIRECT®
European Journal of Operational Research 174 (2006) 1247–1259
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

O.R. Applications

A critical-shaking neighborhood search for the yard allocation problem

Andrew Lim, Zhou Xu *

Department of Industrial Engineering and Engineering Management, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 29 April 2004; accepted 28 January 2005
Available online 18 April 2005

Abstract

The yard allocation problem (YAP) is a real-life resource allocation problem faced by the Port of Singapore Authority (PSA). As the problem is *NP*-hard, we propose an effective meta-heuristic procedure, named critical-shaking neighborhood search. Extensive experiments have shown that the new method can produce higher quality solutions in a much shorter time, as compared with other meta-heuristics in the literature. Further to this, it has also improved or at least achieved the current best solutions to all the benchmark instances of the problem.

Keywords: Metaheuristics; Logistics; Packing; Scheduling

1. Introduction

With more than one hundred thousand ship arrivals every year, the Port of Singapore is one of the busiest ports in the world. Competing pressures in limited land use and competition from other regional and international ports has impelled port planners to use as little space within the container yard as possible to reconcile different requests. Each request has a single time interval and a series of yard space requirements during the interval. Since any yard space allocated to a request cannot be released until the end time point of the request, the length of the required space can either increase or remain unchanged as time progresses.

The yard allocation problem (YAP) was first introduced and defined in [3,4], by extending the berth allocation problem [12,13] and other port operations studies [15,1]. We have adopted the definition of the YAP in [3] as follows.

Let E denote an infinite container yard. We are given a set R of n yard space requests. Every request $R_i \in R$ spans from time TS_i to time TE_i and comprises a series of space requirements $Y_{i,t}$

* Corresponding author. Tel.: +852 23588642; fax: +852 23580062.
E-mail address: xuzhou@ust.hk (Z. Xu).

0377-2217/$ - see front matter
doi:10.1016/j.ejor.2005.01.064

each of length $L_{i,t}$, for all $t \in T_i$, where $T_i = \{TS_i, TS_i + 1, \ldots, TE_i\}$ denotes the spanning time periods of R_i. Let $T_{\max} = \max_{i \in R_i}\{TE_i\}$ indicate the length of the time horizon needing to be considered.

To allocate spaces to requests, an allocation mapping F needs to be chosen to assign a starting position $F(Y_{i,t}) \in E$ to every $Y_{i,t}$ where $i \in R$ and $t \in T_i$. Since spaces allocated to a request will be occupied until the request ends, the mapping F must satisfy that for every request $R_i \in R$ and $t \in T_i$ but $t > TS_i$,

$$F(Y_{i,t}) \leqslant F(Y_{i,t-1}) \quad \text{and} \quad F(Y_{i,t}) + L_{i,t} \geqslant F(Y_{i,t-1}) + L_{i,t-1}. \tag{1}$$

Noting that no space can be allocated to different requests simultaneously, the following constraint must be satisfied. For any two different requests R_i, $R_j \in R$ and $t \in T_i \cap T_j$,

$$F(Y_{i,t}) + L_{i,t} \leqslant F(Y_{j,t}) \quad \text{or} \quad F(Y_{j,t}) + L_{j,t} \leqslant F(Y_{i,t}). \tag{2}$$

The objective of the YAP is to minimize yard space needed. This is given by the following:

$$\min_{F} \max_{i \in R, t \in T_i} (F(Y_{i,t}) + L_{i,t}). \tag{3}$$

For example, Fig. 1 shows a layout with only one *valid* request, say R_3, which spans from $TS_3 = 6$ to $TE_3 = 12$. The starting positions for its space requirements $Y_{3,8}$ and $Y_{3,9}$ are $F(Y_{3,8}) = 5$ and $F(Y_{3,9}) = 3$ respectively.

Fig. 2 shows an example of a valid layout of six requests, from R_1 to R_6. Constraints (1) and (2) are satisfied, and the total yard space needed equals to $F_{3,10} + L_{3,10} = 12$ according to (3).

The YAP has been proven to be *NP*-hard [3]. However, it can be transformed to a two-stage decision [3]. In the first stage, a priority sequence σ is selected, i.e. a unique priority, denoted by $\sigma(i)$, will be determined for each request $R_i \in R$, where $\sigma(i) \in \{1, \ldots, n\}$ and $\sigma(i) \neq \sigma(j)$ for any $j \neq i$. The bigger the $\sigma(i)$, the higher the priority of R_i. In the second stage, an allocation mapping F_σ with an objective value of f_σ can be obtained for the given σ through a polynomial time greedy method.

To make this paper complete, here we briefly introduce the method proposed in [3] to decide F_σ for a given σ. Requests are allocated in the order of their priorities, from the highest to the lowest. Supposing all requests with higher priorities than R_i have been allocated, let us consider positions of $Y_{i,t}$ for $t \in T_i$. Let $H_{i,t}$ denote the maximum occupied place in time $t \in T_i$ after all requests with higher priorities than R_i have been allocated. We can call a recursive procedure $DROP(R_i, TE_i, +\infty, F_\sigma)$, as shown in Algorithm

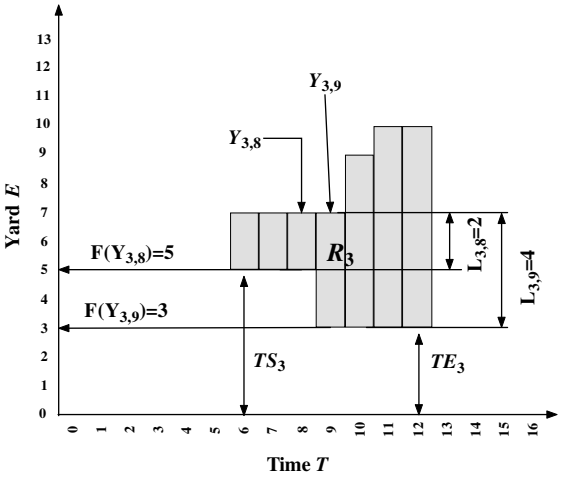

Fig. 1. A *valid* request R_3.

