ICS 2011
12th INFORMS Computing Society Conference

© 2011 INFORMS | ISBN 978-0-9843378-1-1
DOI 10.1287/ics.2011.0009

Factory Crane Scheduling by Dynamic Programming

Ionuţ Aron
WorldQuant LLC, Old Greenwich, Connecticut 06870, ionut.aron@gmail.com

Latife Genç-Kaya
Istanbul Şehir University, Istanbul, Turkey 34662, latifegenc@sehir.edu.tr

Iiro Harjunkoski
ABB Corporate Research, 68526 Ladenburg, Germany, Iiro.Harjunkoski@de.abb.com

Samid Hoda, J. N. Hooker
Tepper School of Business, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
{shoda@andrew.cmu.edu, john@hooker.tepper.cmu.edu}

Abstract We describe a specialized dynamic programming algorithm for factory crane scheduling. An innovative data structure controls the memory requirements of the state space and enables solution of problems of realistic size. The algorithm finds optimal space-time trajectories for two factory cranes or hoists that move along a single overhead track. Each crane is assigned a sequence of pickups and deliveries at specified locations that must be performed within given time windows. The cranes must not interfere with each other, although one may yield to the other. The state space representation permits a wide variety of constraints and objective functions. It is stored in a compressed data structure that uses a cartesian product of intervals of states and an array of two-dimensional circular queues. We also show that only certain types of trajectories need be considered. The algorithm found optimal solutions in less than a minute for medium-sized instances of the problem (160 tasks, spanning four hours). It can also be used to create benchmarks for tuning heuristic algorithms that solve larger instances.

Keywords crane scheduling; dynamic programming

1. Introduction

Manufacturing facilities frequently rely on track-mounted cranes to move in-process materials or equipment from one location to another. A typical arrangement, and the type studied here, allows one or more hoists to move along a single horizontal track that is normally mounted on the ceiling. Each hoist may be mounted on a crossbar that permits lateral movement as the crossbar itself moves longitudinally along the track. A cable suspended from the crossbar raises and lowers a lifting hook or other device.

When a production schedule for the plant is drawn up, cranes must be available to move materials from one processing unit to another at the desired times. The cranes may also transport cleaning or maintenance equipment. Because the cranes operate on a single track, they must be scheduled so as not to interfere with each other. One crane may be required to yield (move out of the way) to permit another crane to pick up or deliver its load.

The problem is combinatorial in nature because one must not only compute a space-time trajectory for each crane, but must decide which crane yields to another and when.

A decision made at one point may create a bottleneck that has unforeseen repercussions much later in the schedule. Production planners may put together a schedule that seems to allow ample time for crane movements, only to find that the crane operators cannot keep up with the schedule.

In this paper we analyze the problem of scheduling two cranes and describe an exact algorithm, based on dynamic programming, to solve it. We assume that each crane has been pre-assigned a sequence of jobs to perform. A crane may be required to carry out multiple tasks at various locations before completing a job, and each job may specify a different set of tasks. Time windows may be given for the job as a whole and/or the tasks within a job.

We selected a dynamic programming approach because it not only solves a problem of this kind but accommodates a wide variety of constraints that may arise in factory settings. Any constraint or objective function that can be defined in terms of the current state vector can be implemented. For example, precedence relations may be imposed between jobs, or between tasks in different jobs. A crane may be allowed to pause, or yield to the other crane, between certain tasks but not others. Many linear and nonlinear objective functions are possible, although in practice the objective is normally to follow the production schedule as closely as possible.

The state space is large, due to the fine space-time granularity with which the problem must be solved, as well as the necessity of keeping up with which task a crane is performing and how long it has been processing that task. To deal with these complications we introduce a novel state space description that represents many states implicitly as a cartesian product of intervals. The state space is efficiently stored and updated in a data structure that uses an array of two-dimensional circular queues. These enhancements accelerate solution by at least an order of magnitude and allow us to solve problems of realistic size within a reasonable time.

This research is part of a larger project in which both heuristic and exact algorithms have been developed for use in crane scheduling software. The heuristic method makes crane assignment and sequencing decisions as well as computing space-time trajectories, and it is fast enough to accommodate large problems involving several cranes. However, once the assignments and sequencing are given, the heuristic method may fail to find feasible trajectories when they exist and reject good solutions as a result. We therefore found it important to solve the trajectory problem exactly for a given assignment and sequencing, as a check on the heuristic method. The exact algorithm has practical value in its own right, because two-crane problems are common in industry, and the algorithm solves instances of respectable size within a minute or so. Yet it has an equally important role in the creation of benchmarks against which heuristic methods can be tested and tuned for best performance.

We begin by deriving structural results for the two-crane problem that restrict the trajectories that must be considered. This not only accelerates solution but simplifies the operation of the cranes by restricting their movements to certain predictable patterns. We then describe the dynamic programming algorithm and the state space representation, and we conclude with computational results and directions for further research.

2. Previous Work

Crane scheduling has received a great deal of study, but little attention has been given to the factory crane scheduling problem addressed here. The literature focuses primarily on two types of problems: movement of materials from one tank to another in an electroplating or similar process (normally referred to as *hoist scheduling* problems), and loading and unloading of container ships in a port. Both differ significantly from the factory crane problem.

A classification scheme for hoist scheduling problems appears in Manier and Bloch [13]. These problems typically require a cyclic schedule involving one or more types of parts,

where parts of each type must visit specified tanks in a fixed sequence. The most common objective is to minimize cycle time.

Much research in this area deals with the single-hoist cyclic scheduling problem (Armstrong et al. [1], Liu et al. [12], Phillips and Unger [17]). Even this restricted problem is NP-complete (Lei and Wang [5]).

Several papers address cyclic two-hoist and multihoist problems. One approach partitions the tanks into contiguous subsets, assigns a hoist to each subset, and schedules each hoist within its partition (Lei and Wang [6], Zhou and Li [23]). A better solution can generally be obtained, however, by allowing a tank to be served by more than one hoist. Models for this problem assign transfer operations to hoists and determine when each operation should begin and end (Che and Chu [2], Lei et al. [7], Leung and Levner [8], Leung and Zhang [9], Leung et al. [10], Manier et al. [14], Riera and Yorke-Smith [18], Rodošek and Wallace [19], Varnier et al. [20], Yang et al. [21]). These models do not explicitly compute space-time trajectories but avoid collisions by selecting departure and arrival times that allow hoists to make the necessary movements without interference.

Our problem differs from the typical hoist scheduling problem in several respects. The schedule is not cyclic. The problem is given as a set of jobs rather than parts to be processed. Each crane is assigned a sequence of jobs rather than transfer operations. A job may consist of several tasks to be performed consecutively by one crane in a specified order. The jobs may all be different. Both loaded and empty cranes may be allowed to pause or yield, and permission to do so may depend on which task is being executed. There may also be precedence constraints and a wide variety of objective functions.

