THE BCS ALGORITHM: OPTIMIZING CRANE SCHEDULES ON MULTIPLE BAYS IN CONJUNCTION WITH CONTINUOUS TIME SIMULATION

by

JAMES STRIEBY

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Industrial & Manufacturing Systems Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2012

Approved by:

Major Professor
Dr. Todd Easton

Abstract

This thesis introduces the Bay Crane Scheduling (BCS) problem and related BCS algorithm. The purpose of this algorithm is to optimize the assignment of jobs to overhead cranes as well as the sequence in which each crane performs its assigned jobs. This problem is unique from other Overhead Crane Scheduling (OCS) problems through its increased complexity. Up until now, OCS problems involve a set number of cranes operating in a single common area, referred to as a bay, and are unable to pass over each other. The BCS problem involves a varying number of active cranes operating in multiple bays. Each crane is allowed to move from one bay to the next, through specific locations called bridges. This is crucial to completing certain "special" jobs that require two cranes operating in unison to transport an item.

The BCS algorithm employs two continuous time simulations in conjunction with an initial job-assignment algorithm and a Simulated Annealing (SA) improvement heuristic in order to minimize the non-productive crane time, while avoiding overloading any crane. To the extent of the author's knowledge, this is the first time a continuous time simulation has been used to model an OC system.

The BCS algorithm was originally developed for a large manufacturing facility, and when it was tested against the facility's current scheduling methods, it shows a 20% improvement in the overall active crane time required to complete equivalent set of jobs. This improved efficiency is crucial to the manufacturing facility being able to increase its production rate without the addition of new cranes. In addition, BCS is statistically shown to be superior to the current strategy. The results from BCS are substantial and practitioners are encouraged to utilize BCS's methodologies to improve other overhead crane systems.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

Scheduling is the decision process involved in assigning resources to perform specific tasks. The goal of this process is to make these assignments in such a way that some user-defined objective is optimized. The concept of scheduling has been in existence for as long as humans. In fact, the rise of Homo sapiens and fall of Neanderthals is partially attributed to the ability of homo-sapiens to plan ahead through the scheduling of hunting parties (Coolidge and Wynn, 2009). By scheduling a specific time to hunt, these early humans were essentially allocating all of their hunting resources (the actual hunters) to the same task, greatly increasing their chances of success. Conversely, Neanderthals, lacking any significant capacity for memory, were unable to consistently hunt in groups, making them far less successful.

This theme of one group becoming more successful than the other due to its ability to plan ahead and more efficiently schedule its resources, persists throughout history and will continue into the foreseeable future. This is what drives competing businesses in today's world to invest time and money into the study of scheduling theory, and is the main reason for this thesis.

This thesis presents a novel approach to optimizing the assignment of specific tasks to an overhead crane system servicing a large manufacturing facility located in central Kansas. This problem is referred to as the Bay-Crane Scheduling (BCS) problem. In order to properly convey the complexity involved in the BCS problem, a background on the operational specifics of this crane system is presented in the following section. The rest of this chapter then presents the specific reasons for this research and highlights the innovative techniques applied to solve the BCS problem.

1.1 Overhead Crane Basics

The manufacturing facility uses overhead cranes to move oversize items from job-shop to job-shop throughout the assembly process. On average, there are anywhere from 4-7 cranes in operation during a typical 8-hour shift, each crane requires a three-man crew in order to safely function. One of these employees sits up in the cab of the crane, and is referred to as the crane operator. The other two members of the crane-crew are down on the factory floor, helping to properly load and unload each item, as well as guide the item through floor traffic as it is being transported. The cab itself can move only horizontally in what is referred to as the *x*-direction, while the track the cab sets on can slide only in the *y*-direction. Figure 1 illustrates the basic setup of an overhead crane.

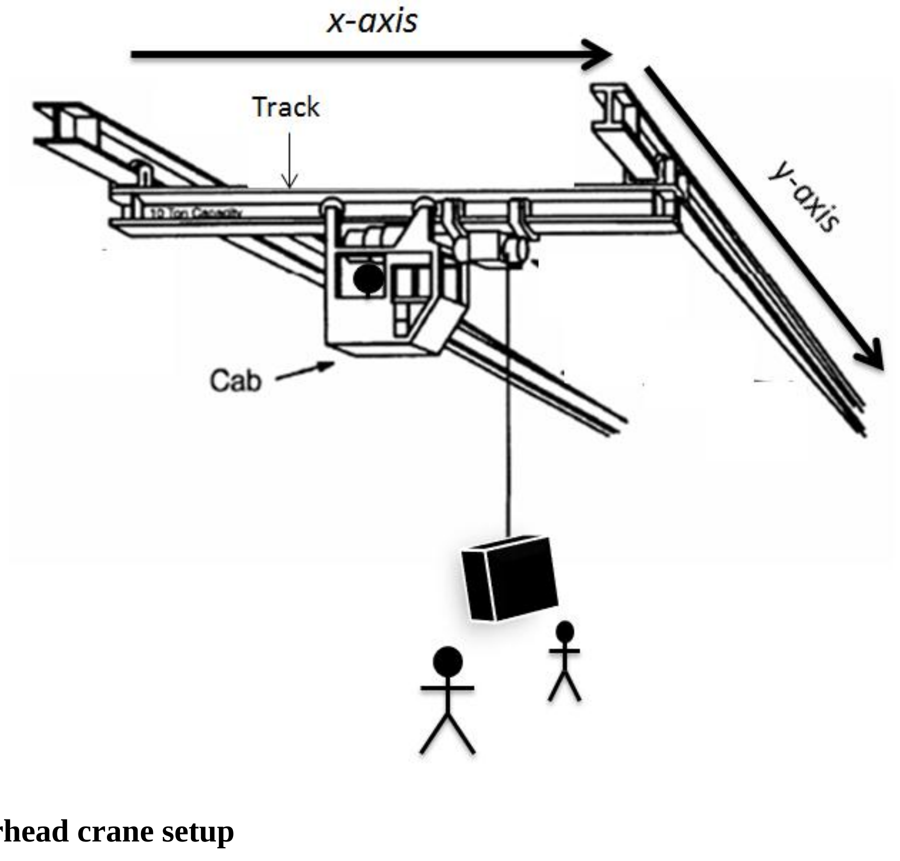


Figure 1 - Overhead crane setup

The main area in which a crane operates is referred to as a bay. Clearly, adjacent cranes operating in the same bay are physically unable to pass each other in the *y*-direction. Whenever a crane is attempting to move to a location currently on the opposite side (in the *y*-direction) of an adjacent crane, this adjacent crane must yield and move out of the way of the other. This basic, single-bay problem has recently been studied in several manufacturing facilities (Ge et al 2011, Goodwin et al 2009, Lieberman and Turksen 2007).

BCS is more complex than previous work on OC systems. This comes from the fact that jobs are attempting to be scheduled to multiple cranes operating in three adjacent bays. Furthermore, a cab can access an adjacent bay by sliding over to a track located in the adjacent bay. These bay transitions occur at portals referred to as bridges. The need for a cab to switch bays arises in cases where the scheduled job's pick-up and drop-off locations are in two different bays. In addition, BCS entails large jobs that require two cranes to transport the object. An overhead diagram of the three-bay facility is displayed in Figure 2.

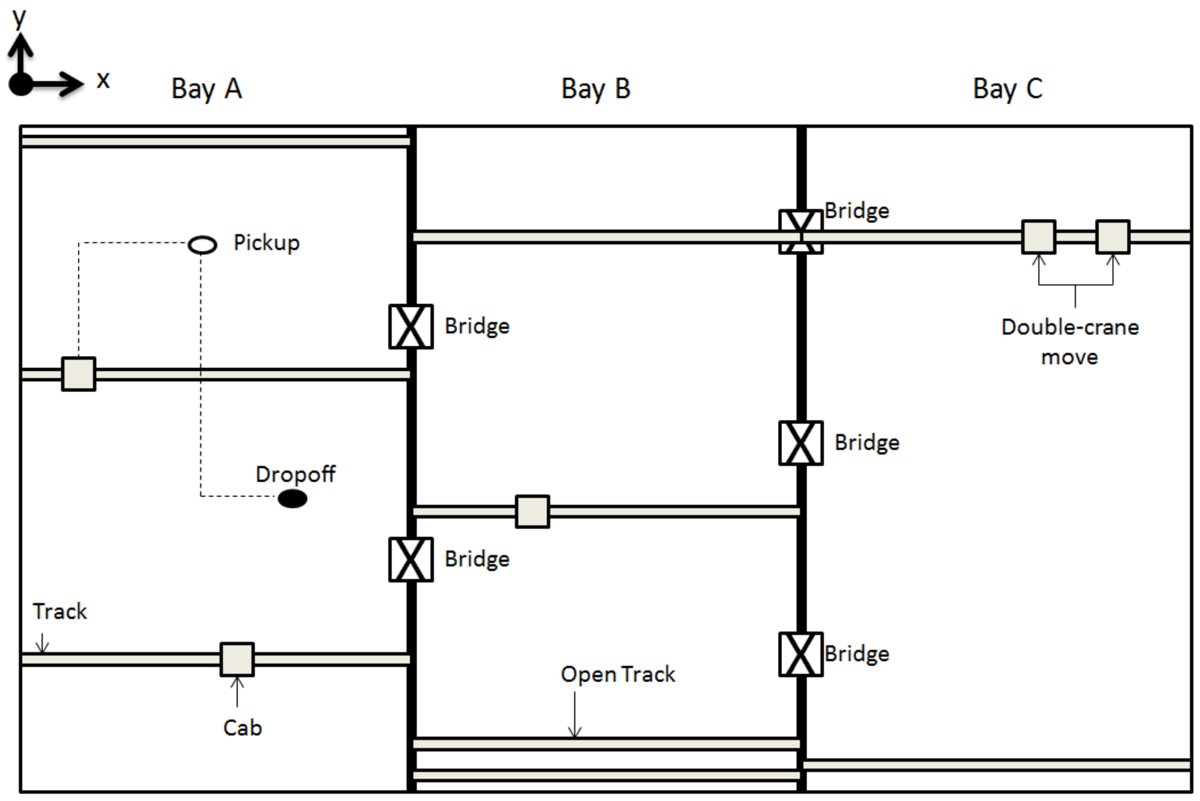


Figure 2 - Overhead diagram of multi-bay facility

Notice there are several tracks in each bay without cabs. All tracks can be remotely operated by a crane dispatcher, the employee in charge of assigning jobs to cranes. Whenever a cab needs to switch bays, the track it is currently on must line up with a track in the destination bay at one of these bridge locations. To the author's knowledge, this multi-bay scenario has never been previously researched, nor has moves requiring multiple cranes.