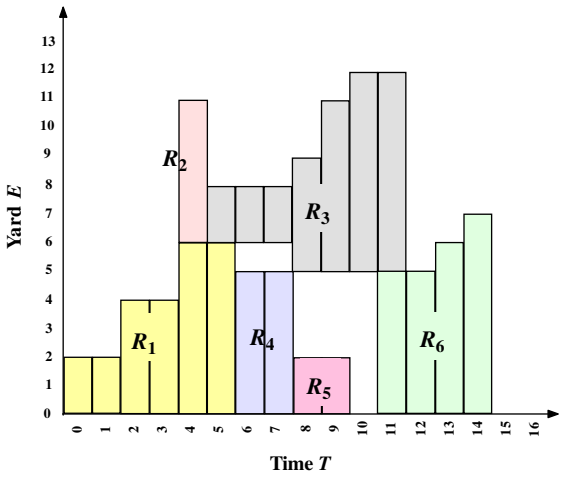

Fig. 2. Six valid requests on yard.

1, to drop $Y_{i,t}$ to a position as low as possible, that is to let $F_\sigma(Y_{i,t})$ be the minimum possible place that is larger than $H_{i,t}$ and satisfies (1) and (2).

To illustrate Algorithm 1, let us consider the six requests of Fig. 2. Assume $\sigma = \{6, 2, 1, 5, 4, 3\}$ and requests except R_3 are decided for their locations. Fig. 3 demonstrates how Algorithm 1 drops R_3. Before dropping as shown in Fig. 3(a), $H_{i,t} = 2$, 2, 4, 4, 11, 6, 5, 5, 2, 2, 0, 5, 5, 6, 7 respectively for $1 \leqslant t \leqslant 14$. In the first recursion of $DROP(R_3, 11, +\infty, F_\sigma)$, we have $e' = 11$ because $L = H_{i,e'} = 5$ is the lowest possible position to drop all $Y_{i,t}$ for $5 \leqslant t \leqslant 11$. Allocating $F_\sigma(Y_{i,11}) = L = 5$, we move to the next recursion $DROP(R_i, 10, 5, F_\sigma)$. In the second recursion, $e' = 11$ because $L = \min(l, H_{i,e'}) = 5$ is the lowest possible position to drop all $Y_{i,t}$ for $5 \leqslant t \leqslant 10$. Allocating $F_\sigma(Y_{i,10}) = L = 5$, we move to the next recursion $DROP(R_i, 9, 5, F_\sigma)$ as shown in Fig. 3(b). Eventually, Fig. 3(c) shows the final layout of R_3. Furthermore, when $\sigma = \{6, 2, 1, 5, 4, 3\}$, the corresponding layout of the six requests is the same as shown in Fig. 2 by dropping R_1, R_4, R_5, R_6, R_2, R_3, through Algorithm 1 sequentially. Therefore, $f_\sigma = 12$ is the total required space by F_σ.

Algorithm 1. $DROP(R_i, e, l, F)$

1: **if** $e \geqslant TS_i$ **then**
2: Let $e' = \max^{-1}\{\min(H_{i,t}, l) + L_{i,t} | TS_i \leqslant t \leqslant e\}$, and thus $L = \min(H_{i,e'}, l)$ is the lowest position to drop space requirements $Y_{i,t}$ for $t = TS_i, TS_i + 1, \ldots, e$;
3: **for** $t = e'$ to e **do**
4: $F(Y_{i,t}) \leftarrow L$;
5: **end for**
6: $DROP(R_i, e' - 1, L, F)$;
7: **end if**

Since [3] has proved that the minimum required space of the YAP equals to $\min_\sigma f_\sigma$, only the optimum priority sequence minimizing f_σ needs to be decided. Therefore, we regard a priority sequence σ as a solution to the YAP and f_σ as its objective value.

Unfortunately, finding the optimum sequence is still *NP*-hard [3]. Therefore, to obtain a near-optimal priority sequence, several meta-heuristics have been attempted in the literature, including simu-

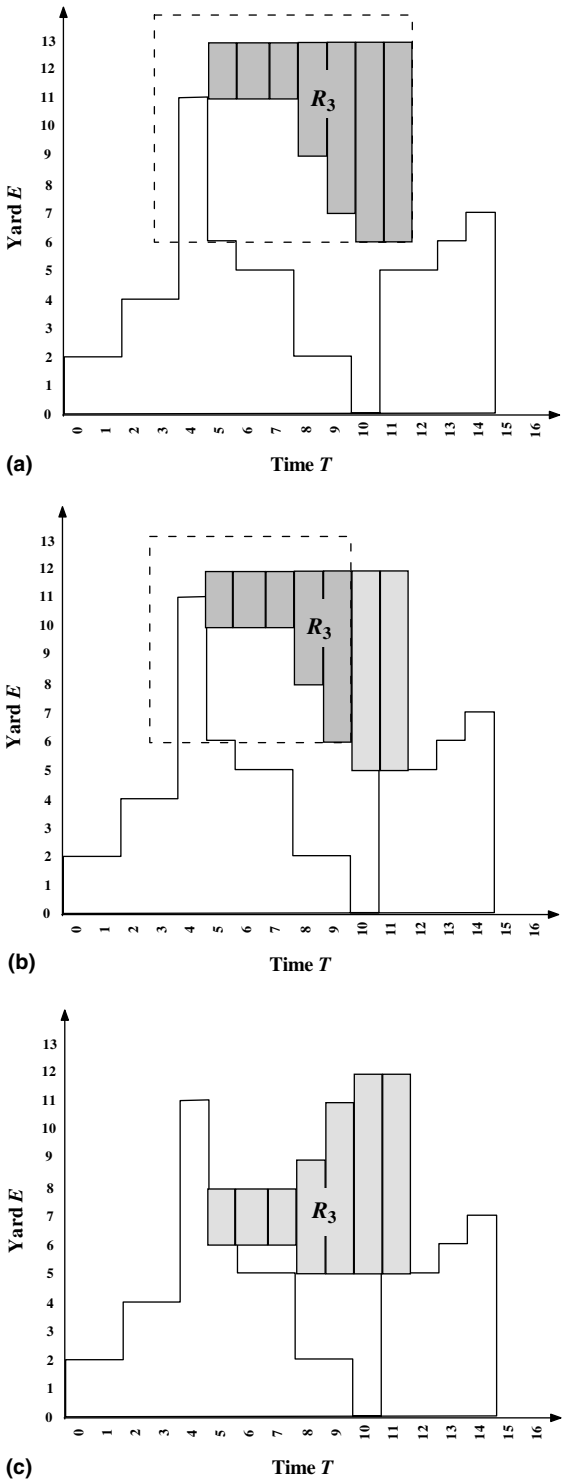

Fig. 3. Dropping R_3 by Algorithm 1.

lated annealing (SA), tabu search (TS), squeaky-wheel optimization (SWO) [3], and genetic algorithm (GA) [2]. However, only GA produces solutions with relatively good qualities, 10% better against other approaches on average and within 8% of the trivial lower bounds in most cases [2]. Even so, the GA is still as time consuming as other approaches. For example, to solve an instance with 213 requests, the GA spends over half an hour to produce a solution that is 14% larger than the lower bound [2].