Port cranes are generally classified as quay cranes and yard cranes. Quay cranes may be mounted on a single track, as are factory cranes, but the scheduling problem differs significantly. The cranes load (or unload) containers into ships rather than transferring items from one location on the track to another. A given crane can reach several ships, or several holds in a single ship, either by rotating its arm or perhaps by moving laterally along the track. The problem is to assign cranes to loading (unloading) tasks, and schedule the tasks, so that the cranes do not interfere with each other (Daganzo [3], Kim and Park [4], Mocchia et al. [15]).

Yard cranes are typically mounted on wheels and can follow certain paths in the dockyard to move containers from one location to another (Ng [16], Zhang et al. [22]). Port and yard cranes clearly present a different type of scheduling problem than factory cranes.

An early study of factory crane scheduling (Lieberman and Turksen [11]) presents heuristic algorithms that obtain noninterfering solutions only under certain conditions. A worst-case bound is derived for makespan in the two-crane case. However, the method is insufficiently general to address the problem considered here. There is no attempt to apply it to realistic problems, and no computational results are reported.

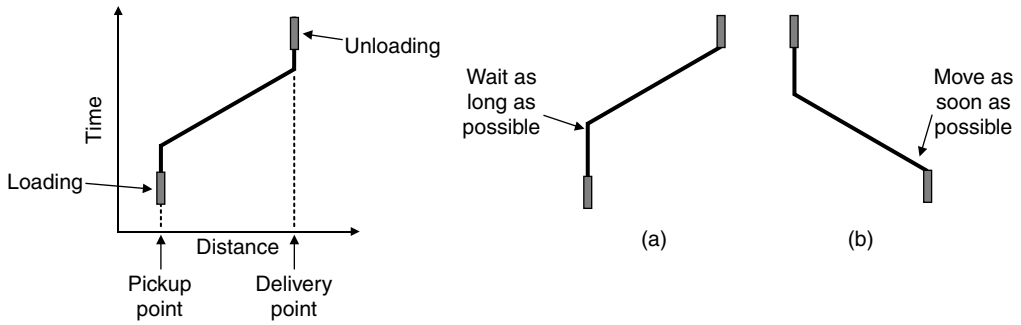
3. The Optimal Trajectory Problem

As noted above, we define a crane scheduling problem to consist of a number of *jobs*, each of which specifies several *tasks* to be performed consecutively. For example, a job may require that a crane pick up a ladle at one location, fill the ladle with molten metal at a second location, deliver the metal to a third location, and then return the ladle. Tasks may also involve maintenance and cleaning activities. The same crane must perform all the tasks in a job and must remain stationary at the appropriate location while processing each task.

The location and processing time for each task are given (Figure 1), as are release times and deadlines. Each crane is pre-assigned a sequence of jobs. Each job must finish before the next job assigned to that crane begins.

In this study we explicitly account only for the longitudinal movements of the crane along the track. We assume that the crane has time to make the necessary lateral and vertical

FIGURE 1. Sample space-time trajectory for one task (left) and extremal trajectory for the left crane (a) when the destination is to the right of the origin, and (b) when the destination is to the left of the origin (right).



Note. Left, the shaded vertical bars denote processing, which in this case consists of loading and unloading.

movements as it moves from one task location to another. This results in little loss of generality, because any additional time necessary for lateral or vertical motion can be built into the processing time for the task.

The problem data are:

R_j, D_j, P_j = release time, deadline, and processing time of task j;
L_j = processing location (stop) for task j;
$c(j)$ = crane assigned to task j;
$v, L_{\max}$ = maximum crane speed and track length; and
δ = minimum crane separation.

Note that we refer to the processing location of a task as a *stop*.

We suppose for generality that there are cranes $1, \ldots, m$, where crane 1 is the *left crane* and crane m the *right crane*, although we solve the problem only for $m = 2$. $T_{\max}$ is the length of the time horizon. The problem variables are:

$x_c(t)$ = position of crane c at time t;
y_{ct} = task being processed by crane c at time t (0 if none); and
τ_j = time at which task j starts processing.

Task j therefore finishes processing at time $\tau_j + P_j$. We assume that the tasks are indexed so that tasks assigned to a given crane are processed in order of increasing indices.

If we measure time in discrete intervals of length Δt, the basic problem with n tasks and m cranes may be stated

$$\min f(\tau)$$

$$\left.\begin{array}{ll} 0 \le x_c(t) \le L_{\max} & \text{(a)} \\ x_c(t) - v\Delta t \le x_c(t+\Delta t) \le x_c(t) + v\Delta t & \text{(b)} \\ y_{ct} > 0 \;\Rightarrow\; x_c(t) = L_{y_{ct}} & \text{(c)} \end{array}\right\} \quad \text{all } c, \text{ all } t \in \mathcal{T},$$

$$x_c(t) \le x_{c+1}(t) - \Delta, \;\; c = 1, \ldots, m-1, \quad \text{all } t \in \mathcal{T}, \qquad \text{(d)} \quad (1)$$

$$\left.\begin{array}{ll} R_j \le \tau_j \le D_j - P_j & \text{(e)} \\ y_{c(j)t} = j, \quad t = \tau_j, \ldots, \tau_j + P_j - \Delta t & \text{(f)} \end{array}\right\} \quad j = 1, \ldots, n,$$

$$\{c(j) = c(j'), \, j < j'\} \;\Rightarrow\; \tau_j < \tau_{j'}, \quad j, j' = 1, \ldots, n, \qquad \text{(g)}$$

$$x_{ct}, \tau_j \in \mathbb{R}, \quad y_{ct} \in \{0, \ldots, n\}, \quad \text{all } c, \text{ all } t \in \mathcal{T},$$

where $\mathcal{T} = \{0, \Delta t, \ldots, T_{\max}\}$. Constraint 1(a) requires that the cranes stay on the track, and (b) that their speed be within the maximum. Constraint 1(c) implies that a crane must be at the right location when it is processing a task. Constraint 1(d) makes sure the cranes do not interfere with each other. Constraint 1(e) enforces the time windows, and 1(f) ensures that processing continues for the required amount of time once it starts. Constraint 1(g) requires that the tasks assigned to a crane be processed in the right order. A number of additional constraints may be imposed in the dynamic programming model, as described in §5 below.

We assume that the objective $f(\tau)$ is a function of the task start times, because this is sufficient for practical application and allows us to prove the structural results below. Because the dynamic programming model requires separability, we assume $f(\tau)$ has the form $\sum_j f_j(\tau_j)$. Yet each $f_j(\tau_j)$ can be any function whatever, which provides a great deal of flexibility. In typical factory applications, we are interested in conforming to the production schedule as closely as possible. Thus we might define $f_j(\tau_j) = p_j(\tau_j - R_j)$, where $p_j(t)$ is any penalty function of the tardiness t we may wish to use, linear or nonlinear, continuous or discontinuous. If we wish to minimize makespan, we can let task n be a dummy task that is constrained to follow all others, and set $f_n(\tau_n) = \tau_n$ and $f_j(\tau_j) = 0$ for $j = 1, \ldots, n-1$.