1.2 Research Motivation

This highly complex BCS problem is seen as an opportunity critical to improving the efficiency and production capability of the manufacturing facility. Currently the entire scheduling task falls on a single crane dispatcher. Whenever an item is ready for transport from one job-shop to the next, an employee within the job-shop currently housing the item puts a request in to the facility's computer system. This request comes up on the crane dispatcher's computer screen, who then decides to which crane the job is assigned, and in what sequence position. The sequence position represents the order in which a specific crane completes its assigned jobs. The dispatcher's goal is to assign incoming requests to the crane currently located nearest the pick-up location. Once an item is picked up by a crane, the dispatcher must also coordinate the drop off portion of the job, by calling down to the job-shop set to receive the item and making sure the crew is prepared for the delivery. This constant coordination of each job, coupled with the fact that there are typically 5-7 cranes simultaneously operating, causes the crane dispatcher to always be overburdened. Furthermore, the particular order in which a crane completes its jobs follows a basic first-in-first-out (FIFO) strategy.

The manufacturing facility previously made several attempts to simulate its crane system using their in-house simulation team. All of these attempts proved to be unsuccessful, as the team became bogged down in all of the complexities involved in the problem, quickly making it no longer cost-effective for the simulation team to continue its work. Eventually, a representative of the manufacturer approached the Kansas State University Industrial and Manufacturing Systems Engineering Department, seeking a professor-and-student pairing capable of creating a useful simulation model and making recommendations for improving the job-scheduling process.

At the same time the manufacturer approached the Industrial and Manufacturing Systems Engineering Department, I was seeking a research topic for my Master’s thesis. It was my desire to find an applied problem in the area of Operations Research, specifically optimization, in which I could contribute to the development of a new algorithm. My desired research topic characteristics and the manufacturer’s problem requirements matched perfectly.

The motivation of this research is twofold. First, it seeks to help escalate the production rate of the manufacturer without the addition of new crane crews, while alleviating the scheduling burden on the crane dispatcher. Second, the research strives to provide optimization techniques that are easily extendable by practitioners to other manufacturing facilities with similar overhead crane systems.

1.3 Research Contributions

This research’s primary contribution is the development of the BCS algorithm, the first of its kind to the author’s knowledge, developed for overhead crane systems operating within multiple bays. The BCS algorithm contains an initial assignment-and-sequencing heuristic employing the use of a continuous time simulation in order to “look ahead” and identify potential crane conflicts that may result in delays. After the completion of the initial heuristic, the BCS algorithm continues with the use of a second continuous time simulation in order to produce the estimated start and end times for each job, and to keep track of the total travel time between jobs and delay time experienced by each crane. Due to the quick computation time of this continuous time simulation, on average taking less than one-hundredth of a second, the last step of the BCS algorithm utilizes a Simulated Annealing heuristic to further optimize the sequence in which each crane performs its assigned jobs.

In order to assess the performance of the BCS algorithm, the current scheduling strategy employed in the manufacturing facility was also simulated. Overall, the BCS algorithm showed great improvements in the efficiency of the crane schedules. On average, the BCS algorithm shows a 20% reduction in the amount of crane time needed to complete the same set of jobs as required by the current strategy. This roughly equates to saving a crane, which requires three employees. Thus, the same jobs can be completed in the same amount of time with one less crane in operation. Furthermore, statistical analysis demonstrates that BCS is statistically superior to the current manufacturer's strategy.

The entire simulation and BCS algorithm was written in c-language in order to provide a common template from which other manufacturing facilities can begin the process of tailor-fitting the algorithm to their specific facilities by adjusting certain parameters within the algorithm. Furthermore, the continuous time simulations used within the BCS algorithm can easily be applied to any OC system, including the more common systems operating in a single bay.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 provides all of the background research involved in the development of the BCS algorithm, beginning with common optimization techniques. These techniques are broken down into exact optimization methods and heuristics, which are in turn broken down into two classes of heuristics, neighborhood and solution-based. Section 2.3 discusses the use of simulation in modeling complex problems, highlighting the two classes of simulation models, discrete and continuous. The final section of chapter 2 provides classic logistics optimization problems, each of which contains aspects involved in the BCS problem.

Chapter 3 presents the actual BCS algorithm, laying out the required inputs, basic steps, and overall structure in the first section. Section 3.2 then discusses possible objective functions for the algorithm culminating in the recommended objective.

The computational results of the BCS algorithm are formally presented in chapter 4. The chapter begins with a discussion on the data used in the BCS problem, then moves on to recommend values for certain parameters within the algorithm. The results are presented and compared to those obtained through the simulation modeling the current scheduling strategy.

Chapter 5 provides a summary of this thesis and an overall conclusion. Ideas for future research are also discussed in this chapter, along with some ideas on how to pursue these future research topics.

Chapter 2 - Background Research

This chapter provides important background information on optimization techniques commonly applied to transportation/logistics problems. These techniques and problems lay the framework for the construction of the BCS algorithm. Exact methods are formally defined in section 2.1, while common approximation heuristics are described in section 2.2. The next section examines the use of simulations as modeling tools, and the final section of this chapter explores several frequently-occurring problems in logistics, summarizing examples of how each has been solved in the past.

2.1 Exact Optimization Methods

It is widely agreed upon that the beginning of modern-era research in optimization began with the theory of Linear Programming (LP). This term was first coined by George Dantzig in his 1947 publication, Maximization of a linear function of variables subject to linear inequalities. Since then, many other optimization methods have been developed and range from more specific instances of LP to different types of heuristics that cannot necessarily guarantee optimality. Almost all of the techniques have been used to solve logistical models.

Dantzig was the first person to publish the term Linear Programming. However, much of the theory had also been developed and used by Leonard Kantorovich in 1939 during the Second World War to optimally allocate resources to troops (Kantorovich, 1940). Both men helped to develop the theory behind the use of linear functions to model some objective as well any limiting factors of the decision variables within that objective function.

All LP formulations are made up of three components: an objective function to be either maximized or minimized, decision variables, whose values are to be determined, and any constraints that the solution cannot violate. Furthermore, all LP models share three basic properties: proportionality, additivity and certainty. Proportionality ensures the amount each variable adds to the objective function and constraints is directly proportional to the value of the variable. The additivity property requires the total contribution of the variables in the functions to be “the direct sum of the individual contributions of each variable,” and the final property simply means the coefficients are known and constant (Taha, 2007). A violation of any one of the three properties results in at least one function in the LP model to become not linear. In standard form, a general LP can be presented as:

$$\begin{aligned} maximize \quad & c^T x \\ subject\ to \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^{m \times 1}$, and $c \in \mathbb{R}^{n}$ all represent known coefficient matrices with m and n being the number of constraints and decision variables, respectively. Note that $x \in \mathbb{R}^{n}$ and represents the vector of decision variables. The last set of constraints, represented by $x \geq 0$, prohibits the decision variables from taking on negative values.

Further restricting the decision variables’ feasible values from any positive, real number to only positive, real integers converts a standard LP formulation to an Integer Program (IP). Allowing $x \in \mathbb{Z}^{n}$ requires the IP to be solved in a different manner than a more general LP. To solve an IP, the first step is to actually relax the integer requirements and solve the basic LP. Although the LP’s solution may not be the optimal IP solution, it is used a starting point to begin searching for the optimal integer solution. New LPs are continuously created to partition the search-space until the optimal IP solution is found.

One common method used to create the new LPs when solving an IP model is called the Branch-and-Bound algorithm. Created in 1960 (Land and Doig), this algorithm starts with the optimal solution of the LP relaxation and iteratively chooses a variable with a non-integer value to "branch" on. For example, if the selected decision variable $x = 1.5$ in the basic LP, it is selected for branching. Two new LPs $x \leq 1$ and $x \geq 2$ are added to the problem and evaluated. The formation and evaluation of these new "branches" continues until all branches are fathomed (it becomes impossible to yield a better IP solution than the current one found).

Although this algorithm is computationally simple, its major weakness comes in choosing the order in which to branch on variables. A simple branch-and-bound algorithm can drastically reduce or extend the time it takes to find the optimal solution simply based on which variable is selected in which iteration. Because of this, a large amount of research has gone into heuristics used to generate attractive branching paths for the branch-and-bound algorithm, as well as heuristics that begin with arbitrary solutions and use other techniques to search the solution space.

2.2 Heuristics

Heuristics are useful optimization techniques for problems with enough complexity to cause an exhaustive search to become impractical. These methods do not guarantee an optimal solution, but instead seek to find an acceptable one within a reasonable amount of time. Because of this, most heuristics are initialized with an arbitrary solution and employ ambiguous termination rules. The methods discussed in this section are divided into two types, neighborhood and solution-based, dependent on the general strategy used to move from one solution to the next within the search-space.

2.2.1 Neighborhood-based Heuristics

After defining the initial solution, neighborhood-based heuristics proceed from solution to solution by adjusting the current combination of variable values by some well-defined, but usually minor, detail. This well-defined rule is said to be the definition of the neighborhood. For any given solution, there typically exist multiple other solutions that could be derived from the same rule. All of these solutions are said to be neighbors, or in the same neighborhood. For example, if the rule is to randomly select two decision variables and swap their values, then the given solution *X=[1,2,3]* has three neighboring solutions: *X=[1,3,2]*, *X=[2,1,3]* and *X=[3,2,1]*. While all of the heuristics discussed in this section share a common search outline, they all differ in the way the choice is made to move from one solution to the next.

A heuristic that chooses to move from one solution to the next based solely on maximizing one function is referred to as hill climbing. Although it is a greedy heuristic, its advantage lies in the fact that all neighbors are examined each iteration, and is therefore effective for finding locally optimal solutions. However, its effectiveness at finding globally optimal solutions is largely situational and depends on the neighborhood design and starting solution (Pinedo, 2012).

Another common heuristic with a unique decision-making process is called Tabu Search. Originally developed to optimize employee scheduling (Glover and McMillan, 1986), this heuristic has been applied to many different research problems including vehicle routing (Taillard, 1997), job-shop scheduling (Nowicki, 1996), and chemical-process design (Wang et al, 1993).