This paper proposes a more effective meta-heuristic procedure, named critical-shaking neighborhood search (CSNS), to solve the YAP. The basic idea behind the CSNS is to improve the quality of the priority sequence iteratively, from an initial random sequence, by picking some critical requests, shaking their priorities randomly, and then exploring a local search. Extensive experiments have shown that the CSNS can produce solutions to the YAP with higher qualities in a much shorter time, as compared with other meta-heuristics proposed in the literature. Moreover, the CSNS has improved or at least achieved current best solutions to all the benchmark instances of the YAP.

The rest of the paper will be organized as follows. In Section 2, the general framework of the critical-shaking neighborhood search (CSNS) will be described, followed by details of its three essential components illustrated in Sections 3–5 respectively. Section 6 reports and analyzes the experimental results. Finally, conclusions are given in Section 7.

2. General framework

As shown in Algorithm 2, the critical-shaking neighborhood search begins with an initial priority sequence σ_0 for the n requests. It can be generated randomly as [2–4] did for tabu search, simulated annealing and squeaky-wheel optimization. However, experiments have shown that randomly generated solutions are far from (almost 50% larger than) optimum solutions and consume a large amount of time to be improved. Therefore, an effective greedy heuristic will be proposed in Section 3 to generate initial solutions with good qualities.

Therefore, starting with σ_0, an iterative improvement is made as follows. Let σ denote the current priorities. First, we will analyze the structure of σ to pick a set S of k critical requests. Changing the priorities of those critical requests may lead to an improvement of the priority sequence with a high probability. Since it is hard to determine an optimal priority sequence for the k critical requests, we shake their priorities randomly to generate a new priority sequence σ'. The idea of picking and shaking is similar to that of squeaky-wheel optimization (SWO) which was first introduced by Joslin and Clements [6,7,9,10], and was applied to solve the YAP in [3]. However, the SWO proposed in [3] constructs a new priority sequence by always increasing priorities of the critical request, instead of shaking them randomly as we will do in this paper. Experiments have shown that the random shaking is effective in leading to a fast convergence to priority sequences with high qualities. Moreover, since the SWO in [3] picks requests that occupy extra space against an expected limit, there is always a bias to a few requests. *To* overcome this bias, we will adopt a different policy to pick critical requests in Section 5, which is shown to be more reasonable and effective.

Algorithm 2. A General Framework of the Critical-Shaking Neighborhood Search

1: Generate an initial σ_0 of priorities for the n requests randomly;
2: Let $\sigma \leftarrow \sigma_0$ and $\sigma_{\text{opt}} \leftarrow \sigma_0$;
3: let $L \leftarrow L_{\max}$;
4: **while** $L > 0$ **do**
5: Analyze the current σ to pick up a set S of k critical requests from it;
6: Shake priorities of requests in S to generate a new σ';
7: Call a local neighborhood search to improve the σ' to σ'';
8: **if** σ'' is better than σ_{opt} **then**
9: $\sigma_{\text{opt}} \leftarrow \sigma''$;
10: $L \leftarrow L_{\max}$;
11: $\sigma \leftarrow \sigma''$;
12: **else**

```
13:       if σ'' is not worse than σ then
14:          σ ← σ'';
15:       end if
16:       L ← L − 1;
17:    end if
18: end while
```

To improve σ', a local search will be called. Various neighborhoods can be defined. We will consider two of them in Section 4, which are the swap and the move neighborhoods. To enhance the efficiency, the size of neighborhoods is reduced by introducing particular constraints that will have few side-effects on the solution quality but will significantly lessen the time to explore the neighborhoods.

Suppose σ'' is the improved sequence from σ' after the local search. We will update the current best sequence σ_{opt} by σ'' if σ'' is better than σ_{opt}, and move the current sequence σ to σ'' if σ'' is not worse than σ. The iterations will continue until no improvement of σ_{opt} is observed for $L_{\max}$ times. We will fix the value of $L_{\max}$ in experiments as shown in Section 6.

In the following three sections, we will illustrate the key components of the critical-shaking neighborhood search, including the generation of initial priority sequences, picking and shaking critical requests, and an exploration of a local search.

3. Generating initial solutions

In previous literature, initial priority sequences are generated randomly for meta-heuristics [2–4]. However, according to our experiments (shown in Section 6.2), we have found that randomly generated sequences are far away from (i.e., almost 50% larger than) their optimal sequences. This may be one of the reasons for the limited success of those meta-heuristics in [2–4]. Therefore, we propose an effective greedy heuristic instead to generate initial sequences with high qualities, within 20% difference from the lower bounds in most test-cases.

The basic idea of the greedy heuristic is to iteratively assign priorities to a set of requests to maximize the usage of a particular space, until all requests have been allocated. Assuming that the b highest priorities, i.e. $n, n-1, \ldots, n-b+1$, have been assigned to b requests, now let us consider the assignment of priority $n-b$. For every unassigned request R_i and time $t \in T_i$, let $F^b(Y_{i,t})$ denote the place that $Y_{i,t}$ will be allocated if request i has a priority of $n-b$, which can be computed by the recursive dropping Algorithm 1.

Let h indicate the minimum value of $F^b(Y_{i,t})$ among all unassigned requests R_i and all periods $t \in T_i$. Let $R^b = \{R_{i_1}, R_{i_2}, \ldots, R_{i_c}\}$ denote the set of all c requests with $\min_{t \in T_{i_j}}(F^b(Y_{i,t})) = h$ for $1 \leqslant j \leqslant c$, implying that R_{i_j} will occupy the space $[h, h+1)$ if its priority is $n-b$. We now select a subset of unallocated requests from R^b to maximize the usage of the space $[h, h+1)$ by a dynamic programming method as follows.