4. Canonical Trajectories

Optimal control of the cranes is much easier to calculate when it is recognized that only certain trajectories need be considered, namely those we call canonical trajectories. We will show that when there are two cranes, some pair of canonical trajectories is optimal. This substantially reduces the number of state transitions that must be enumerated and makes a dynamic programming approach computationally practical for this problem.

Let a *processing schedule* for a given crane consist of the vector τ of task start times. We define the *extremal* trajectory with respect to τ for the left crane to be one that moves to the right as late as possible, and moves to the left as early as possible (Figure 1). The extremal trajectory for the right crane moves to the left as late as possible and to the right as early as possible.

More precisely, if $L_j \le L_{j+1}$, the extremal trajectory for the left crane for $t \in [\tau_j, \tau_{j+1}]$ is given by

$$\bar{x}_{1t} = \begin{cases} L_j & \text{for } t \in [\tau_j, T_j] \\ L_j + v(t - T_j) & \text{for } t \in [T_j, \tau_{j+1}], \end{cases}$$

where $T_j = \tau_{j+1} - (L_{j+1} - L_j)/v$. If $L_j > L_{j+1}$, it is given by

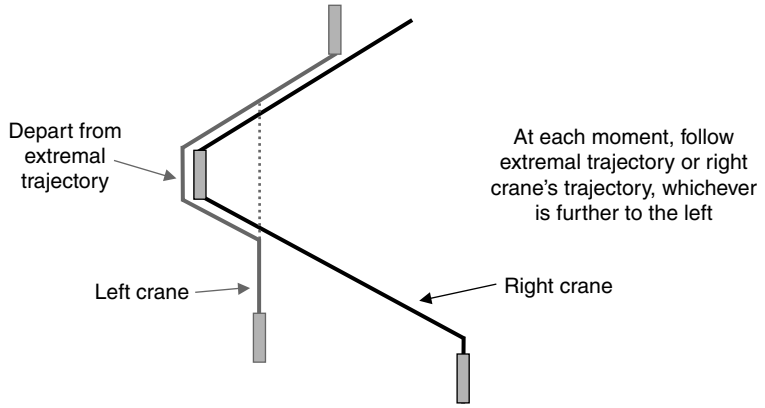
$$\bar{x}_{1t} = \begin{cases} L_j & \text{for } t \in [\tau_j, \tau_j + P_j] \\ L_j + v(t - \tau_j - P_j) & \text{for } t \in [\tau_j + P_j, T'_j] \\ L_{j+1} & \text{for } t \in [T'_j, \tau_{j+1}], \end{cases}$$

where $T'_j = \tau_j + P_j + (L_{j+1} - L_j)/v$.

A trajectory for the left crane is *canonical* with respect to the right crane if at each moment it is the leftmost of the left crane's extremal trajectory and the right crane's actual trajectory, allowing for separation Δ (Figure 2). More precisely, trajectory x'_1 is canonical for the left crane, with respect to trajectory x_2 for the right crane, if $x'_1(t) = \min\{\bar{x}_1(t), x_2(t) - \Delta\}$ at each time t. Similarly, trajectory $x'_2(t)$ is canonical for the right crane if $x'_2(t) = \max\{\bar{x}_2(t), x_1(t) + \Delta\}$. Finally, a pair of trajectories is canonical if the trajectories are canonical with respect to each other.

Theorem 1. *If the two-crane problem* (1) *has an optimal solution, then some optimal pair of trajectories is canonical.*

FIGURE 2. Canonical trajectory for the left crane (leftmost solid line).



Proof. The idea of the proof is to replace the left crane's optimal trajectory with a canonical trajectory with respect to the right crane's optimal trajectory. Then assign the right crane a canonical trajectory with respect to the left crane's new trajectory, and assign the left crane a canonical trajectory with respect to the right crane's new trajectory. At this point it is shown that the trajectories are canonical with respect to each other. We will see that these replacements can be made without changing the objective function value, which means the canonical trajectories are optimal, and the theorem follows.

Thus let $x^* = (x_1^*,\ x_2^*)$ be a pair of optimal trajectories for a two-crane problem. Let $\bar{x}_1$, $\bar{x}_2$ be extremal trajectories for the left and right cranes with respect to the processing schedules in the optimal trajectories. By definition, the extremal trajectories have the same processing schedules as the original trajectories.

Consider the canonical trajectory x_1' for the left crane with respect to x_2^*, which is given by $x_1'(t) = \min\{\bar{x}_1(t), x_2^*(t) - \Delta\}$. We claim that (x_1', x_2^*) is optimal. First note that $\bar{x}_1(t) = x_1^*(t)$ if the left crane is processing at time t, by definition of $\bar{x}_1$. Thus $\bar{x}_1(t) = x_1^*(t) \le x_2^*(t) - \Delta$ for all such t, because x_1^*, x_2^* is a feasible schedule. This implies that $x_1'(t) = x_1^*(t)$ at all times t when the left crane is processing. The left crane can therefore retain its original processing schedule. Because the objective function value does not change, (x_1', x_2^*) is optimal if it is feasible.

Furthermore, (x_1', x_2^*) is feasible because the cranes do not interfere with each other, and the speed of the left crane is never greater than v. The cranes do not interfere with each other because $x_1'(t) \le x_2^*(t) - \Delta$ for all t, from the defintion of $x_1'(t)$. To show that the speed of the left crane is never more than v it suffices to show that the average speed in the left-to-right direction between any pair of time points t_1, t_2 is never more than v, and similarly for the average speed in the right-to-left direction. The former is

$$\frac{x_1'(t_2) - x_1'(t_1)}{t_2 - t_1} = \frac{\min\{\bar{x}_1(t_2), x_2^*(t_2) - \Delta\} - \min\{\bar{x}_1(t_1), x_1^*(t_1) - \Delta\}}{t_2 - t_1}$$
$$\le \max\left\{ \frac{\bar{x}_1(t_2) - \bar{x}_1(t_1)}{t_2 - t_1},\ \frac{x_2^*(t_2) - x_2^*(t_1)}{t_2 - t_1} \right\} \le v$$

where the first inequality is due to the fact that

$$\min\{a, b\} - \min\{c, d\} \le \max\{a - c,\ b - d\}$$

for any a, b, c, d, and the second inequality due to the fact that $\bar{x}_1$ and x_2^* are feasible trajectories. The speed in the right-to-left direction is similarly bounded.

Now consider the canonical trajectory x_2' for the right crane with respect to x_1', given by $x_2'(t) = \max\{\bar{x}_2(t), x_1'(t) + \Delta\}$. It can be shown as above that the right crane can retain its processing schedule, and (x_1', x_2') is therefore optimal.