Tabu Search uses a deterministic process of storing a finite number of solutions that have already been explored in a Tabu list. Any new solutions are only accepted if they are not currently listed. Given the typically large solution space for most problems out there, it seems unlikely that the same exact solution would be visited multiple times, regardless of the existence of a Tabu list. Thus it is common practice to instead check for similar characteristics when comparing a candidate solution to those already visited (Glover, 1989). There are many other common adjustments people have made in order to improve the effectiveness of Tabu Search including the addition of probabilistic properties (Chiu et al, 1996), random restarts (Battiti and Tecchiolli, 1994) and combining other heuristics (Kelly and Osman, 1996).

The final neighborhood-based heuristic discussed in the chapter is called Simulated Annealing (SA). The name comes from the annealing process in metallurgy, where newly forged metal pieces are forced to cool at a slower rate. When the metal is first heated, the atoms move around in an “excited” manner. As the metal cools, these atoms begin to “settle down” forming crystal structures. The slower the rate at which the metal is allowed to cool, the larger these crystal structures become, thereby increasing the overall tensile strength of the metal. This idea of controlling the rate at which the metal is cooled is translated to controlling the rate at which new solutions are accepted in the search through the solution space. Thus, this is the first heuristic discussed that utilizes probabilities in its basic definition.

The goal of SA is to counteract the greediness of always selecting the best available option by allowing a candidate solution to be accepted even if its objective value is worse than the current best (Kirkpatrick et al, 1983). This is accomplished by first comparing the current to the candidate solution. If the candidate solution has a less desirable objective value, it is still accepted with the probability $P = e^{-\frac{|f(S_k)-f(S_c)|}{\beta_k}}$ where $f(S_k)$ is the objective value of the current schedule in the k^{th} iteration,$f(S_c)$ is the objective value of the candidate schedule, and β_k is the decreasing control parameter. The control parameter β_k is typically chosen to be a^k for some a

between zero and one, and is called the annealing temperature (Pinedo, 2012). If the candidate solution has an improving objective value, it is accepted with probability $1 - e^{-\frac{|f(S_k)-f(S_c)|}{\beta_k}}$. This decision procedure allows the heuristic to move away from locally optimal solutions by occasionally accepting a worse candidate. However, because the control parameter is decreasing with each iteration, it becomes less likely that SA will continue down a path of inferior solutions.

Due to SA's versatility, the algorithm has been applied across a wide array of industries and classical research problems. From computer chip design (Betz, 1997) to crystalline structure prediction (Penettier et al, 1990), to the Traveling Salesman Problem (Allwright, 1989), SA's adaptability makes it a sound choice for many applications.

Although these neighborhood-based heuristics all use different strategies to move from one solution to the next throughout the search-space, each carries only one solution at a time through its iterations. The next section focuses on heuristics that expand their computation to utilize multiple solutions in the development of potential candidates.

2.2.2 Solution-based Heuristics

Solution-based heuristics move through the search space through the combination and manipulation of past solutions. These heuristics tend to be more computationally complex than the neighborhood-based, and some of these solution-based techniques even employ a neighborhood-based heuristic as a local improvement step. The increased complexity of the heuristics examined in this section is also due to the fact that multiple solutions are carried through each iteration in order to more accurately direct the solution search.

Solution-based heuristics also share the commonality of being based on occurrences in nature. Genetic Algorithms (GA) view the search though solution-space as a population's growth and decline through time. Ant Colony Optimization (ACO) tries to model the search for a decent solution as a colony’s search for food, and Particle Swarm Optimization (PSO) mimics swarm intelligence similar to that of a beehive or flock of birds. The basics of these three techniques are outlined in the rest of this section.

The roots of GAs are traced back to early computer simulations of the evolutionary process developed in the mid-to-late 1950s (Bariccelli 1954, Bariccelli 1957, Fraser 1957). While this early work laid the foundation for the basic structure of GAs, the evolutionary model wasn’t applied to the field of optimization until the 1960s, when Hans Bremmerman published several papers on the subject (UC Berkeley, 1996). More recently GAs have been used to optimize the design of pharmaceuticals (Bellew et al 1998), wireless radio networks (Chen et al 2010) and data mining techniques (Punch and Minaei-Bidgoli, 2001).

The GA views each solution as an individual member of a larger population and each iteration as a generation of set population. Each generation, some of the individuals die off (by simply removing the solution) and some procreate, producing child solutions. Individuals that die off tend to be less fit than others in the population. A solution’s fitness is most commonly decided by its objective value, although there are other strategies such as seeking certain characteristics within the individual solution. The creation of child solutions can be simple or complex, depending on the design of the heuristic. Most commonly, two parents are selected to contribute random parts of their sequences to their child in what is usually referred to as a crossover (Pinedo, 2012). Along with the birth and death of individuals, random mutations are also used to further help differentiate a child from its parents. Mutation is seen to be more effective if the parameter controlling its probability varies from generation to generation.

ACO was first proposed in 1992 by Marco Dorigo in his PhD thesis, *Optimization, Learning and Natural Algorithms*. Since then, ACO continues to be used to solve varying optimization problems, including vehicle routing (Toth and Vigo, 2002) job-shop scheduling (Baesens et al 2007) and the generalized assignment problem (Lourenco and Serra, 2002).

ACO uses a strategy to move through the search-space different from any of the heuristics previously discussed. While it does operate using multiple solutions, the way in which these solutions are found is unique to ACO and PSO. Multiple artificial ants begin at different points in the search-space, meaning each ant has its own starting solution. A local-search procedure is applied to each solution seeking some local improvement. These procedures are often a neighborhood-based heuristic. These locally-improved solutions are then compared to the objective value of the best solution known so far. The key to ACO lies in pheromone trails. This is how individual ants in the colony communicate with each other. If a “good” solution is found by an ant, pheromones are released to signal other ants. If more ants find the trail, more pheromones are released strengthening the signal. Conversely if a pheromone trail is ignored by other ants, it dissipates. Mathematically, these pheromone trails are updated every iteration, causing an ant’s movement throughout the search-space to be governed by both its individual knowledge as well as the information provided by the rest of the colony.

Although PSO shares some similarities with the basic structure of ACO, it also differs in how the individual particles are directed to move. Research into swarm intelligence began with Craig Reynolds publishing a behavior model of flocks of birds in 1987. He modeled individual’s movements in the flock through rules of separation, alignment, and cohesion (Reynolds, 1987). Then in 1995, Kennedy and Eberhart published the first paper on PSO.

As with ACO, many particles are initialized within the solution space. Each performs a local search of its area and communicates its best-solution to the rest of the swarm. Once again, the swarm’s overall best solution plays a role in the direction of the particles next movement.

However, PSO can combat being trapped at a local optimum by defining sub-swarms. For instance, an individual particle's swarm may be defined as only the three closest particles, even though the heuristic may be searching with 20 particles. These sub-swarms could also be defined as particles that are within a certain "distance" (usually judged by objective value) of each other. The difficulty that arises in PSO is finding a design balance that provides efficient results without converging too quickly to a local optimum. At the same time, well-designed PSO methods have proven to be very effective in combinatorial optimization problems (Kennedy et al, 2004).

Heuristics and algorithms all rely on certainty of data, meaning it is always assumed that the values of parameters inherent in the system are known. For example, when constructing an objective function, the associated cost parameter of each variable is assumed to be known and remain a constant value. If variable *X* always contributes a cost of three times its value, then it is represented by *3X* in the problem's objective function. Or, if a problem involves an arrival rate of parts, a heuristic must assume a constant rate in order to function. In reality this assumption may be invalid. It is more likely that parts arrive according to some probability distribution. This means the value of the arrival rate parameter continues to take on different values within the distribution as time progresses. This introduction of stochastic parameters greatly increases the solution space, so much so, that a heuristic may not be able to come up with a decent solution in a reasonable amount of time. When heuristics and other algorithms are no longer viable options for optimization, simulation is frequently used to model the problem. This allows the testing of different variable-value combinations along with the ever-changing stochastic parameters. The design of the simulation is important in constructing a useful model, as is discussed in the next section.

2.3 System Simulations

When discussing techniques used to solve logistical models, it is important to mention the use of simulations as optimization tools. As computers and simulation software continue to become more powerful, systems of increasing complexity continue to be modeled to provide realistic feedback of the effects of adjusting specific system parameters. Simulations cannot guarantee optimality and can only check the results of some "what-if" scenario. For example, a production facility may want to know what happens to their production measurements (throughput, work-in-progress, etc.) when a number of new machines are added, or if the factory layout is reconfigured. In industry, simulations are used to help find "optimal" solutions. In actuality, these are merely the best answer to all of the "what-if" scenarios analyzed. Some researchers have even included heuristics with simulation environments (Better et al, 2008).

Every simulation is broadly classified as either a discrete or continuous time model. The discrete time models comprise the vast majority of simulation models. In fact, very little research has been done on continuous simulations as they are a recent advancement in simulation enabled by the improved computational power of recent computers.

The choice of modeling a system discretely versus continuously refers to how the program views the passage of time, and the occurrence of events during that time. In a discrete-time model, a change in time is only noticed if there is a change in in the system. The key to a discrete event simulation is an event calendar. The event calendar keeps the time for each next event in ascending sorted order. Nothing changes in the system until an event occurs. Time moves forward directly to the time of this next event. At this event, more events may be created and are added to the event calendar. Statistics are updated. Thus, time leaps forward at different time intervals during the simulation (e.g. time = .076, 1.23, 4.51, 4.55,…).

In contrast, a continuous simulation moves time forward in small identical time increments (e.g. time = .01, .02, .03, .04, …). As time moves forward, the system is examined to determine if any changes to the system occurred during this time period. Such changes are accounted for and the system, including any relevant statistics, is updated in each time period. Recognizing the difference between discrete and continuous time models relies on the understanding of the idea that changes over a time interval (made up of time units) are different from changes per each one of the time units within that time interval.

When choosing to model time in either discrete or continuous format, it is important to take into consideration the information required as output. Professors Ossimitz and Mrotzek use a simplistic example to illustrate the process of matching up requirements of the model with the proper time-structure in their 2008 paper on system dynamics. Consider someone driving their car for a two-hour trip. If the driver wishes to know the average speed they traveled, they need only look at the odometer twice, at $t=0$ and $t=2$ hours (the odometer measures distance traveled). The difference in the values on the odometer (O_2-O_0) divided by the time interval of 2 hours will give the driver their average speed. This model corresponds to the discrete time-structure where a change is recognized over a time interval.

Now consider the same scenario, only the driver now wishes to know the exact location of the car at any point during the trip. Because the output requirements have changed, so must the formatting of time in the model. In order to calculate the location of the car at a specific point in time, the driver must now keep track of the location of the car at every point in time during the trip. In this model, time must be continuous, and could be practically modeled using "infinitesimally small" intervals, each one-second in duration. Clearly, the choice in time-structure is critical to design a useful model. For the purposes of the BCS algorithm, a continuous simulation was selected, due to the fact that cranes move, and any crane may have to be delayed at any given point in time due to the position of surrounding cranes. Thus, the position of the cranes should be known at each point in time.