Let $A(t)$ denote the maximum usage of the space $[h, h+1)$ from time 0 to time t, by allocating space $[h, h+1)$ to some requests, which is in R^b and is ended by time t, without overlapping each other. Thus, values of $A(t)$ can be computed recursively from $t = 0$ to $T_{\max}$ as follows.

$$A(0) = 0;$$

$$A(t) = \max\{A(TS_{i_j}) + L_{i_j} \mid \text{for all } R_{i_j} \in R^b, \text{ and } TE_{i_j} \leqslant t\} \text{ for } 1 \leqslant t \leqslant T_{\max}. \quad (4)$$

It can be seen that $A(T_{\max})$ is the maximum usage of space $[h, h+1)$ for R^b. Suppose S^b is the best subset of requests of R^b selected to achieve $A(T_{\max})$. Note that the $|S^b|$ requests of S^b do not overlap each other. We can assign priorities $n-b, n-b-1, \ldots, n-b-|S^b|+1$ to requests in S^b sequentially. Iteratively in this way, all requests can be assigned their priorities.

The main process above is formulated in Algorithm 3. To clarify the process further, let us consider the six requests shown in Fig. 2. Initially, $R' = \{R_1, R_2, R_3, R_4, R_5\}$. During the first loop, to select requests from $R^1 = R'$, we compute $A(t)$ as follows. First, $A(0) = A(1) = A(2) = A(3) = 0$. Then, $A(4) = 1$ for $TE_2 = 4$, and $A(5) = A(6) = 5$ because R_1 spans from time 0 to time 5. Similarly according to (4), we can obtain

$A(7) = A(8) = 7$, $A(9) = A(10) = 4(11) = 4(12) = 4(13) = 9$, and $A(14) = 13$. Accordingly, R_1, R_4, R_5, R_6 will be assigned 6, 5, 4, 3 as their priorities, because they maximize the utilities of the space $[0, 1)$ by occupying it for thirteen time periods. Afterwards, in the second and the third loops, R_3 and R_2 will be assigned 1 and 2 as their priorities respectively. Hence, the σ_0 produced by Algorithm 3 for the six requests will be $\sigma_0 = (6, 1, 2, 5, 4, 3)$.

Algorithm 3. Generating Initial Sequences in a Greedy Way

1: Let $R' \leftarrow R$, where R' denote the set of unallocated requests;
2: $b \leftarrow 1$;
3: **while** $R' \neq \emptyset$ **do**
4: Compute $F^b(Y_{i,t})$ for $t \in T_i$ by Algorithm 1, where $i \in R'$;
5: Let $h \leftarrow \min\{F^b(Y_{i,t}) | i \in R', t \in T\}$;
6: Determine R^b, the subset of requests in R' that will occupy the space $[h, h+1)$ if its priority is $n - b$.
7: Select a subset S^b of requests from R^b to maximize the usage of the space $[h, h+1)$ by (4) through dynamic programming;
8: Assign $n - b, n - b - 1, \ldots, n - b - |S^b| + 1$ to priorities $\sigma_0(i_j)$ of requests $R_{i_j} \in S^b$;
9: $b \leftarrow b + |S^b|$ and $R' \leftarrow R' - S^b$;
10: **end while**
11: Return σ_0.

Besides producing sequences with good qualities as shown in Section 6.2, Algorithm 3 has a marvellous time performance as well, since it produces sequences with good qualities almost instantly ($\leqslant$0.001 seconds) in the experiments.

Although the greedy method produces reasonably good sequences in a short time, the critical-shaking neighbor search (CSNS) proposed in this paper does not depend on it to produce sequences with good qualities. From experiments shown in Section 6.2, the CSNS always provides high-quality sequences from any initial sequences generated randomly. However, the use of the greedy method to provide an initial sequence to the CSNS will result in a shorter computation time.

4. Local neighborhood search

Given a priority sequence σ' for the n requests, let us consider how to improve its quality by a local neighborhood search. We begin with the definition of the following two operators:

- $Swap(\sigma', i, j)$: which is to swap $\sigma'(i)$ and $\sigma'(j)$ for requests R_i and R_j;
- $Move(\sigma', i, k)$: which is to move R_i to the $(n - k + 1)$th place in the priority order, leading $\sigma'(i)$ to become k and others to be changed properly.

For example, supposing $\sigma' = (6, 1, 2, 5, 4, 3)$ for the six requests in Fig. 2, we know requests can be ordered from highest priority to lowest priority as R_1, R_4, R_5, R_6, R_3, R_2. This leads $Swap(\sigma', 1, 6) = (3, 1, 2, 5, 4, 6)$, and $Move(\sigma', 3, 6) = (5, 1, 6, 4, 3, 2)$ since requests will be ordered as after R_3, R_1, R_4, R_5, R_6, R_2 moving.

Based on the two operators above, an exhaustive local search can be invented to choose the best operator iteratively to improve the total required space required of the current sequence σ' as much as possible, until no improvements can be made. In order to select the best operator among the neighborhood, the local search will have to explore $O(n^2)$ size of neighborhoods, for both *Swap* and *Move*. To reduce the size, only a part of the neighborhoods of σ' will be considered in our local search as follows.

For every request $R_i \in R$, let N_i denote the set of requests whose spanning time periods overlaps with those of R_i. Thus, we select the best operator to reduce the total required space most, only among those $Swap(\sigma', i, j)$ with $j \in N_i$ and those $Move(\sigma', i, k)$ with $k = \sigma'(j)$ where $j \in N_i$. Accordingly, the size of neighborhoods that are explored has been reduced to $O(n\delta)$ for both *Swap* and *Move*, where δ is the maximum size of N_i for $R_i \in R$, which is less than n in most cases. Hence,

the partial neighborhood search can be described in Algorithm 4.