Finally, let x_1'' be the canonical trajectory for the left crane with respect to x_2', given by $x_1''(t) = \min\{\bar{x}_1(t), x_2'(t) - \Delta\}$. Again (x_1'', x_2') is optimal. Since x_1'' is canonical with respect to x_2', to prove the theorem it suffices to show that x_2' is canonical with respect to x_1''; that is, $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = x_2'(t)$ for all t. To show this we consider four cases for each time t.

Case 1. $\bar{x}_1(t) + \Delta \le \bar{x}_2(t)$. We first show that

$$(x_1''(t), x_2'(t)) = (\bar{x}_1(t), \bar{x}_2(t)). \tag{2}$$

Because $x_1'(t) = \min\{\bar{x}_1(t), x_2^*(t) - \Delta\}$, $x_1'(t)$ is equal to $\bar{x}_1(t)$ or $x_2^*(t) - \Delta$. In the latter case, we have

$$x_2'(t) = \max\{\bar{x}_2(t), x_1'(t) + \Delta\} = \max\{\bar{x}_2(t), x_2^*(t)\} = \bar{x}_2(t),$$

and

$$x_1''(t) = \min\{\bar{x}_1, x_2'(t) - \Delta\} = \min\{\bar{x}_1, \bar{x}_2(t) - \Delta\} = \bar{x}_1(t).$$

If, on the other hand, $x_1'(t) = \bar{x}_1(t)$, we have $x_2'(t) = \max\{\bar{x}_2(t), \bar{x}_1(t) + \Delta\} = \bar{x}_2(t)$ and again $x_1''(t) = \bar{x}_1(t)$. Now from (2) we have

$$\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), \bar{x}_1(t) + \Delta\} = \bar{x}_2(t) = x_2'(t)$$

as claimed.

The remaining cases suppose $\bar{x}_2(t) < \bar{x}_1(t) + \Delta$ and consider the situations in which $x_2^*(t)$ is less than or equal to $\bar{x}_2(t)$, between $\bar{x}_2(t)$ and $\bar{x}_1(t) + \Delta$, and greater than $\bar{x}_1(t) + \Delta$.

Case 2. $x_2^*(t) \le \bar{x}_2(t) < \bar{x}_1(t) + \Delta$. It can be checked that $(x_1''(t), x_2'(t)) = (\bar{x}_2(t) - \Delta, \bar{x}_2(t))$ and $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), \bar{x}_2(t)\} = \bar{x}_2(t) = x_2'(t)$, as claimed.

Case 3. $\bar{x}_2(t) < x_2^*(t) \le \bar{x}_1(t) + \Delta$. Here $(x_1''(t), x_2'(t)) = (x_2^*(t) - \Delta, x_2^*(t))$ and $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), x_2^*(t)\} = x_2^*(t) = x_2'(t)$.

Case 4. $\bar{x}_2(t) < \bar{x}_1(t) + \Delta < x_2^*(t)$. Here $(x_1''(t), x_2'(t)) = (\bar{x}_1(t), \bar{x}_1(t) + \Delta)$ and $\max\{\bar{x}_2(t), x_1''(t) + \Delta\} = \max\{\bar{x}_2(t), \bar{x}_1(t) + \Delta\} = \bar{x}_1(t) + \Delta = x_2'(t)$. This completes the proof.

The properties of canonical trajectories allow us to consider a very restricted subset of trajectories when computing the optimum.

Corollary 1. *If the two-crane problem has an optimal solution, then there is an optimal solution with the following characteristics:*

(a) *While not processing a task, the left (right) crane is never to the right (left) of both the previous and the next stop.*

(b) *While not processing a task, the left (right) crane is moving in a direction toward its next stop if it is to the right (left) of the previous or next stop.*

(c) *A crane never moves in the direction away from its next stop unless it is adjacent to the other crane at all times during such motion.*

(d) *While not processing a task, the left (right) crane can be stationary only if it is* (i) *at the previous or the next stop, whichever is further to the left (right), or* (ii) *adjacent to the other crane.*

Proof. (a) If crane 1 (the left crane) is to the right of both its previous and next stop at some time t, then $x_1(t) > \bar{x}_1(t)$. This is impossible in a canonical trajectory, in which $x_1(t) = \min\{\bar{x}_1(t), x_2(t) - \Delta\}$. The argument is similar for crane 2.

(b) Suppose crane 1 is to the right of its previous stop. Due to (a), it is not to the right of its next stop, which must therefore be to the right of the previous stop. We cannot have $x_1(t) > \bar{x}_1(t)$ as in (a), and we cannot have $x_1(t) < \bar{x}_1(t)$, since this means the crane cannot reach its next stop in time. So crane 1 is on its canonical trajectory, which means that it is moving toward its next stop. The argument is similar if crane is to the right of the next stop.

(c) From (a) and (b), at a given time t crane 1 can be moving in the direction opposite its next stop only if it is at or to the left of both the previous and next stops. This means that it will be to the left of both at time $t+\Delta t$, so that $x_1(t+\Delta t) < \bar{x}_1(t+\Delta t)$. But since

$$x_1(t+\Delta t) = \min\{\bar{x}_1(t+\Delta t), x_2(t+\Delta t) - \Delta\},$$

this means $x_1(t+\Delta t) = x_2(t+\Delta t) - \Delta$, and crane 1 is adjacent to the other crane. Since crane 1 is moving left between t and $t+\Delta t$, it must be adjacent to the other crane at time t as well.

(d) From (a) and (b), a stationary crane 1 must be at or to the left both the previous and the next stop. If it is at one of them, then (i) applies. If it is to the left of both, then $x_1(t) < \bar{x}_1(t)$, which again implies that $x_1(t) = x_2(t) - \Delta$, and (ii) holds.

5. Dynamic Programming Recursion

The optimal control problem for the cranes is not simply a matter of computing an optimal space-time trajectory. It is complicated by three factors: (a) each crane must perform tasks in a certain order; (b) each task must be performed at a certain location for a certain amount of time; and (c) the cranes must not interfere with each other. Dynamic programming has the flexibility to deal with these and other constraints while preserving optimality (up to the precision allowed by the space and time granularity). The drawback is a potentially exploding state space, but we will show how to keep it under control for problems of reasonable size. To simplify notation, we assume from here out that $\Delta t = 1$.