Continuous time simulations play a vital role in several industries, the foremost being the Defense and Aerospace Industry, where continuous models are critical to autopilot functions (Wang, 1999), missile guidance systems (Blaukamp and Hawley, 2010) and outer-space navigation (Perozzi and Salvo, 2008). Continuous simulations are also often used in modeling chemical reactions (Andrews and Bray, 2004) and biological populations (Allen and Dytham, 2009). The is the first continuous time simulation of an OC system, to the best of the author’s knowledge.

All of the optimization techniques discussed in this section have both advantages and disadvantages for certain types of problems. The effectiveness of any one method relies on the specific design parameters and their relevance to an individual problem’s structure. To better understand the process of matching problems and solution techniques, the next section outlines some common logistics problems, illustrating how they have been solved in the past. All of these problems contain aspects relating to characteristics of the BCS problem.

2.4 Logistics Research

Logistics focuses on optimizing the flow of resources from origins to destinations in order to meet some given requirements. These requirements can vary greatly, from meeting production needs to supplying enough goods to fulfill customer demands. The goal is to ensure every resource is in the correct place at the correct time, thereby minimizing the total cost incurred. While logistics typically concentrate on the transportation of physical goods between facilities, it is becoming increasingly common for historical logistics-optimization methods to be applied to processes occurring within the facilities themselves.

The rest of this chapter outlines several basic classical logistics problems and details common variations of each. Examples of how these problems have previously been solved are also provided. Each of the classic problems contains features related to OC Scheduling (OCS), the final problem discussed in the chapter.

The first problem is typically referred to as the Vehicle Routing Problem (VRP). In general, the VRP involves deciding which route each vehicle in a fleet takes to meet all delivery requirements. This is a common problem in the airline (Lan et al, 2007), trucking (Shah, 2008), and intermodal shipping (Cordeau et al, 2008) industries. However, the VRP is also applied in less-commonly-thought-of situations such as the formulation of military and emergency response strategy (Ozdamar and Yi, 2006).

The origins of the next two problems are from only one industry, specifically railroad planning. Major railroad networks allow for an enormous amount of possible routing schedules for any given car. Due to this complexity, railroad planning typically involves splitting the process into several different optimization problems, all attempting to minimize some overall cost function. One of these problems is called the Railroad Blocking Problem (RBP) and is used to decide which cars make up which trains, and in what order. Once this first optimization is complete, typical VRP methods are applied to entire trains of cars. After calculating optimal routes, it is likely trains eventually meet somewhere on the tracks. To avoid collisions, one train must yield by either switching tracks, or taking a side spur and waiting for the other train to pass. Deciding which train yields depends on many factors, and is therefore an important optimization problem, typically referred to as the Meet-Pass Problem (MPP). The final problem examines the assignment of jobs to cranes operating within a factory, and is commonly referred to as Overhead Crane Scheduling (OCS). All of these problems are further explained in the following sections.

2.4.1 Vehicle Routing Problem

The VRP problem was first formerly analyzed in 1959, as a fleet of gasoline trucks delivering from a central depot to satellite stations with varying demand (Dantzig and Ramer). The problem was modeled as an LP attempting to minimize the distance traveled for the entire fleet while simultaneously meeting all demand constraints. This problem represents a basic formulation of the VRP. Over the years, many variations of the VRP have been developed to model classes of restriction commonly found in real-world scenarios. These variations include having pickup and delivery locations, limited vehicle capacities, delivery time-windows, route-length restrictions, etc. The implications of several of these constraint classes are discussed below.

Let the route-network of a basic VRP be represented by a graph $G = (N, E)$ where N is the set of nodes representing the depots (starting locations of the fleet) and the stations (delivery locations), and E is the set of undirected edges representing the available routes between the nodes. Edge *{i,j}* is the edge connecting node i to node j and $i \neq j$. Note the use of undirected edges over arcs implies there are no one-way-only streets present in the network. Let $C = \{c_{ij}: \{i, j\} \in E\}$ be the accompanying cost matrix. This cost of the edge can be in any form relevant to the problem (distance, time, dollars, etc.) as long as it properly signifies the penalty incurred for traveling between two given nodes. According to Gilbert Laporte (1992), the VRP consists of coming up with the least-expensive routing while ensuring:

i. Each delivery node is visited exactly once by one vehicle;
ii. Each vehicle starts and ends at the depots;
iii. All side constraints are met;

The third item in Laporte's list is where standard VRP variations are introduced, perhaps the most common of which is the Capacitated VRP (CVRP).

The CVRP is simply a VRP with limited vehicle capacities. While this additional constraint seems simplistic, it has been the study of many researchers over the past 40 years (Lal et al, 2009). Due to the fact that vehicles may now have to return to the central depot between customer deliveries in order to load more product, the complexity of a CVRP tends to be greater than that of a normal VRP. This has spurred many different optimization approaches over the years, including the Generalized Assignment Heuristic (Fisher & Jaikumar, 1981), Parallel Saving Algorithm (Altinkemer & Gavish, 1991), Column Generation Based Heuristic (Lal et al, 2009), and many others. Most of these optimization techniques involve breaking up the overall problem into separate sub-problems, dealing with vehicle routings and capacity loadings separately. Then these sub-problems are merged and adjusted until a "decent" overall solution is obtained.

Another common side constraint is the incorporation of release dates. In a standard VRP, all of the deliveries are known at the beginning of the optimization. When release dates are added, each delivery is demanded at a different time, and the delivery request is unknown up until that time. This greatly adds to the complexity of the problem. Now, the optimization technique employed in developing optimal routings must periodically be re-run as new deliveries are constantly being released into the system. How often the optimization technique must be re-used depends on the distribution of the time between release dates. The time period between optimizations should be large enough to allow multiple delivery requests to be released. As evidenced, when the VRP becomes more complex, it is useful to break the problem down into time “phases.” A similar process is used in BCS and is a common strategy in other logistics problems as well, such as the formation and routing of trains in the railroad industry.

2.4.2 Railroad Blocking Problem

The RBP involves assigning cars to shipping “blocks” made up of cars with common routing points. Each train is designated a group of blocks based on its route, and all of these assignments are re-evaluated at each station within the railroad network. It is important to note that the origin and destination of a “block” may not necessarily be the same as any of the origins or destinations of any of the cars within that block (Barnhart et al, 1998). This occurs whenever cars are being transported between two intermediate points that are neither its origin nor its destination. This problem relates to a special occurrence in the BCS problem. Whenever a job requires two cranes to pick up and deliver the item, a decision on which two cranes will make up the “block” with common routing points needs to be made.

The majority of the research attempting to optimize the RBP focuses on modeling the network flow as an Integer Program (IP) and then attempting to solve through either an optimization algorithm or heuristic search. Barnhart, Newton and Vance (1998) modeled a major railroad’s system as a network flow problem with nodes representing possible destinations and arcs representing possible “blocks.” To solve, they used a branch-and-bound algorithm “in which attractive paths for each shipment are generated by solving a shortest path problem.” In 19 test cases, the solutions were never less than 96.1% of optimal.

More recent research has turned from the traditional optimization techniques towards heuristic search to solve RBP models. In 2007, a group of engineers working for Innovative Scheduling implemented a Very Large-Scale Neighborhood (VLSN) search method to solve the RBP using data provided by several major railroad companies (Ahuja, Jha and Liu). The design of the neighborhood changes depending on the desired objective function and constraints. For example, the algorithm can be used to create entirely new blocking plans from scratch, or it can take a current blocking plan as input and only seek to improve based upon the desired objective. One neighborhood definition used to improve upon an input blocking plan allows at most one block swap causing at most a 25% increase in the traveling distance of the cars within the block. This neighborhood-search design allows the algorithm to search for solutions that actually increase the distance traveled. This particular design showed a 2.5% reduction in total costs when applied to the data provided by several major railroad companies. While this doesn't seem significant, this translates to approximately $20 million in annual cost savings.

Once a railroad has completed its blocking plan and set its potential train schedule, there is commonly one more problem to optimize. Given the linear, no-pass structure of railroads, and the limited amount of track space, it is common for trains to be assigned overlapping routes. When this occurs, two trains may end up on a collision course with each other, and one must yield by moving off the main route and waiting for the other train to pass. Deciding which train must yield is commonly referred to as the "Meet/Pass Problem" and is further explained in the following section.

2.4.3 Meet/Pass Problem

The Meet/Pass Problem (MPP) is typically handled by dispatchers working for the railroad. A dispatcher is responsible for deciding which trains are delayed when a possible conflict occurs in their assigned territory. This decision takes many factors into account including the varying train speeds, the product being hauled, the size and class of the train, as well as track conditions and locations. Over the past decade, much of the research into optimizing this decision revolves around network flow models (D'Ariano et al, 2007).

In 2007, Zhou and Zhong published their work on branch-and-bound algorithms with enhanced lower bounds used to resolve the MPP in single-track scenarios. A single-track scenario occurs when two trains meet on a single, main track with only one side track available for the delayed train. While this scenario seems trivial, this is actually regarded as the more difficult-to-handle scenario by train dispatchers (the other scenario being multiple tracks in parallel with connecting tracks between them). The single-track scenario involves more careful planning than the multi-track, due to the fact that the dispatchers simply have more track capacity and available routings to minimize delays in the multi-track scenario. However, from an algorithm design point-of-view just the opposite is true. The increased complexity of the multi-track scenario makes it more difficult to optimize the Meet/Pass problem.

2.4.4 Overhead Crane Scheduling

Of all the logistics problems discussed in this chapter, the scheduling of OC systems is relatively new. Most commonly, overhead cranes are used in large production facilities to move oversize items throughout the factory floor.

In essence, this problem can be viewed as a CVRP with pick-up and delivery and release dates. Each crane represents a vehicle with a capacity of one. The complexity in creating a “good” schedule arises from the fact that the cranes are unable to pass one another. This is similar to the way trains are unable to pass each other on the same track. In most overhead crane systems, all the cranes are operating on a single track. Thus, the problem involves a combination of aspects from a VRP and MPP. In the BCS problem, some jobs require two cranes to transport the item. The decision of which two cranes are assigned the job is similar to the RBP decision of which cars make up a block with common routing points. This aspect is unique to the BCS problem, as evidenced by existing research into the simpler OCS problem.