Algorithm 4. Partial Neighborhood Search to improve σ'

1: Let σ'' denote the improved solution from σ';
2: $\sigma'' \leftarrow \sigma'$;
3: *Stop* $\leftarrow$ false;
4: **while** Not *Stop* **do**
5: Choose the best operator, among $Swap(\sigma', i, j)$ with $j \in N_i$ and $Move(\sigma', i, k)$ with $k = \sigma''(j)$ and $j \in N_i$, to generate a new priority sequence x so that the corresponding total required space f_x is minimized.
6: **if** x is better than σ'' **then**
7: $\sigma'' \leftarrow x$;
8: **else**
9: *Stop* $\leftarrow$ true;
10: **end if**
11: **end while**
12: Return σ''.

Further to this, the reduction of neighborhood size has been found to have few side-effects on the quality of the final sequence, as revealed in the testing. For a typical instance shown in Fig. 4, we compare the convergence curves of CSNS with a partial $O(n\delta)$ neighborhood search to that with a full $O(n^2)$ neighborhood search, based on their performance, to solve an instance named R213 which is published in [3]. Both curves start at the same initial sequence with a required space length of 329, and record points, each of which represents the time and the smallest required space length found before the time by the two methods, until the two methods stop respectively. It has been shown that both methods reach solutions of similar qualities (177 and 181), but the CSNS with partial search is significantly faster (almost ten times) than that with the full search scheme. This may be attributed to the small time complexity of the partial search scheme, and the small gap from solutions that CSNS produces to the optimal values. Therefore, even a full neighborhood search cannot reduce the gap further.

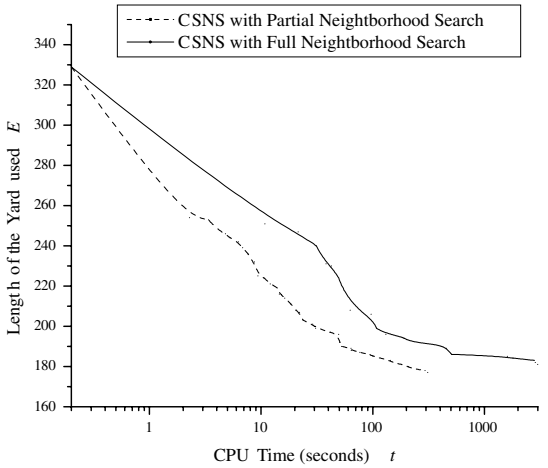

Fig. 4. Comparison between full neighborhood search and partial neighborhood search.

For the above reasons, we will adopt the partial neighborhood search in the CSNS of Algorithm 2 to solve the YAP.

5. Picking and shaking

After the local neighborhood search, the improved priority sequence α will fall in a local optimum. To escape from it, σ needs to be changed properly, like the simulated annealing [8,11,14] which accepts a change to a worse solution with some probabilities, and the squeaky- wheel optimization (SWO) [3] which increases the priorities of critical requests. We will adopt a picking and shaking approach to help σ escape the local optimum before exploring a new neighborhood search. Its basic idea is similar to the SWO proposed in [3], except the following two points.

5.1. Picking critical requests

Firstly, a different measurement from that in SWO of [3] is adopted to pick critical requests. Recall that in the SWO proposed by [3], critical requests are always chosen from those requests that occupy extra yard spaces over a threshold B, where B is one less than the current best yard length. It can be seen that such a threshold-policy often chooses requests with low priorities. However, sometimes, it is more critical to decrease the

priorities of those requests currently with higher priorities. For example, Fig. 5 shows another feasible layout for the six requests of Fig. 2, where the corresponding priority sequence is $\sigma_1 = (6, 3, 4, 2, 1, 5)$ requiring a 13-length yard space. The most efficient way to improve the layout is to reduce the priority of R_3 to 1, i.e. $Move(\sigma_1, 3, 1)$, so that the new priority sequence becomes $(6, 4, 1, 3, 2, 5)$ corresponding to the same layout as Fig. 2, which requires 12-length space only. In contrast, increasing any one of the six requests for σ_1 is useless.

Therefore, in order to measure and pick critical requests more accurately, let us define the following score function $c_\sigma(Y_{i,t})$ for each space requirement of each request R_i according to the priority sequence σ. Based on that, the score function for request R_i is $c_\sigma(R_i) = \sum_{t \in T_i} c_\sigma(Y_{i,t})$, so that the higher the $c_\sigma(R_i)$, the more critical the request R_i.

Now let us define the score function $c_\sigma(Y_{i,t})$ recursively from requests with lower priorities to those with higher priorities. We use B to denote the length of the allocated yard space for sequence σ. For every $t \in T_i$, if the space $[B-1, B)$ is allocated to the requirement $Y_{i,t}$, i.e., $F_\sigma(Y_{i,t}) + L_{i,t} = B$, then its score value $c_\sigma(Y_{i,t})$ is 1. Otherwise, consider whether a space requirement $Y_{j,t}$ exists or not, such that $Y_{j,t}$ is allocated consecutively above $Y_{i,t}$, where request R_j has a lower priority than R_i. This is equivalent to finding a request R_j and time $t \in T_j$ so that

$$F_\sigma(Y_{j,t}) = F_\sigma(Y_{i,t}) + L_{i,t}. \tag{5}$$

If no such $Y_{j,t}$ exists, set $c_\sigma(Y_{i,t})$ to be 0. Otherwise, let $c_\sigma(Y_{i,t})$ share a portion of $c_\sigma(Y_j)$, i.e., let $c_\sigma(Y_{i,t}) = c_\sigma(R_j)/n(R_j)$, where $n(R_j)$ is the number of space requirements whose score will share a portion of $c_\sigma(R_j)$. Obviously, $n(R_j)$ can be directly pre-computed during the calculation of yard length B for a given σ.

Formally speaking, $c_\sigma(R_i)$ and $c_\sigma(Y_{i,t})$ are computed recursively from requests with lower priorities to those with higher priorities as follows.

$$c_\sigma(Y_{i,t}) = \begin{cases} 1 & \text{if } F_\sigma(Y_{i,t}) + L_{i,t} = B, \\ c_\sigma(R_j)/n(R_j) & \text{if } j \text{ and } t \text{ exists} \\ & \quad \text{to satisfy (5)}, \\ 0 & \text{otherwise.} \end{cases}$$

$$c_\sigma(R_i) = \sum_{t \in T_i} c_\sigma(Y_{i,t}), \tag{6}$$

For example, consider the score function for the six requests, whose priority sequence is $\sigma_1 = (6, 3, 4, 2, 1, 5)$, as shown in Fig. 5. Both $c_{\sigma_1}(R_4)$ and $c_{\sigma_1}(R_5)$ are 2, and $c_{\sigma_1}(R_2)$ is 0. Since R_3 is the only request whose score shares $c_{\sigma_1}(R_4)$ and $c_{\sigma_1}(R_5)$, we know $c_{\sigma_1}(R_3) = 4$. As $c_{\sigma_1}(R_3)$ are shared by $c_{\sigma_1}(R_1)$ and $c_{\sigma_1}(R_6)$, both $c_{\sigma_1}(R_1)$ and $c_{\sigma_1}(R_6)$ are 2. Therefore, as expected, R_3 exhibits to be the most critical request.