There are three state variables for each crane. One is the crane location x_{ct} as defined in model (1). The second is the task y_{ct} assigned to crane c at time t. This the task currently being processed, or if the crane is not currently processing, the next task to be processd by crane c. This differs from y_{ct} in the model (1) in that we no longer set $y_{ct} = 0$ when crane c is not processing. The third state variable is

$$u_{ct} = \begin{cases} \text{amount of time crane } c \text{ will have been processing at time } t+1 \\ (0 \text{ if the crane is neither processing nor starts processing at time } t). \end{cases}$$

In principle the recursion is straightforward, although a practical implementation requires careful management of state transitions and data structures. Let $x_t = (x_{1t}, x_{2t})$, and similarly for y_t and u_t. Also let $z_t = (x_t, y_t, u_t)$. It is convenient to use a forward recursion:

$$g_{t+1}(z_{t+1}) = \min_{z_t \in S^{-1}(z_{t+1})} \{h(t, y_t, u_t) + g_t(z_t)\}, \tag{3}$$

where $g_t(z_t)$ is the cost of an optimal trajectory between the initial state and state z_t at time t, $h(t, y_t, u_t)$ is the cost incurred at time t, and $S^{-1}(z_{t+1})$ is the set of states at time t from which the system can move to state z_{t+1} at time $t+1$.

Because the objective function is $f(\tau) = \sum_j f_j(\tau_j)$ as specified earlier, we incur cost $f_j(t)$ when the crane c assigned task j starts task j at time t (i.e., $y_{ct} = j$ and $u_{ct} = 1$). Thus $h(t, y_t, u_t) = \sum_c h_c(t, y_t, u_t)$, where

$$h_c(t, y_t, u_t) = \begin{cases} f_{y_{ct}}(t) & \text{if } u_{ct} = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The boundary condition is $g_0(z_0) = 0$, where z_0 is the initial state. The optimal cost is $g_{T_{\max}}(z_{T_{\max}})$, where $z_{T_{\max}}$ is the desired terminal state.

The possible state transitions are restricted by the rules in Corollary 1. This allows us to exclude states from $S^{-1}(z_{t+1})$ and therefore reduce computation. Rule (a), for example, prevents the left crane from moving to the right if this would put it to the right of both the current and the next stop. Let y' be the stop that precedes stop $y_{1,\,t+1}$ in the list of

stops assigned to the left crane (crane 1). If $x_{1,t+1} = \max\{L_{y_{1,t+1}}, L_{y'}\}$ in state z_{t+1}, then x_{1t} cannot be equal to $x_{1,t+1}+1$ in any state belonging to $S^{-1}(z_{t+1})$, because this would mean that the left crane was to the right of both its previous and next stop. The reasoning is similar for the right crane.

Rule (b) of the corollary requires the left crane to move toward its next stop if it is to the right of the previous or next stop. This means that if $x_{1,t+1} \geq L_{y'}$ and $x_{1,t+1} < L_{y_{1,t+1}}$, then x_{1t} cannot be equal to $x_{1,t+1}+1$ in any state belonging to $S^{-1}(z_{t+1})$, because this would mean that the crane failed to move toward its next stop even though it was to the right of its previous stop. Similarly, if $x_{1,t+1} \leq L_{y'}$ and $x_{1,t+1} > L_{y_{1,t+1}}$, then x_{1t} cannot be equal to $x_{1,t+1}-1$ in any state belonging to $S^{-1}(z_{t+1})$, because this would mean that the crane failed to move toward its next stop even though it was to the right of its next stop. Rules (c) and (d) of the corollary are similarly implemented.

By a suitable definition of $S^{-1}(z_{t+1})$, we can impose any additional constraint that can be defined in terms of the current state and that is consistent with a canonical trajectory. For example, we can require the crane c processing task j to start moving to the next task j' as soon as processing is finished. This is possible because the state at time t tells us whether crane c is processing task j ($y_{ct} = j$) and will be finished at time $t+1$ ($u_{ct} = P_j$). The set $S^{-1}(z_{t+1})$ is again defined by formulating the constraint in reverse: if crane c is assigned task j' at time $t+1$ ($y_{c,t+1} = j'$) and is still in the same location as task j ($x_{c,t+1} = L_j$), then there is no feasible predecessor for the current state ($S^{-1}(z_{t+1}) = \varnothing$). Similarly, we can prohibit crane c from yielding (moving away from its next stop) after it finishes processing task j. A variety of precedence constraints can also be implemented. For example, we can require that the right crane start processing task j' only after the left crane finishes task j. At any given time t, we can determine whether the left crane has finished task j by checking whether $y_{1t} > j$, and if so we allow the right crane to start task j'. Finally, we can impose bounds on the processing time rather than specify it exactly, because state variable u_{ct} indicates how long the current task has been processed so far.

For each state z_{t+1} the recursion (3) computes the minimum $g_{t+1}(z_{t+1})$ and the state $z_t = s_{t+1}^{-1}(z_{t+1})$ that achieves the minimum. Thus $s_{t+1}^{-1}(z_{t+1})$ points to the state that would precede z_{t+1} in the optimal trajectory if z_{t+1} were in the optimal trajectory. For a basic recursion, the cost table $g_{t+1}(\cdot)$ is stored in memory until $g_{t+2}(\cdot)$ is computed, and then released (this is modified in the next section). Thus only two consecutive cost tables need be stored in memory at any one time. The table $s_{t+1}^{-1}(\cdot)$ of pointers is stored offline. Then if z_T is the final state, we can retrace the optimal solution in reverse order by reading the tables $s_{t+1}^{-1}(\cdot)$ into memory one at a time and setting $z_t = s_{t+1}^{-1}(z_{t+1})$ for $t = N-1, N-2, \ldots, 0$.

6. Reduction of the State Space

We can substantially reduce the size of the state space if we observe that in practical problems, the cranes spend more time processing than moving. The typical processing time for a state ranges from two to five minutes (sometimes much longer), while the typical transit time to the next location is well under a minute. Furthermore, the state variables representing location and task assignment (x_{ct} and y_{ct}) cannot change while the crane is processing.

These facts suggests that the processing time state variable u_{ct} should be replaced by an *interval* $U_{ct} = [u_{ct}^{lo}, u_{ct}^{hi}] = \{u_{ct}^{lo}, u_{ct}^{lo}+1, \ldots, u_{ct}^{hi}\}$ of consecutive processing times, where $u_{ct}^{lo} \geq 0$ and $u_{ct}^{hi} \leq P_{y_{ct}}$. A single "state" $(x_t, y_t, U_{ct}) = (x_t, y_t, (U_{1t}, U_{2t}))$ now represents a set of states, namely the Cartesian product

$$\{(x_t, y_t, (i,j)) \mid i \in U_{1t},\, j \in U_{2t}\}.$$

The possible state transitions for either crane c are shown in Table 1. The transitions in the table are feasible only if they satisfy other constraints in the problem, including those

TABLE 1. Possible state transitions for crane c using an interval-valued state variable for processing time.