In 2009, a group of researchers from the University of Newcastle studied a copper smelting factory's OC system. The factory utilizes four cranes operating on a single-track in their smelting process. In seeking to improve the scheduling of jobs to cranes, the researchers utilize a Genetic Algorithm to randomly generate new solutions. In their implementation, jobs can be assigned to any of the cranes in any order. These potential solutions are read into a discrete event simulation to calculate the objective function value. Minimizing the makespan, or the completion time of the last job, is the desired objective for their particular case. The simulation itself takes into account the potential delaying of cranes due to job assignments that force the cranes to attempt to take overlapping routes at the same time. If these crane movement restrictions could be completely ignored (cranes are allowed to pass over each other) the minimum possible objective value is 8,380. This objective value represents the total amount of active crane time. After running 2500 iterations of the genetic algorithm the best solution's objective value is 9482, roughly 13% away from the ideal, yet infeasible lower bound.

In 2011, a similar OCS problem was studied at a metal-works factory in China. The factory employs two overhead cranes to transport unfinished metal pieces to different refinement stations spread throughout the factory floor. Some of the assumptions made by the group greatly simplify the problem, including the assumption that all jobs are released at the beginning of the day, and that a crane cannot be delayed once it begins a job. This way, if two jobs (one on each crane) have overlapping routes, the cranes alternate completing the jobs, with one crane waiting until the other has completely finished the task before beginning its own job. The amount of time the second crane must wait before beginning its conflicting job is referred to as the job's idle time. The job with the largest idle time is identified as the "key task."

The researchers developed a two-phase heuristic that first seeks to create an initial feasible solution, and then looks to improve the makespan of the solution by transferring “key tasks” from one crane to the other. The initial solution is created by dividing the factory floor in two, longitudinally. Jobs whose starting locations are in the “top area” are assigned to one crane, while the other jobs (with starting locations in the “bottom area”) are assigned to the other crane. After this initial assignment, the “key task” is identified and moved to a random position on the other crane in an attempt to minimize the makespan.

In order to test the heuristic the researchers were provided with a real set of 27 jobs that, when actually completed in the past, took 435 minutes. Using their schedule heuristic, and a discrete event simulation, it was estimated the jobs could be completed in 280 minutes. While this is a large potential improvement, it was unclear whether or not this improvement would actually be realized should their heuristic be implemented in the factory.

Both of these previous studies in OCS chose to minimize the makespan as an objective. When assigning jobs to machines, this is one of the most prevalent objective functions. A schedule with a smaller makespan is generally believed to be a more efficient schedule.

The makespan is a common objective in scheduling theory, and represents the end time of the last job completed (Parker, 1995). This objective, represented by C_{max}, is calculated for a set of jobs $J = \{1, \ldots, n\}$, assigned to a set of machines $M = \{1, \ldots, m\}$. Traditionally, this is calculated by first defining the completion time on machine k, $C_k = \sum_{i \in J_k} p_i$ where J_k is the set of jobs assigned to machine k, and p_i is the process time of job i. Then the makespan is the maximum completion time of all machines, or $C_{max} = max_{k \in M} C_k$. This may be difficult to capture in the instance of OCS, due to the fact that cranes cannot pass over one another. When two cranes are attempting to reach locations on opposite sides of one another, one crane must yield for a specific amount of time, thus increasing the current job’s process time. Therefore the process time of any single job varies depending on the ordering of all the jobs in the system.

This gives rise to the need of alternate objective functions based on the total amount of delay time (occurring whenever one crane is forced to yield to another) and the total travel time between jobs. The sum of these two values is viewed as the total amount of non-productive crane time (i.e. time that can be avoided with a more efficient schedule). Another option is to combine all three objective functions. This results in the objective of minimizing the non-productive crane time, with a reasonable makespan. The reasonable makespan implies the avoidance of overloading one crane by ensuring all active cranes have similar workloads. This is the objective the BCS algorithm attempts to optimize.

This section has reviewed several common logistics optimization problems. Although all of these can be related to some form of a crane schedule, their industry-specific requirements provide enough additional complexity, that each is considered a different optimization problem, requiring differing techniques to solve. Chapter 3 formally presents the BCS problem and subsequent algorithm, highlighting the relationship between certain characteristics of the classical logistical problems previously discussed.

Chapter 3 - Bay Crane Scheduling

As previously mentioned, the Bay Crane Scheduling (BCS) problem combines certain aspects of several other traditional logistics optimization problems. Similar to a CVRP with Pickup/Delivery, release dates, and all capacities equal to one, BCS creates an ordered schedule of pickups and deliveries for a fleet of vehicles. By assigning each vehicle's capacity to be one item, the vehicles are forced to alternate picking up an item and then immediately delivering that same item. The fact that each delivery has its own unique release date implies the BCS algorithm is re-run every so often, in order to schedule newly released requests.

Differing from the traditional VRP, the BCS does not require the vehicles to return to any initial depot nor does it restrict the number of visits to a delivery location to be exactly one. It is possible for repeated jobs to come up in BCS and therefore multiple trips to delivery locations may be required.

Due to special situations arising as a result of scheduling cranes versus traditional delivery vehicles (i.e. trucks or planes); the BCS also draws from characteristics of the traditional Railroad Blocking Problem (RBP). A job, consisting of a pickup and delivery request, can require two cranes due to the size of the item being transported. When this occurs, the decision as to which two cranes the job should be assigned is similar to deciding which cars go into which blocks in the RBP. The idea is to create an efficient pairing, or block, of cranes based on common routing destinations.

The linear movement restriction of cranes over traditional vehicles implies the possibility of delays. In each bay, a crane cannot pass another in the *y*-direction. This means that when multiple cranes' routes overlap all but one of these cranes waits and possibly moves with the non-delayed crane. This is similar to the Meet/Pass problem common to the railroad industry.

The similar route restrictions due to operating on tracks apply, and therefore decisions pertaining to which trains yield are constantly evaluated in an attempt to minimize overall delay-cost in the system. Unlike the MPP, there are no side-tracks available for a crane to temporarily occupy while another crane passes.

While the BCS problem may initially seem like a simple OCS problem, there are drastic differences between the two. The OCS problems previously studied all operate in a single bay with a fixed number of cranes. The BCS problem involves three adjacent bays, and allows for cranes to switch between the bays at certain access points, called bridges. The number of active cranes varies from shift to shift in the BCS problem, and there exist special job-requests that require two cranes to move items, as well as jobs with pick-up and drop-off locations in different bays. All of these differences cause the BCS problem to be more complex than any OCS problem previously studied.

Combining existing research on these traditional problems with new techniques designed specifically for overhead crane routings led to the creation of the BCS algorithm formally defined in section 3.1. Several common objective functions are defined and discussed in section 3.2, followed by a small-scale example of the algorithm in section 3.3.

3.1 The BCS Algorithm

This section defines the required input, main steps, and output of the BCS algorithm. This algorithm is designed to assign jobs to cranes located in separate, yet accessible bays in a large production facility. As previously stated, the jobs have associated release dates, implying the algorithm is run multiple times as more jobs are released. The algorithm is run every hour throughout the simulation to schedule jobs that have been released within that previous hour. Whenever the algorithm is re-run, the first few jobs on each crane remain fixed to maintain a consistent, short-term schedule for the crane and floor crews. To test the BCS algorithm, it is used to schedule jobs over several consecutive 8-hour shifts.

The first step of the algorithm is to assign jobs to cranes in no particular order. After the initial allocation of jobs to cranes, the jobs are then sequenced in an effort to optimize the objective function. Next a schedule is produced using a continuous time simulation. This schedule defines estimated starting and ending times of all jobs in the system as well as the total amount of delay time and travel time between jobs experienced by all of the cranes. This allows the calculation of the objective function value and provides a reasonable starting solution. Because the run time of the algorithm up to this point takes less than 0.5 seconds, an improvement heuristic search is applied in an attempt to further optimize the schedule. The BCS algorithm begins with properly defining its input.

3.1.1 Input

The key inputs to the BCS algorithm include:

$C = \{c_1, \ldots, c_m\}$, a set of m cranes. Each crane k has parameters:

(vx_k, vy_k), travel speeds in the x direction and y direction

$J = \{j_1, \ldots, j_n\}$, a set of n jobs. Each job i has parameters:

r_i, release time
p_i, estimated process time
o_i, a priority ordering
(sx_i, sy_i, sb_i), a starting x and y location and a beginning bay
(ex_i, ey_i, eb_i), an ending x and y location and an ending bay
nc_i, the number of cranes required

$B = \{b_1, \ldots, b_q\}$, a set of q bays. Each bay h has:

$(x_{1_h}, y_{1_h}), (x_{2_h}, y_{2_h}), (x_{3_h}, y_{3_h}), (x_{4_h}, y_{4_h})$, four extreme points defining the bay's boundaries

$CD = (cd_1, \dots, cd_u)$, a set of u cross docking locations. Each cross dock l has:

(cb_l, cy_l), a lower bay and y location; observe that this is a cross dock between bay cb_l and bay $cb_l + 1$ and located at height cy_l in between the two bays.

With the required input properly defined, the algorithm begins by establishing the initial position of each crane in its respective bay. After this routine is completed, the cranes' positions are set to mimic the distribution of job locations. If the jobs are evenly dispersed throughout the entire bay area, cranes are also evenly spaced throughout that bay. However, if job locations tend to cluster towards one end of the bay, then the cranes' initial locations are set towards that end of the bay. Figure 3 below illustrates this idea. The purpose of this step is to maintain a reasonable makespan.