According to the score function $c_\sigma(R_i)$ of every request R_i, a set S of k critical requests is picked in the following way. Suppose $c_{\max}$ is the maximum score among $c_\sigma(R_i)$ for all $R_i \in R$. We will randomly choose k requests from those with scores higher than $\rho c_{\max}$ to construct S. Different values of parameters k and ρ will affect the performance, as mentioned later in Section 6.

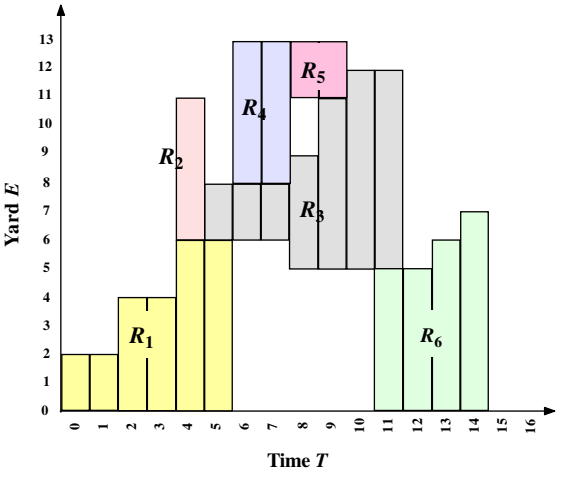

Fig. 5. R_3 is more critical than either R_1 or R_2.

5.2. Shaking critical requests

Secondly, when critical requests are picked, they will be shaken in a different way from that used in the SWO proposed by [3]. Recall that the SWO in [3] always increases the priorities of critical requests because it assumes such an increment

will improve the quality of the solution. However, that is not the case for the YAP problem, especially when some critical requests have high priorities already, like the request R_3 shown in Fig. 5. Therefore, a wiser method that we have used here is to shake their priority randomly, in order to guarantee a chance of meeting a correct priority change for critical requests.

Let S represent the set of critical requests. For each request $R_i \in S$, we randomly choose a priority k having $k = \sigma(j)$ where $j \in N_i$, and then apply the $Move(\sigma, i, k)$ operation on σ. Therefore, a new solution σ' is produced. This picking and shaking process is illustrated in Algorithm 5. Experiments shown in Section 6 reveal that the shaking policy out-performs the traditional increment policy, by nearly 2% better in terms of solution quality.

Algorithm 5. Picking and Shaking σ

1: Pre-computing the length B of the yard used according to σ, and the values of $n(R_i)$ for every $R_i \in R$;
2: Compute the score functions $c_\sigma(R_i)$ and $c_\sigma(Y_{i,t})$ recursively by (6);
3: Pick k requests randomly, from those requests with scores higher than $\rho c_{\max}$ to construct S;
4: For each critical request $R_i \in S$, randomly choose a priority k having $k = \sigma(j)$ where $j \in N_i$, and then apply the $Move(\sigma, i, k)$ operation on σ to produce a new σ' of priorities;
5: Return σ'.

6. Experimental results

6.1. Overview

Extensive experiments have been conducted on a Pentium IV 2.40 GHz personal computer with the program coded in C++. For testing, we adopted the nine benchmark instances, which were randomly generated and shared in http://www.comp.nus.edu.sg/~fuzh/YAP by [3].

Table 1 summarizes the experimental results to compare CSNS with other previous methods. The first column, INST, lists the name of the instances. Then, the LB denotes the lower bound of the min-

Table 1
Experimental results

INST	LB	BEST	M-LS GREEDY		SWO + TS		GA		CSNS + GREEDY		CSNS + RAND	
			OBJ[a]	OBJ[a]	OBJ[a]	CPU[b]	OBJ[a]	CPU[2]	OBJ[a]	CPU[b]	OBJ[a]	CPU[b]
R126	21	22	25	28	24	906.33	24	100.67	**22**	0.53	**22**	2.2
R117	34	34	34	36	**34**	940.33	**34**	147.33	**34**	0.03	**34**	0.5
R145	39	39	43	42	40	1125.00	**39**	261.33	**39**	0.11	**39**	1.56
R178	50	53	64	61	58	1963.33	55	341.00	**53**	10.33	53	104.5
R188	74	77	90	88	82	2610.67	79	440.33	**77**	89.33	**77**	153.44
R173	77	77	89	85	83	2430.67	79	558.33	**77**	20.44	**77**	14.77
R250	83	85	107	92	94	521.00	89	1210.67	**85**	35.75	**85**	60.77
R236	97	98	124	119	107	5673.67	101	817.67	100	76.59	**98**	405.2
R213	164	177	205	198	190	6327.67	187	1440.67	178	275.44	**177**	310.83
AVG	71.00	73.56	86.78	83.22	79.11	2499.85	76.33	590.89	73.89	56.51	73. 6	117.09

[a] Indicates the length of the yard used.
[b] Indicates the time (in seconds) to achieve OBJ.