State at time t	State at time $t+1$
1. $(x_{ct}, y_{ct}, [0,0])$	$(x', y_{ct}, [0,0])^{1}$ or $(x', y_{ct}, [0,1])^{1,2}$ or $(x', y_{ct}, [1,1])^{1,2,3}$
2. $(x_{ct}, y_{ct}, [0,u_2])^{4}$	$(x_{ct}, y_{ct}, [0,u_2+1])$ or $(x_{ct}, y_{ct}, [1,u_2+1])^{2,4}$
3. $(x_{ct}, y_{ct}, [0,P_{y_{ct}}])$	$(x_{ct}, y_{ct}, [0,P_{y_{ct}}])$ or $(x_{ct}, y_{ct}, [1,P_{y_{ct}}])^{3}$ or $(x_{ct}, y', [0,0])^{5}$ or $(x_{ct}, y', [0,1])^{2,5}$ or $(x_{ct}, y', [1,1])^{2,3,5}$
4. $(x_{ct}, y_{ct}, [u_1,u_2])^{4,6}$	$(x_{ct}, y_{ct}, [u_1+1,u_2+1])$
5. $(x_{ct}, y_{ct}, [u_1,P_{y_{ct}}])^{6}$	$(x_{ct}, y_{ct}, [u_1+1,P_{y_{ct}}])$ or $(x_{ct}, y', [0,0])^{5}$ or $(x_{ct}, y', [0,1])^{2,5}$ or $(x_{ct}, y', [1,1])^{2,3,5}$

[1]The next location x' is $x_{ct}-1$, x_{ct}, or $x_{ct}+1$.
[2]This transition is possible only if task y_{ct} processes at location x'.
[3]This transition is possible only if task y_{ct} can start no later than time $t+1$.
[4]Here $0 < u_2 < P_{y_{ct}}$.
[5]Task y' is the task that follows task y_{ct} on crane c.
[6]Here $u_1 > 0$.

based on time windows, the physical length of the track, and interactions with the other crane. The transitions can be explained, line by line, as follows:

(1) Because the processing time interval is the singleton $[0,0]$, the crane can be in motion and can in particular move to either adjacent location. When it arrives at the next location, the currently assigned task can start processing if the crane is in the correct position, in which case the state interval is $U_{ct} = [0,1]$ to represent two possible states: one in which the task does not start processing at time $t+1$, and one in which it does (the interval is $[1,1]$ if the deadline forces the task to start processing at $t+1$). If the crane is in the wrong location for the task, the state remains $[0,0]$.

(2) None of the states in the interval $[0,u_2]$ allow processing to finish at time $t+1$. So all of the processing time states advance by one—except possibly the zero state, in which processing has not yet started and can be delayed yet again if the deadline permits it.

(3) The last state in the interval $[0,P_{y_{ct}}]$ allows processing to finish at time $t+1$. This state splits off from the interval and assumes one of the processing state intervals in line 1. The other states evolve as in line 2.

(4) Because the task is underway in all states, all processing times advance by one.

(5) This is similar to line 3 except that there is no zero state.

There is no need to store a pointer $s_{t+1}^{-1}(x_t, y_t, (i,j))$ for every state $(x_t, y_t, (i,j))$ in (x_t, y_t, U_t). This is because when $u_{ct} \geq 2$, the state of crane c preceding (x_{ct}, y_{ct}, u_{ct}) must be $(x_{ct}, y_{ct}, u_{ct}-1)$. Thus we store $s_{t+1}^{-1}(x_t, y_t, (i,j))$ only when $i \leq 1$ or $j \leq 1$.

However, we must store the cost $g_{t+1}(x_t, y_t, (i,j))$ for every (i,j), because it is potentially different for every (i,j). Fortunately, it is not necessary to update this entire table at each time period, because most of the costs evolve in a predictable fashion. If $i,j \geq 2$, then

$$g_{t+1}\big(x_y, y_t, (i,j)\big) = g_t\big(x_t, y_t, (i-1,j-1)\big).$$

So for each pair of tasks (y,y') we maintain a two-dimensional circular queue $Q_{yy'}(\cdot,\cdot)$ in which the cost

$$g_{t+1}((L_y, L_{y'}), (y,y'), (i,j)) \tag{4}$$

for $i,j \geq 2$ is stored at location

$$Q_{yy'}\big((t+i-2) \bmod M, (t+j-2) \bmod M\big),$$

where M is the size of the array $Q_{yy'}(\cdot,\cdot)$ (i.e., the longest possible processing time). In each period we insert the cost (4) into Q only for pairs (i,j) in which $i=2$ or $j=2$; the costs for other pairs with $i,j \geq 2$ were computed in previous periods. Thus only one

row and one column of the Q array are altered in each time period, which substantially reduces computation time. When $i \leq 1$ or $j \leq 1$, the cost (4) is stored as a table entry $g_{t+1}(x_t, y_t, (i, j))$ that is updated at every time period, as with pointers.

The array $Q_{yy'}(\cdot, \cdot)$ is created when the state $((L_y, L_{y'}), (y, y'), (i, j))$ is first encountered with $i, j \geq 2$. The array is kept in memory over multiple periods until it is no longer updated, at which time it is deleted.

7. Experimental Results

We report computational tests on a representative problem that is based on an actual industry scheduling scenario. There are 60 jobs, of which four jobs consist of one task, eleven consist of two tasks, and 45 consist of three tasks, for a total of 161 tasks. We obtain smaller instances by scheduling only some of the jobs, namely the first ten (in order of release time), the first twenty, and so forth. Results on other problems we have examined are similar. In particular, we found that the computation time depends primarily on the width of the time windows, regardless of the problem solved.

Release times were obtained from the production schedule, but no deadlines were given. Because of the sensitivity of computation time to time windows, we initially set the deadline of each job to be 40 minutes after each release time, with the expectation that these may have to be relaxed to obtain a feasible solution.

We divided the 108.5-meter track into ten equal segments, so that each distance unit represents 10.85 meters. Each crane can traverse the length of the track in about one minute. Because we want the crane to move one distance unit for each time unit, we set the time unit at six seconds. The 60-job schedule requires about four hours to complete, which means that the dynamic programming procedure has about $T_{\max} = 2{,}400$ time stages.