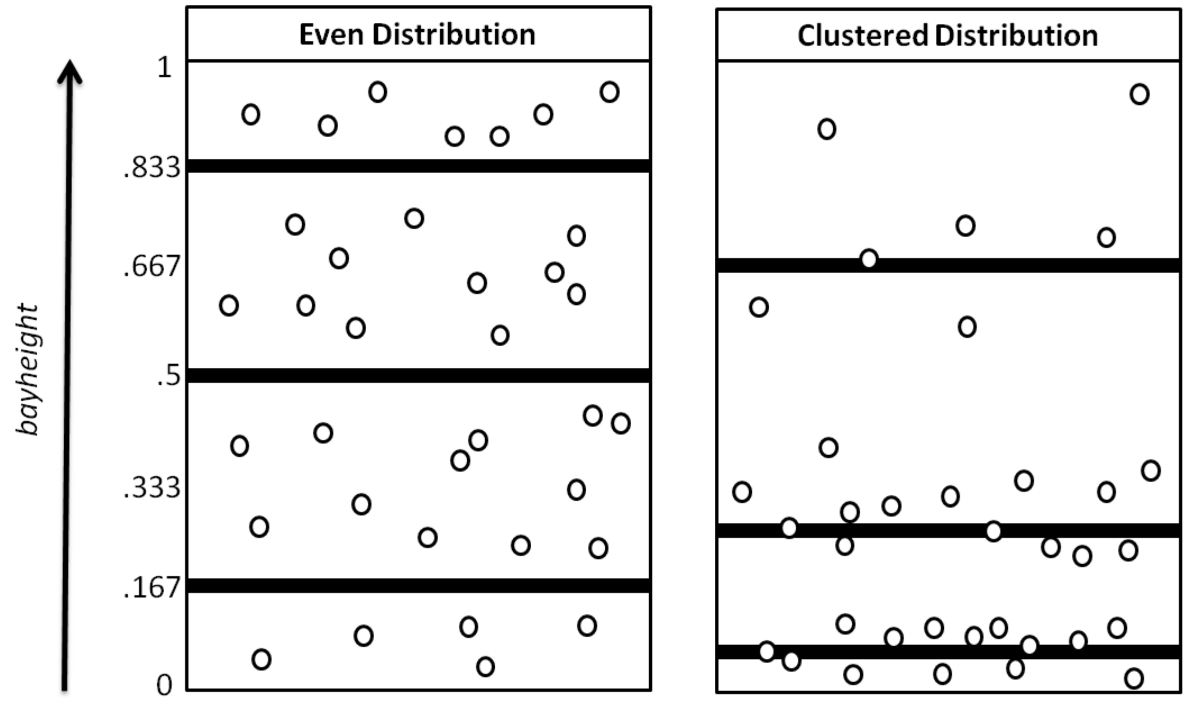


Figure 3 - Possible crane alignments

Note that a crane’s position is represented by a single *y*-coordinate corresponding to a location along the vertical axis of each bay. This axis is referred to as the *bayheight*. Also observe that $0 \leq bayheight \leq 1$, and all location-coordinates of jobs and cranes are scaled accordingly. It is also important to point out the location of each job is formally defined to be the average *bayheight* of its pickup and drop off locations. As can be seen in Figure 4, Case 1 illustrates the scenario of evenly distributed job locations and matching crane positioning. The second case shows a possible crane initialization when job locations tend to group more towards one end of the bay. This involves a two-step process of initial assignment based on distance, and then adjustment based on crane workloads.

In all bays, for each job, the distances between that job and all cranes in that bay are calculated and compared. Note, the cranes are originally assigned to be evenly distributed (i.e. *bayheights* at 0.25 and 0.75 for two cranes in a bay, at 0.167, 0.5, and 0.833 for three cranes in a bay, etc.). A job is assigned to the crane nearest its location, with the job location still being defined as the average *bayheight* of its beginning and ending coordinates.

One of the drastic differences between the BCS problem and other traditional crane scheduling problems is the fact that some jobs require two cranes to transport an item. When assigning a job requiring two cranes, the first crane to which the job is assigned is selected in the same manner as previously described. Due to its requirements, however, this job must also be assigned to another crane. The algorithm searches for the next closest crane to the job that is positioned in the same bay as the job and first crane. If there is only one crane in the bay with the job, the algorithm instead searches for a crane with the closest *bayheight* to the job’s location, only this time searching through the cranes in the other bays.

Always selecting the crane(s) nearest the job's location results in a greedy allocation of jobs to cranes. In order to combat this and avoid overloading one crane, the process time estimates are summed for each crane. If the absolute difference of the total process times of two cranes in a bay is greater than a certain time-threshold, d, then jobs are transferred from the crane with the higher total process time (c_{max}) to the crane with the lower total (c_{min}). It is important the threshold parameter is relatively small enough to avoid overburdening one crane.

When switching jobs from one crane to another, there are four possible scenarios that could arise pertaining to the positioning of the cranes involved in the switch. In all cases one or more jobs are removed from c_{max}, and an equal number of jobs are added to c_{min}. Jobs are transferred one at a time and this process is repeated until the absolute difference of the workloads between c_{max} and c_{min} in a bay is less than the threshold parameter, d. Continually transferring jobs may enter into an infinite cycle, and this possibility depends on the estimated workloads of c_{max} and c_{min} (p_{max} and p_{min} respectively), the value of d, and the estimated process time of the job to be transferred, p_t. This process will enter an infinite cycle if c_{max} and c_{min} are adjacent, and $p_t \geq \frac{(p_{max}-p_{min})+d}{2}$.

In order to avoid being trapped in the infinite cycle, the BCS algorithm keeps a count of how many times transfers occur in each bay. If this count goes above a number of iteration threshold, *NI*, then the cycle terminates, and the algorithm continues to the next step. This prohibits an infinite cycle.

As mentioned above, when transferring jobs from c_{max} to c_{min} the selection of the *transfer job* in each iteration depends on which one of the four crane-positioning scenarios the state of the system reflects. Figure 2 below illustrates the first two scenarios both immediately before and after a job is transferred. These two scenarios represent the cases where no intermediate cranes are positioned between c_{max} and c_{min}.

In Figure 2, the cranes c_{max} and c_{min} are labeled in both cases, and the diagonally-striped zone represents the area containing jobs assigned to c_{max}. Note jobs assigned to c_{max} are represented by squares, while jobs assigned to c_{min} are represented by circles. In both cases the *transfer job* is identified by the darkened square.

In Case 1, c_{max} is located directly below c_{min} along the *bayheight* axis. In this scenario the job currently assigned to c_{max} with the highest *bayheight* location is identified as the transfer job and moved to c_{min}. The "Case 1: Post-transfer" panel (upper-right in frame 2) shows how the zone in which jobs assigned to c_{max} are located contracts while c_{min}'s zone expands to encompass the *transfer job*. Case 2 differs from Case 1 only in that the positioning of c_{max} and c_{min} is now directly opposite of that in Case 1. c_{max} is now located directly above c_{min}. It makes intuitive sense then, that the job to be transferred from c_{max} is now the job with the lowest *bayheight* location.