imum required space, which can simply be chosen as the maximum total length of spaces that are requested simultaneously [3]. In the *BEST* column, current best lengths of required space are presented for each instance. As a comparison, solutions by a multi-start local search and the greedy Algorithm 3 are listed in the columns M-LS and GREEDY respectively. Here, the multi-start local search (M-LS) is to improve various random initial priority sequences by the local search Algorithm 4 and to report the best among all by 360 seconds. Besides these two algorithms, the previous two best methods for the YAP are the squeaky-wheel optimization hybrid with tabu search proposed by [3] and the genetic algorithm proposed by [2]. Both of them provide solutions with good qualities in a relatively short time. Therefore, experimental results from [3] and [2] are also involved in columns of SWO + TS and GA in Table 1. Since [3] and [2] made experiments on a Pentium IV 800 MHz personal computer which is three times slower than ours, their running times have been transformed by a factor of 13 when being cited in Table 1. The last two columns of Table 1 report experimental results of our algorithms. We have used CSNS + GREEDY and CSNS + RAND to denote the two versions of the critical-shaking neighborhood search (CSNS), starting with different ways to generate their initial solutions. The CSNS + GREEDY generates initial solutions by the greedy method of Algorithm 3, while the CSNS + RAND generates them randomly. With the exception of this difference, both of them have the same configurations, including adopting a partial neighborhood search, picking only $k = 1$ critical request among those with at least $\rho = 0.5$ times the maximum critical score, and shaking it randomly. We have run both of the two versions of CSNS five times for every instance, and the best results are reported for each. Furthermore, to simplify the parameter tuning, we chose $L_{\max} = 10^3$ based on a sampling of some values of $L_{\max}$. From the experimental results, $L_{\max} = 10^3$ provides very satisfactory results. Therefore, we did not make extensive test on the choice of $L_{\max}$ as this is not the major focus of our experimentation.

According to Table 1, both of the two versions of CSNS outperform previous best algorithms in terms of both solution qualities and time performances. Among the nine instances, the CSNS + RAND achieves nine best solutions by breaking seven previous benchmarks, and the CSNS + GREEDY achieves seven best solutions by breaking five previous benchmarks. In contrast, the SWO + TS of [3] achieved only one best solution and the GA of [2] achieved only two. It is also worth mentioning that three bench-marks of R117, R145 and R173 have been optimally solved by both the CSNS + RAND and the CSNS + GREEDY, because their best solutions equal to the corresponding lower bounds. Fig. 6 shows the optimum solution to R173, which has never been reported in previous literature. The dark shape consists of a subset of requests of R173, which need to be dealt with simultaneously. Since their total length is equivalent to the total required space, the layout shown in Fig. 6 must be optimum.

Further to the significant improvements in solution qualities, both the CSNS + GREEDY and the CSNS + RAND exhibit an extremely high speed in time performance. When producing better solutions than SWO + TS of [3] and GA of [2], the two CSNS consume much less time. On average, the CSNS + GREEDY and the CSNS + RAND are more than 10 and 5 times respectively faster than the GA of [2], and almost 50 and 25 times respectively faster than the SWO + TS of [3].

Table 1 also reveals that the greedy Algorithm 3 returns relatively good solutions efficiently. It is

Fig. 6. The optimum scheduling of the instance R173.

slightly better than the simple multi-start local search, but consumes much less time (i.e., works almost instantly).

In the following two sub-sections, we analyze the effects on the performance of critical-shaking neighborhood search (CSNS) of initial solutions and picking-shaking policies. This can also explain the reasons for our current configurations of the CSNS.

6.2. Effects of initial solutions

Comparing the CSNS + RAND and the CSNS + GREEDY in Table 1, we have observed that the CSNS + RAND produces slightly better solutions than the CSNS + GREEDY does. This observation is surprising because initial solutions generated by the greedy Algorithm 3 are much better than those generated randomly, as shown in Table 2. Hence, we know that the solution quality of the CSNS does not depend on the initial solution, and the CSNS can provide high-quality solutions from any initial solutions generated randomly in the experiment.

To see the stability of the CSNS more clearly, Table 3 lists the statistics of minimum objective values (MIN), maximum objective values (MAX), average objective values (AVG) and

Table 2
Initial solutions: random vs. greedy

INST	LB	RAND		GREEDY	
		OBJ[a]	GAP (%)[b]	OBJ[c]	GAP (%)[b]
R126	21	36.6	74.29	28	33.33
R117	34	51.6	51.76	36	5.88
R145	39	69.8	78.97	42	7.69
R178	50	97.6	95.20	61	22.00
R188	74	147.6	99.46	88	18.92
R173	77	136	76.62	85	10.39
R250	83	164	97.59	92	10.84
R236	97	187.4	93.20	119	22.68
R213	164	329	100.61	198	20.73
AVG	71.00	135.51	85.30	83.22	16.94

[a] Represents the average objective value over five random initial solutions.
[b] Represents the difference between the objective value and the lower bound as the percentage of the lower bound.
[c] Represents the objective value of the greedy initial solution.

Table 3
Statistics of objective values by the CSNS with the random initial solutions

INST	MIN		MAX		AVG		STD	
	GREEDY[1]	RAND[2]	GREEDY[1]	RAND[2]	GREEDY[1]	RAND[2]	GREEDY[1]	RAND[2]
R126	22	22	22	23	22	22.4	0.00	0.55
R117	34	34	34	34	34	34	0.00	0.00
R145	39	39	39	39	39	39	0.00	0.00
R178	53	53	54	55	53.8	53.8	0.84	0.84
R188	77	77	80	80	78.6	77.8	1.14	1.30
R173	77	77	79	78	77.8	77.2	0.84	0.45
R250	85	85	86	87	85.4	85.8	0.55	1.10
R236	100	98	101	102	100.8	100.6	0.84	1.67
R213	178	177	180	180	179	179.6	1.00	2.19
AVG	73.89	73.56	75.00	75.33	74.49	74.47	0.58	0.90

[1](or [2]) indicates statistics of objective values by the critical-shaking neighborhood search with greedy (or random resp.) initial solutions.

standard deviations (STEV) of objective values, based on experimental results produced by running both the CSNS + RAND and the CSNS + GREEDY five times. The table shows that the two versions of the CSNS have small gaps among their minimum, maximum and average values, and that both of them have small standard deviations (i.e., less than 1).

However, the CSNS + GREEDY exhibits more stability than CSNS + RANDOM does, in terms of relatively smaller standard deviations. The CSNS + GREEDY also has a speed advantage, almost two times faster than the CSNS + RANDOM according to Table 1. This is attributed to its good qualities of initial solutions generated in a greedy way.

Hence, generating initial solutions greedily instead of randomly hardly improves the solution qualities of the CSNS but enhances its speed and stability.