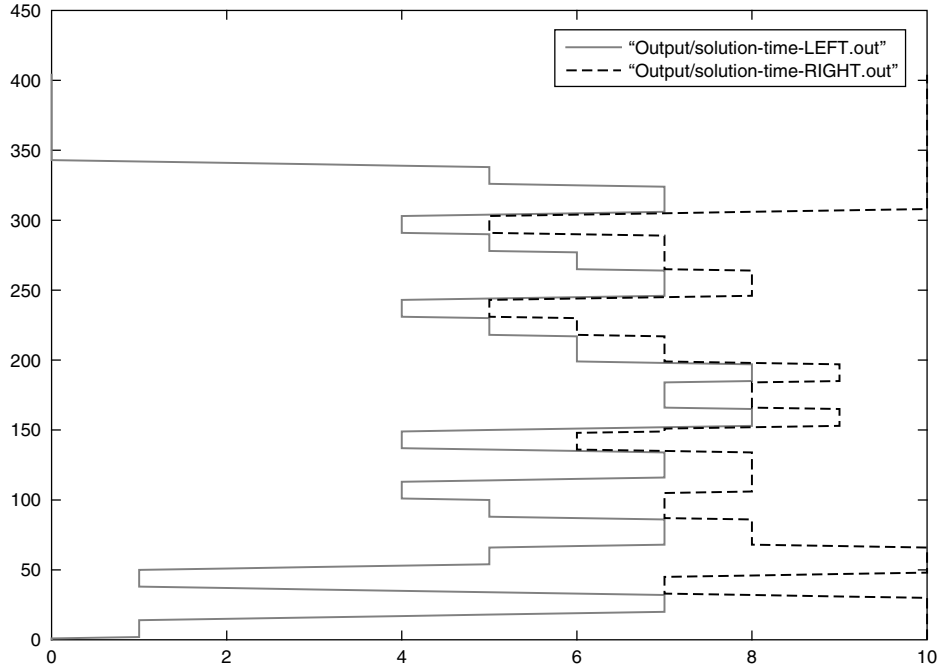
Table 2 shows computation times obtained on a desktop PC running Windows XP with a Pentium D processor 820 (2.8 GHz). The assignment and sequencing of jobs used in each instance is the best one that was obtained by a heuristic procedure. Feasible solutions were found for all the instances except the full 60-job problem. To obtain a feasible solution of this problem, we enlarged the time windows from 40 to 95 minutes by postponing the deadlines. This illustrates the combinatorial nature of the problem, because the addition of only ten jobs created new bottlenecks that delayed at least one job nearly 95 minutes beyond its release time. Wider time windows result in a larger state space and thus greater computation time. Nonetheless, the 60-job problem with 95-minute windows was solved in well under a minute.

The optimal trajectories for two selected instances appear in Figures 3 and 4. The horizontal axis represents distance along the track in 10.85-meter units. The vertical axis represents time in 6-second units. Thus the schedule for the 30-job problem spans about 1,350 time units, or 135 minutes. The space-time trajectory of the left crane appears as a solid line, and as a dashed line for the right crane. The left crane begins and ends at the leftmost position,

Table 2. Computational results for subsets of the 60-job problem.

Jobs	Time window (mins)	Computation time (sec)
10	40	6.8
20	40	7.6
30	40	15.8
40	40	16.7
50	40	18.8
60	95	48.1

FIGURE 3. Optimal solution of the 10-job instance.



and analogously for the right crane. Note that the cranes are at rest most of the time. The trajectories are canonical trajectories as defined above, which ensures a certain consistency in the way the two cranes interact.

The number of states was always less than 500 for the 10-job instance, less than 1,000 for 30 jobs, and less than 2,000 for 60 jobs—even though the theoretical maximum is astronomical. Figure 5 tracks the evolution of state space size over time for the 60-job problem.

FIGURE 4. Optimal solution of the 30-job instance.

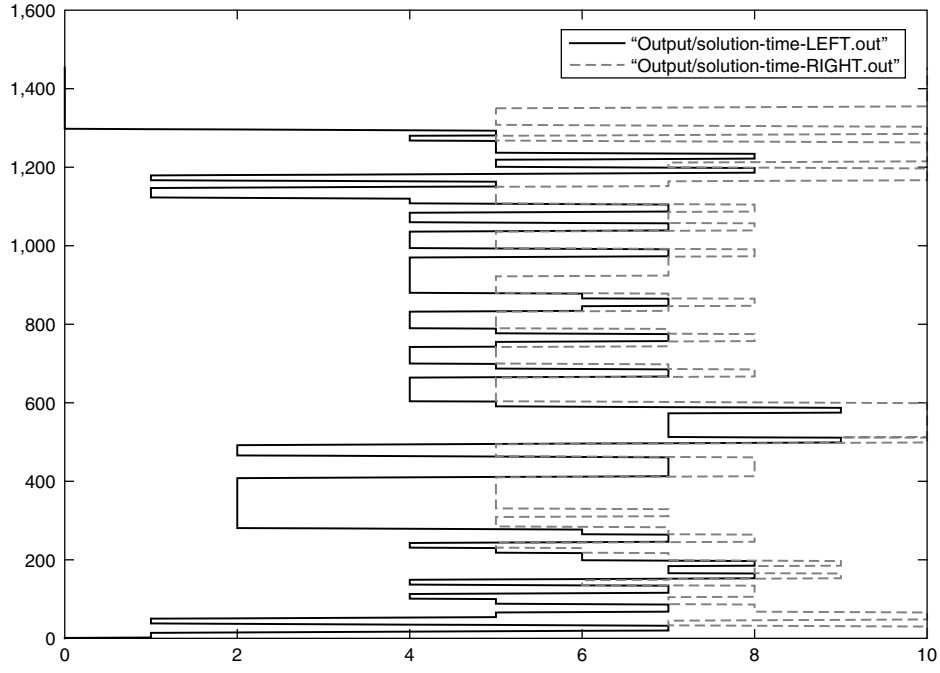
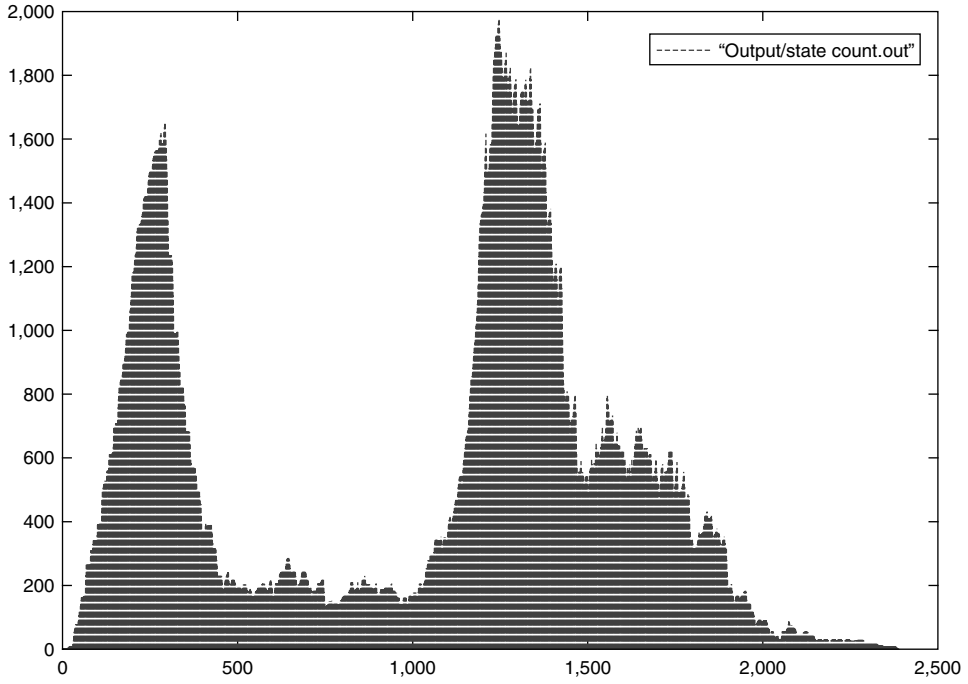


FIGURE 5. Evolution of the state space size for the 60-job instance.