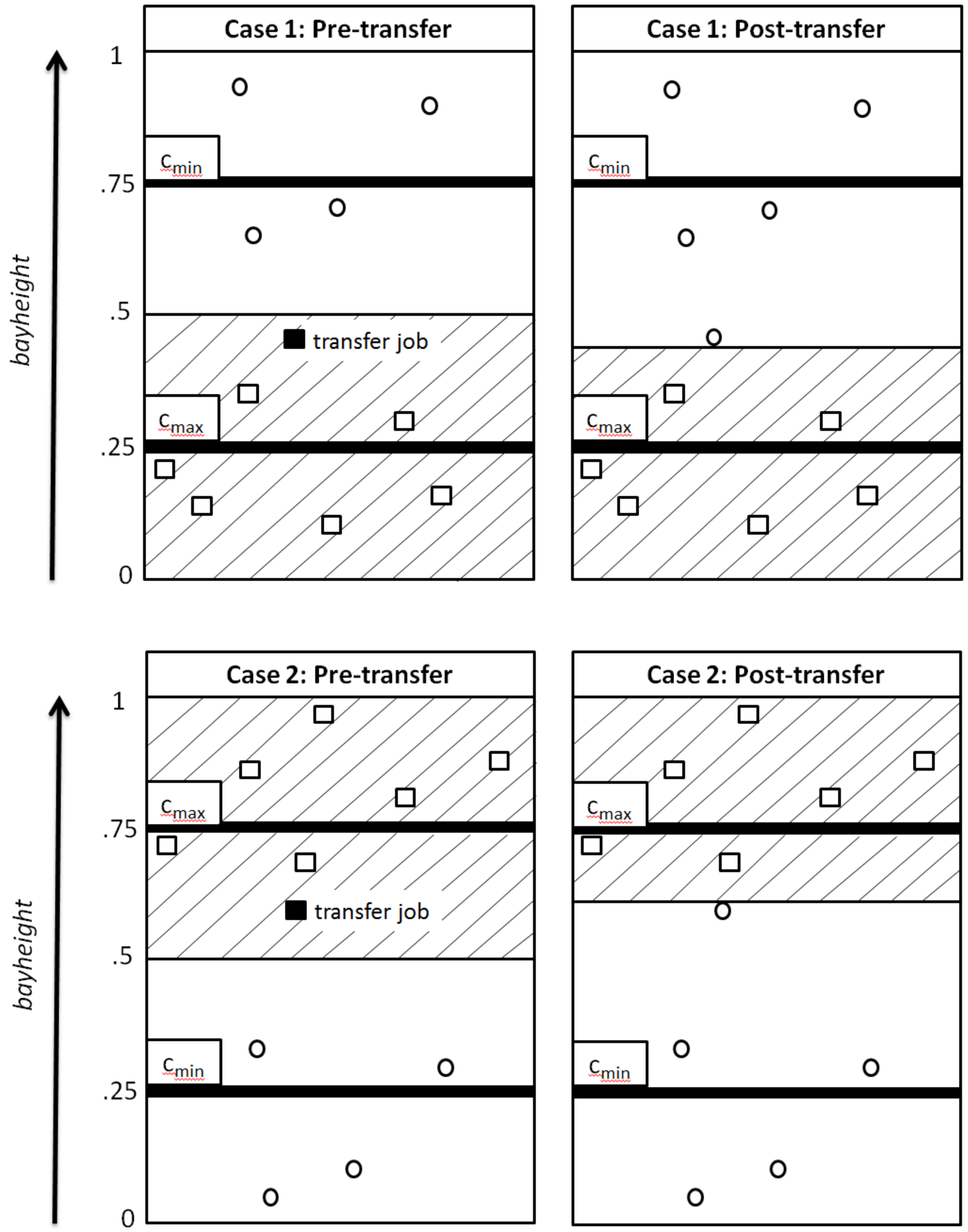


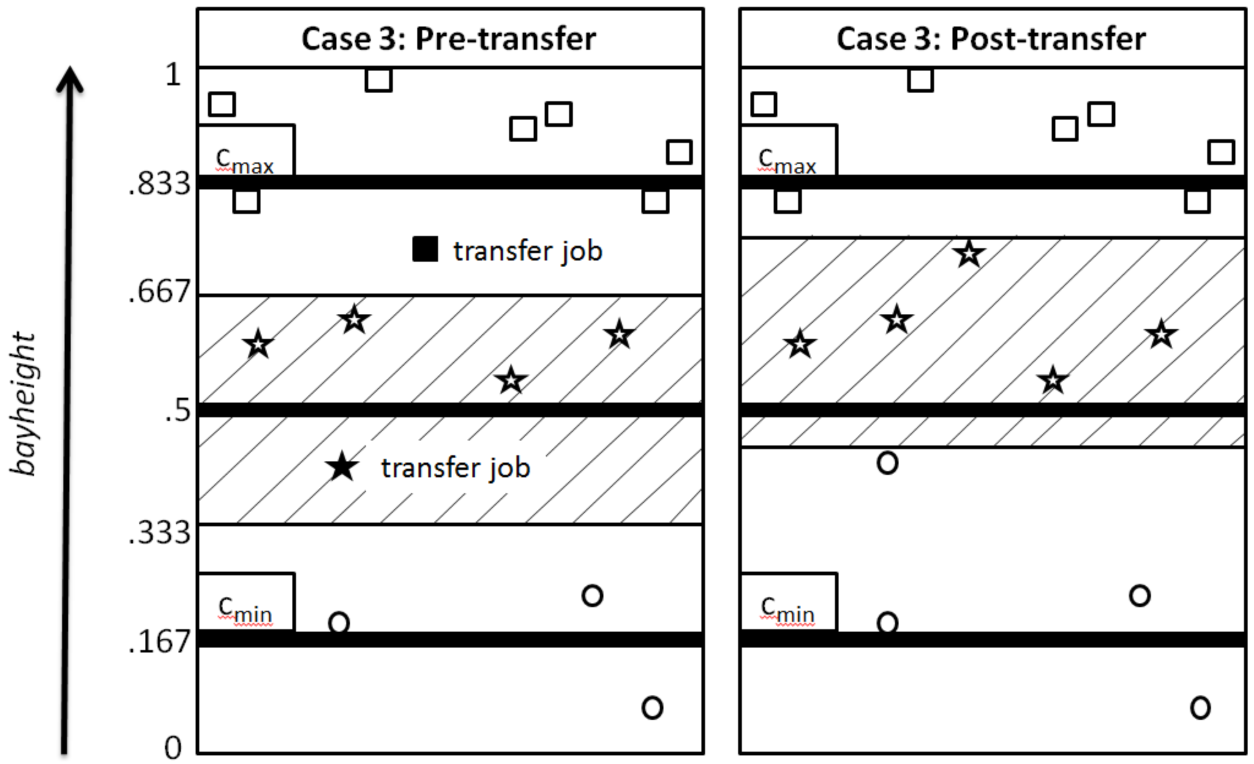
Figure 4 - Job-switch cases 1-2

In the next two crane-positioning scenarios (Cases 3-4) there is now at least one intermediate crane located between c_{max} and c_{min}. Figure 5 demonstrates the transfer of jobs in both of these cases.

In Figure 5 c_{max} and c_{min} are clearly labeled with squares once again representing jobs assigned to c_{max}, and circles symbolizing jobs assigned to c_{min}. The jobs assigned to the intermediate crane (unlabeled) are represented by stars, and the diagonally-striped zone represents the intermediate crane's job-zone.

In Case 3 c_{max} is positioned higher in the bay than c_{min}, thus jobs are transferred downward and the candidate job to be removed from c_{max} has the lowest *bayheight* location. This job is reassigned to the intermediate crane adjacent to c_{max} and in the direction of c_{min}. Next, the candidate job to be moved is identified on this intermediate crane as the job with the lowest *bayheight* and transferred to the next adjacent crane in the direction of c_{min}. This process is repeated until a job is moved to c_{min}. Whenever jobs are transferred from crane to crane, the cranes' respective job zones expand or contract accordingly. The "Post-transfer" panels in both cases illustrate this idea. This process may again repeat.

Case 4 differs from Case 3 only in that the positioning of c_{max} and c_{min} is flipped. In Case 4, the jobs transfer up from c_{max} towards c_{min}. This means the candidate jobs-for-transfer on each crane now have the highest *bayheight* locations.



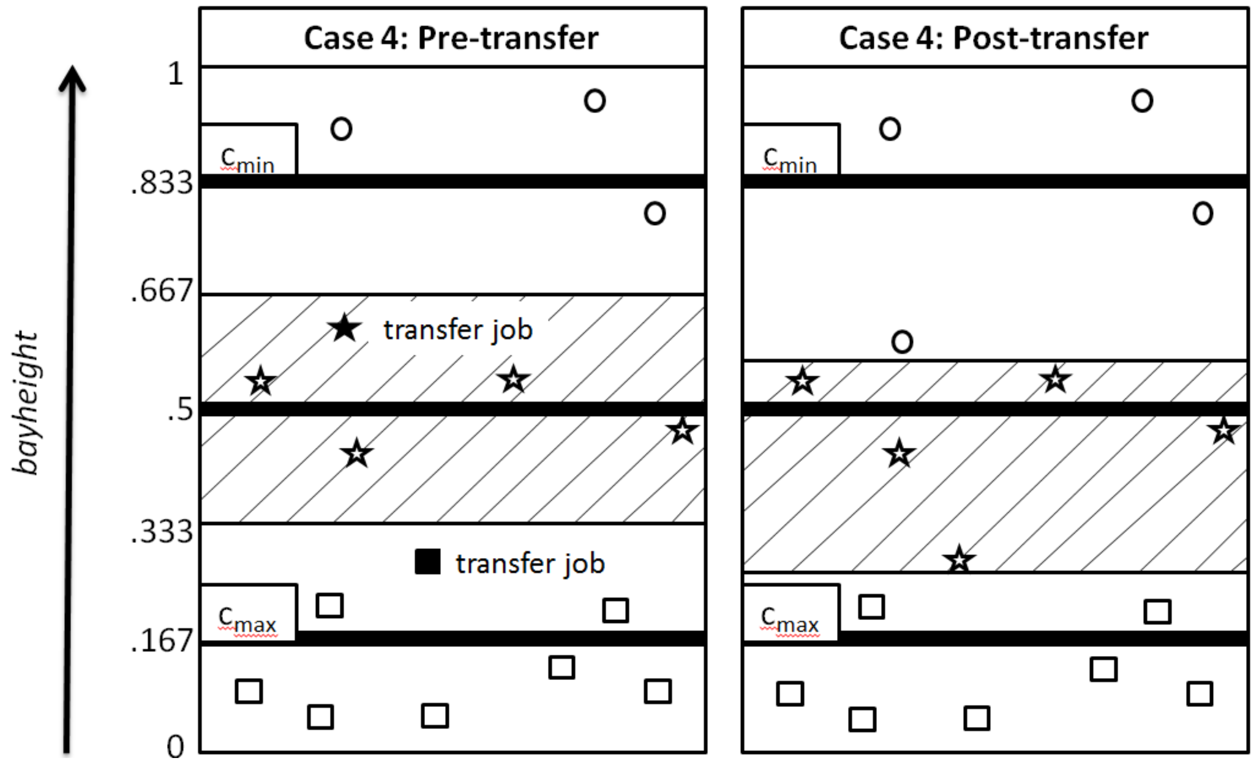


Figure 5 - Job-switch cases 3-4

As previously stated, once the absolute difference in the workloads of c_{max} and c_{min} falls below *d*, or the number of transfer iterations exceeds *NI*, the algorithm moves on to the next step of job sequencing. The end result of this logic provides a set of jobs assigned to cranes with a reasonable makespan. In addition, the jobs are arranged so that there should be very few delays as they are organized on cranes based upon their starting and ending locations.

3.1.4 Job Sequencing

Now that all active cranes have jobs assigned to them, the next step is deciding in what order each crane completes its jobs. When ordering the jobs on each crane, it is important to note the first few jobs in the sequence are viewed as "fixed" in place and cannot be adjusted. This comes from the incorporation of release dates. BCS's main steps are periodically re-run as more jobs become available to schedule. Newly released jobs may be assigned to a position in front of a job that was released in a previous period. However, no newly released jobs are ever allowed to supplant jobs located in the "fixed" positions. This is to provide both crane operation crews and floor operators with some sort of definite short-term schedule, as well as to avoid constantly pushing a "bad" job to the back of the sequence. The number of fixed jobs is a parameter whose value mostly depends on personal preference of the crane operators and the saturation level of jobs in the system. This parameter is set by the user.

Sequencing the jobs begins at the job-slot directly after the last fixed job position. A sequence score, ss_i, is calculated for all jobs assigned to the crane that are not currently in a fixed position. The value of ss_i is 100 times job i's priority order plus the time it takes to travel the distance between the last job scheduled on the crane and job i. Equivalently, $ss_i = 100 \times o_i + td[(ex_f, ey_f, eb_f), (sx_i, sy_i, sb_i)]$ where o_i is the priority score of job i (a smaller priority score corresponds to a higher priority), and $td[(ex_f, ey_f, eb_f), (sx_i, sy_i, sb_i)]$ is the time it takes for a crane to travel the distance between the ending location of the last scheduled job position, *f*, and the starting location of job *i*. Instead of greedily taking the job that minimizes this sequence score, the algorithm checks to see if placing the job in that position causes any travel conflicts with other cranes, resulting in delays.

The algorithm employs a series of continuous time simulations to verify whether or not a delay results from that particular job assignment. Each simulation involves only two cranes, one being the crane whose jobs are currently being sequenced (c_s), and the other being an active crane whose jobs were already ordered (c_o). This simulation is then repeated for all such ordered cranes.

The beginning time of the simulation is set to the completion time of the job previously ordered job on c_s and the end-time of the simulation is set to the time when the next potentially scheduled job on c_s is completed. From there, time is incrementally driven forward by some small amount (e.g. 0.1-minute increments). Whenever time progresses, the locations of the two cranes are updated. A delay is said to occur whenever one of the cranes attempts to move to the opposite side of the other crane. This situation can arise when both cranes are moving, or if one crane is stationary because it is in either the loading or unloading phase of a job. Because cranes cannot physically pass each other in the same bay along the *bayheight* axis, one of the two cranes involved in the situation must either stop moving or get out of the other cranes way. For example, if c_s is initially located above c_o in the bay, and then at some point during the simulation, c_s becomes positioned below c_o, a delay has to occur. If the simulation runs to completion, with no delay scenarios occurring, then the placement of the potential job on c_s does not cause any conflicts with other cranes.

This continuous time simulation terminates early due to several different reasons. If c_s and c_o are located in different bays, or if c_o is located in the same bay as c_s, but has completed all its jobs by the time the simulation begins, the simulation immediately terminates and reports back to the algorithm that no delays occurred. Also, as soon as a delay scenario is detected the simulation ends and reports back the potential job causes conflicts with other cranes.