6.3. Effects of picking and shaking policies

As we have claimed in Section 6.1, during the picking and shaking stages of the critical-shaking neighborhood search, only $k = 1$ critical request is picked among those requests having at least $\rho = 0.5$ times the maximum critical score, and it is shaken randomly. To illustrate the reason for the current parameter values and policies adopted, let us make the following comparisons for tuning parameters.

Firstly, Table 4 compares objective values of the CSNS with different values of k. Here, the value of ρ is fixed to be 0.5. The programs with various k are run five times for each of the nine instances, and the best result of each is presented in Table 4. It can be observed that the solution qualities decreases when k increases. Therefore, we chose $k = 1$ for the CSNS.

Secondly, let us consider the parameter ρ. Table 5 presents the solution qualities produced by the CSNS with different values of ρ, where k is fixed to be one. The programs with various ρ are run five times for each of the nine instances, and the best result is listed in Table 5. It can be seen that the CSNS with $\rho > 0$ is much better than that with $\rho = 0$, which supports the effectiveness of the score function (6) in picking critical requests. Therefore, we adopted $\rho = 0.5$, i.e. the best choice according to Table 5.

Finally, let us compare the two different policies of changing priorities for critical requests. One is the shaking policy (SHAKING) which shakes the priority of requests randomly. The other is the increasing policy (INCREASING), proposed by [3] for the SWO, which always increases the priority of critical requests. Here, we fix $\rho = 0.5$ and $k = 1$. According to Table 6, the CSNS with the shaking policy exhibits a slightly better performance than that with the increasing policy, where there is nearly 2% improvement in the quality of solutions. This leads us to choose the CSNS with the shaking policy to solve the YAP.

Table 4
Comparisons of objective values by the CSNS with different k and $\rho = 0.5$

INST	$k = 1$	$k = 2$	$k = 3$
R126	22	23	23
R117	34	34	34
R145	39	39	39
R178	53	55	56
R188	77	79	81
R173	77	78	78
R250	85	88	90
R236	98	102	103
R213	177	177	181
AVG	73.56	75.00	76.11

Table 5
Comparison of objective values by the CSNS with different p and $k = 1$

INST	$\rho = 0.00$	$\rho = 0.25$	$\rho = 0.50$	$\rho = 0.754$	$\rho = 1.00$
R126	24	22	22	22	22
R117	34	34	34	34	34
R145	39	39	39	39	39
R178	56	54	53	53	53
R188	81	77	77	78	78
R173	79	77	77	77	77
R250	90	85	85	84	85
R236	107	100	98	100	100
R213	178	177	177	178	181
AVG	76.44	73.89	73.56	73.89	74.33

Table 6
Comparison of objective values by the CSNS with SHAKING and INCREASING

INST	SHAKING	INCREASING
R126	22	22
R117	34	34
R145	39	39
R178	53	54
R188	77	79
R173	77	77
R250	85	86
R236	98	101
R213	177	182
AVG	73.56	74.89

7. Conclusions

The critical-shaking neighborhood search (CSNS) proposed in this paper is very effective in solving the yard allocation problem (YAP). The basic idea behind the CSNS is to improve the quality of the priority sequence iteratively, from an initial random sequence, by picking some critical requests, shaking their priorities randomly, and then exploring a local search.

Extensive experiments have been conducted to examine the performance of the CSNS in solving the YAP. They have shown that the CSNS has broken or as least achieved all the nine benchmarks mentioned in previous literature. Compared with previous approaches, the CSNS has improved solution qualities in a small amount of time and held a strong stability over different initial solutions. In addition, this paper has also reported the method to configure the proper values of parameters of the CSNS.

The CSNS for the YAP can be directly applied to other berth allocation problems. Our future works include adopting the CSNS to solve the extensive YAP proposed by [5] where two dimensional spaces are requested.

References

[1] E.K. Bish, T.Y. Leong, C.L. Li, J.W.C. Ng, D. Simchi-Levi, Analysis of a new vehicle scheduling and location problem, Naval Research Logistics 48 (2001) 363–385.
[2] P. Chen, Z. Fu, A. Lim, Using genetic algorithms to solve the yard allocation problem, in: Proceedings of Genetic and Evolutionary Computation Conference 2002 (A recombination of the GP-2002 and the ICGA-2002), New York City, USA, 2002, pp. 1049–1056.
[3] P. Chen, Z. Fu, A. Lim, The yard allocation problem, in: Proceedings of the Eighteenth American Association for AI National Conference (AAAI), Edmonton, Alberta, Canada, 2002, pp. 3–8.
[4] P. Chen, Z. Fu, A. Lim, Port yard optimization problem, IEEE Transactions on Robotics and Automation 1 (1) (2004) 26–37.
[5] P. Chen, Z. Fu, A. Lim, B. Rodrigues, The general yard allocation problem, in: Proceedings of Genetic and Evolutionary Computation Conference 2003, Chicago, Illinois, USA, 2003, pp. 1986–1997.
[6] D. Clements, J. Crawford, D. Joslin, G. Nemhauser, M. Puttlitz, M. Savelsbergh, Heuristic optimization: A hybrid ai/or approach, in: Workshop on Industrial Constraint-Directed Scheduling, 1997.
[7] D. Draper, A. Jonsson, D. Clements, D. Joslin, Cyclic scheduling, in: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999, pp. 1016–1021.
[8] B. Hajek, Cooling schedules for optimal annealing, Mathematics of Operations Research 13 (1988) 311–329.
[9] D. Joslin, D. Clements, Squeaky wheel optimization, in: Proceedings of the Fifteenth American Association for AI National Conference (AAAI), Madison, WI, 1998, pp. 340–346.
[10] D.E. Joslin, D.P. Clements, Squeaky wheel optimization, Journal of Artificial Intelligence Research 10 (1999) 353–373.
[11] S. Kirpatric, C.D. Gelatt Jr., M.P. Vecchi, Optimization by simulated annealing, Science 220 (1983) 671–680.
[12] A. Lim, On the ship berthing problem, Operations Research Letters 22 (1998) 105–110.
[13] A. Lim, An effective ship berthing algorithm, in: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999, pp. 594–599.
[14] R. Otten, L. Van Ginneken, The Annealing Algorithm, Kluwer Academic Publishers, Boston, 1989.
[15] F. Sabria, C. Daganzo, Queuing systems with scheduled arrivals and established service order, Transportation Research B 23 (1989) 159–175.