The horizontal axis corresponds to time stages, which again are separated by six seconds. The vertical axis is the number of states at each time stage.

Typically, only a few time windows must be wide to allow a feasible solution, because only a few jobs must be delayed. Yet it is difficult or impossible to predict which are the critical jobs. It is therefore necessary to be able to solve problems in which all of the time windows are wide, perhaps on the order of 90 minutes as in the 60-job instance. It was to accommodate wide time windows that we developed the state space reduction techniques of §6.

Table 3 reveals the critical importance of these techniques. For each of the three problem instances, the table shows the average time and state space size required to compute the optimal trajectories for ten different job assignments and sequencings. Without the state space reduction technique, the dynamic programming algorithm could scale up to only 30 jobs, and even then only for narrow time windows. The time windows were reduced to make the problem tractable without state space reduction, while still maintaining feasibility. State space reduction cuts the peak number of states, and therefore the memory requirements, by a factor of 20 or more. It reduces the computation time by a factor of ten. The advantage is doubtless even greater for larger instances.

TABLE 3. Effect of state space reduction on state space size and computation time.

		Average number of states		Peak number of states		Average time (sec)	
Jobs	Time window (min)	Before	After	Before	After	Before	After
10	25	3,224	139	9,477	465	15.8	2.0
20	35	3,200	144	22,204	927	82.6	8.6
30	35	3,204	216	22,204	940	143.8	15.0

Notes. Each instance is solved for 10 different jobs assignments and sequencings. "Before" and "After" refer to results before and after state space reduction, respectively.

8. Conclusions and Future Research

We presented a specialized dynamic programming algorithm that computes optimal space-time trajectories for two interacting factory cranes. The state space is economically represented in such a way that medium-sized problems can be solved to optimality. The technique is useful both for solving a significant number of practical problems and as a benchmarking and calibration tool for heuristic methods that solve larger problems. The dynamic programming model accommodates a wide variety of constraints that often arise in this type of problem.

We also proved structural theorems to show that only certain types of trajectories need be considered to obtain an optimal solution. This not only accelerates solution of the problem but simplifies the operation of the cranes by restricting their movements to certain patterns.

An obvious direction for future research is to attempt to generalize the structural results to three or more cranes. Another useful research program would be a systematic empirical comparison of heuristic methods with the exact algorithm described here to determine how best to design and tune a heuristic algorithm.

References

[1] R. Armstrong, L. Lei, and S. Gu. A bounding scheme for deriving the minimal cycle time of a single-transporter N-stage process with time-window constraints. *European Journal of Operational Research* 78:130–140, 1994.

[2] A. Che and C. Chu. Single-track multi-hoist scheduling problem: A collision-free resolution based on a branch-and-bound approach. *International Journal of Production Research* 42: 2435–2456, 2004.

[3] C. F. Daganzo. The crane scheduling problem. *Transportation Research Part B* 23:159–175, 1989.

[4] K. H. Kim and Y.-M. Park. A crane scheduling method for port container terminals. *European Journal of Operational Research* 156:752–768, 2004.

[5] L. Lei and T. J. Wang. A proof: The cyclic hoist scheduling problem is NP-complete. Working paper, Rutgers University, New Brunswick, NJ, 1989.

[6] L. Lei and T. J. Wang. The minimum common-cycle algorithm for cycle scheduling of two material handling hoists with time window constraints. *Management Science* 37:1629–1639, 1991.

[7] L. Lei, R. Armstrong, and S. Gu. Minimizing the fleet size with dependent time-window and single-track constraints. *Operations Research Letters* 14:91–98, 1993.

[8] J. Leung and E. Levner. An efficient algorithm for multi-hoist cyclic scheduling with fixed processing times. *Operations Research Letters* 34:465–472, 2006.

[9] J. Leung and G. Zhang. Optimal cyclic scheduling for printed circuit board production lines with multiple hoists and general processing sequence. *IEEE Transactions on Robotics and Automation* 19:480–484, 2003.

[10] J. M. Y. Leung, G. Zhang, X. Yang, R. Mak, and K. Lam. Optimal cyclic multi-hoist scheduling: A mixed integer programming approach. *Operations Research* 52:965–976, 2004.

[11] R. W. Lieberman and I. B. Turksen. Two operation crane scheduling problems. *IIE Transactions* 14:147–155, 1982.

[12] J. Liu, Y. Jiang, and Z. Zhou. Cyclic scheduling of a single hoist in extended electroplating lines: A comprehensive integer programming solution. *IIE Transactions* 34:905–914, 2002.

[13] M.-A. Manier and C. Bloch. A classification for hoist schedling problems. *International Journal of Flexible Manufacturing Systems* 15:37–55, 2003.

[14] M.-A. Manier, C. Varnier, and P. Baptiste. Constraint-base model for the cyclic multi-hoists scheduling problem. *Production Planning and Control* 11:244–257, 2000.

[15] L. Mocchia, J.-F. Cordeau, M. Gaudioso, and G. Laporte. A branch-and-cut algorithm for the quay crane scheduling problem in a container terminal. *Naval Research Logistics* 53:45–59, 2005.

[16] W. C. Ng. Crane scheduling in container yards with inter-crane interference. *European Journal of Operational Research* 164:64–78, 2005.

[17] L. W. Phillips and P. S. Unger. Mathematical programming solution of a hoist scheduling problem. *AIIE Transactions* 8:219–321, 1976.

[18] D. Riera and N. Yorke-Smith. An improved hybrid model for the generic hoist scheduling problem. *Annals of Operations Research* 115:173–191, 2002.

[19] R. Rodošek and M. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. M. Maher and J.-F. Puget, eds. *Principle and Practice of Constraint Programming* (*CP* 1998), Vol. 1520. Springer, Berlin, 385–399, 1998.

[20] C. Varnier, A. Bachelu, and P. Baptiste. Resolution of the cyclic multi-hoists scheduling problem with overlapping partitions. *INFOR* 35:309–324, 1997.

[21] G. Yang, D. P. Ju, W. M. Zheng, and K. Lam. Solving multiple hoist scheduling problems by use of simulated annealing. *Transportation Research Part B* 36:537–555, 2001.

[22] C. Zhang, Y.-W. Wan, J. Liu, and R. J. Linn. Dynamic crane deployment in container storage yards. *Ruan Jian Xue Bao* (*Journal of Software*) 12:11–17, 2002.

[23] Z. Zhou and L. Li. A solution for cyclic scheduling of multi-hoists without overlapping. *Annals of Operations Research* 168:5–21, 2009.