The ability of the BCS algorithm to detect potential delay scenarios allows the selection of a job with a higher score, if it avoids causing any delays. If all possible jobs are going to cause a delay, then the job with the minimum sequence score is taken. Once all the jobs have been assigned to and sequenced on each crane, the BCS algorithm once again employs continuous time simulation to generate estimated schedule times (beginning and ending) for all the jobs, as well as keep track of the amount of delay time each crane experiences.

3.1.5 Schedule Generation through Continuous Time Simulation

In order to generate estimated schedule times for all the jobs, each crane's movements are simulated in a continuous time format. When generating the scheduled times for the first crane in each bay, it is assumed this crane never experiences a delay. If this first crane ends up conflicting with another crane in its bay, the other crane is forced to yield. The first job on this first crane is assumed to begin at time zero. This job's end time then equals its start time plus its process time. Each job's process time is broken down into three parts: the time it takes to load the item being transported, the time it takes to transport the item to its destination, and the time it takes to unload the item. The next job's start time then equals the ending time of the previous job plus the time it takes to travel from the previous job's ending location to the current job's starting location. This sequence continues through all of the jobs on the first crane.

For all subsequent cranes in the bay, the same schedule generation process is used. The first job on each crane still begins at time zero, and its ending time is based off of the job's process time. All subsequent jobs on these cranes have start times that depend directly on the ending time of the previous job, plus the time it takes for the crane to move to the current job's starting location. However, now the movements of any previously scheduled crane are also taken into account, and any conflicts result in the addition of delay time to the current crane's job-schedule. Delays can occur at any point in the simulation. For example a crane may be delayed when actually processing the job (loading, delivering, or unloading the item), or when the crane is traveling to its next job.

Unlike the previous continuous simulation, which terminates as soon as a delay scenario is recognized, this continuous time simulation lasts through the entire delay in order to calculate the appropriate amount of delay time to be added into the crane's schedule. It is important to clearly define what constitutes the entirety of a delay in these simulations. Figure 5 below illustrates a typical delay-scenario.

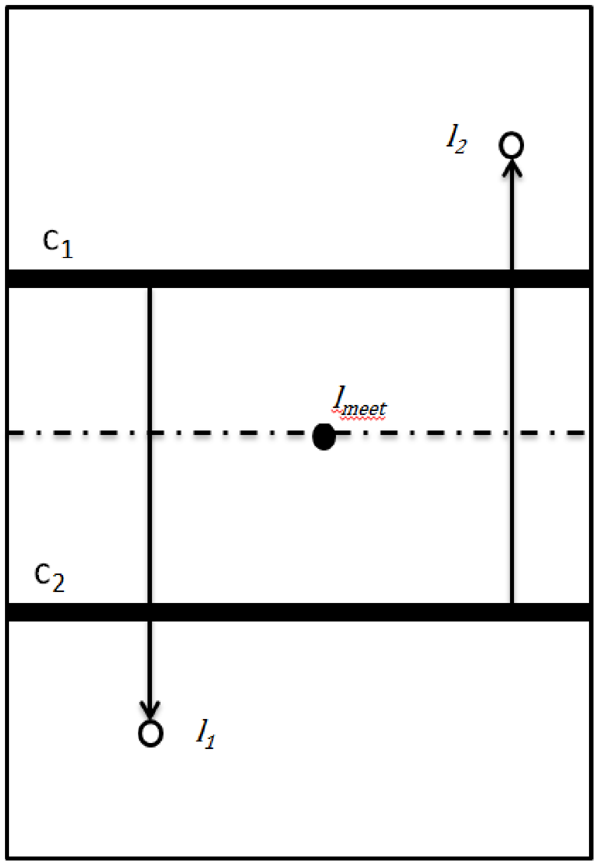


Figure 6 - Delay-causing scenario

In Figure 6 above, c_1 is moving down to drop off an item at location l_1 while c_2 is attempting to move up to location l_2 to pick up an item for transport. In this case c_2 is arbitrarily chosen to yield, and must therefore move with c_1 to allow it to reach location l_1. Had the two cranes continued on their original path, they would've met at a specific location l_{meet} and time t_{meet}. This time is set as the beginning of the delay time for c_2. The delay will end for c_2 when one of three things occur:

1. c_1 completes the rest of its jobs
2. c_1 leaves the bay
3. c_1 returns to a *bayheight* location $\geq l_2$

If c_1 completes all of its jobs, there is no reason for c_2 to continue to be delayed. When this occurs, c_1 is forced to immediately move out of the way of c_2. The delay of c_2 also terminates if c_1 exits the current bay. This occurs whenever c_1 is processing a job requiring a cross-bay move (where the job's starting and ending locations are in two different bays). Finally, if c_1 returns to a *bayheight* equal to or above the location c_2 is attempting to reach, the delay time of c_2 ceases to continue, as c_2 can either begin loading or unloading its next job.

The time the delay ends t_{stop} is set to the current time when one of the three previous conditions is met, and the total delay time of c_2 caused by this conflict is the value of the expression $t_{stop} - t_{meet}$. These calculated delays are added into the schedule appropriately, and the simulation continues to run until all jobs have a scheduled beginning and end time. Also, the total amount of delay time experienced by each crane is stored in the output variable dl_k, and the total amount of travel time between jobs (excluding delays) is output as tt_k.

On average, this step of generating a schedule through continuous time simulation takes less than one-hundredth of a second. This rapid computational time allows for the addition of a neighborhood-based improvement heuristic to the BCS algorithm. This also means the time increments through which the continuous time simulation progresses, which are currently set to 0.1 minutes, could be set to a smaller value, thereby increasing the accuracy of the simulation. However, given the large scale of the manufacturing facility, a smaller increment is deemed unnecessary. The additional improvement heuristic is used to search for better sequences of jobs on the cranes. Whenever a new solution is produced, this schedule-generating simulation is called to evaluate its quality.

3.1.6 Improvement Heuristic

Once the schedule has been updated for the current solution, a Simulated Annealing search heuristic is applied for further optimization. In this case, the job contributing the most delay time to the schedule, job j_d, is randomly switched with an eligible job. To be eligible a job must be assigned to the same crane as j_d. Also, the eligible job must have a release date prior to the start time of j_d and a current start time after the release date of j_d.

After the switch is made, the schedule is then recalculated along with a new objective score Z, which can either be improving or non-improving. An improving solution is kept with probability $P(X) = e^{-A^I(Z_{old}-Z_{new})}$, conversely an inferior solution is accepted with probability $1 - e^{-A^I(Z_{old}-Z_{new})}$, where A is the annealing temperature, I is the iteration count of the heuristic, Z_{old} is the previous solution's objective score and Z_{new} is the new score. This probability is set up so the chances of taking a non-improving score decrease with the number of iterations.

The annealing constant is a way to force the chances of taking an improving score to tend towards a specific direction. Again, a higher iteration increases the chances of moving to an improving solution. The heuristic's ability to accept a less desirable solution helps to avoid being trapped at a local optimum. Throughout the annealing process, an overall best *Z*-score is stored and updated as necessary. A variety of objectives could be used in the calculation of *Z*, and are discussed in section 3.2. This score is eventually reported as output. Upon running a set number of iterations, BCS outputs the results.

3.1.7 Output

The output is a set of ordered schedules $S = (SC_1, \dots, SC_m)$ where each SC_k is a set of ordered jobs, $SC_k = (j_{SC_{k_1}}, \dots, j_{SC_{k_{|SC_k|}}})$ for each $k = (1, \dots, m)$ such that $J \subseteq \cup_{k=1}^{m} SC_k$ and furthermore, any job needing $w \geq 2$ cranes is contained in w of the SC_k's. Every job has an estimated beginning and ending time (st_i and e_i respectively), and each crane has a total delay time (dl_k) and travel time between jobs (tt_k). Note the travel time between jobs does not include any delays that may have occurred, as this is captured in the delay time. A score, Z, corresponding to the value of the objective function and representing the quality of S is also output. The selection of useful objective functions is discussed below.

3.2 Objective Functions

When solving the BCS problem there are a few different objective functions that make intuitive sense to optimize. The most traditional of these is the makespan, or the time the last machine has completed its jobs. It is widely accepted that minimizing the makespan implies an efficient use of the machines (Gary, 1976). However, as discussed in section 2.4.4, other objectives such as minimizing the total delay time or the total travel time between jobs are also useful in the instance of crane scheduling. These objectives are formally defined in accordance to the BCS problem in the following sections.

3.2.1 Minimize Makespan

The makespan is the maximum completion time of all machines, or $C_{max} = max_{k \in M} C_k$ where C_k is the ending time of the last job, $e_{j_{|SC_k|}}$ on crane k. While this seems slightly complicated to directly calculate, recall these values are automatically calculated and output in the continuous time simulation used to generate the actual schedule times and C_k is the end time of the last job on crane k. The makespan objective provides a decent way to generate an overall efficient schedule for the cranes.

3.2.2 Minimize Travel Time

The objective to minimize the travel time between jobs on each crane forces the BCS algorithm to search for solutions with efficient routings of the cranes. The idea here is to minimize one part of the non-productive crane time (the other part of non-productive crane time being delays). The total travel time $TT = \sum_{k \in C} tt_k$ where tt_k is the total travel time of crane k. As stated above, the advantage of using this objective function is that it will provide efficient routings of cranes. However, it may force more delays in order to generate these routes.

3.2.3 Minimize Delays

The objective function minimizing total delay is formally defined as $TD = \sum_{k \in C} dl_k$ where dl_k is the total delay time experienced by crane k. This objective seeks to avoid delays as much as possible. This is useful when trying to minimize the more-avoidable, and often the larger part of non-productive crane time. No matter how efficient the routes are, there will always be some sort of travel time between jobs. Once a delay starts it tends to continue for longer than most travel times between jobs. Thus it may be helpful to seek schedules that are willing to sacrifice some of the route savings in order to save more of the non-productive crane time that occurs from delays.

3.2.4 Minimize Nonproductive Crane Time

Note that another useful objective function is to simply minimize the combination of both travel time and delay time. This way the schedules generated minimizes the overall non-productive crane time. This is the actual objective used in the SA improvement heuristic. Given the embedded requirement in the BCS algorithm that all cranes are provided similar workloads, this non-productive crane time is minimized while providing a reasonable makespan for the cranes. BCS's simulated annealing achieves a reasonable makespan by choice of parameter d, and then not swapping jobs between cranes.

The next chapter displays the computational results of the BCS algorithm. The data used to test the algorithm is discussed. Next, recommendations are made for certain parameter values in the algorithm. Finally, the BCS algorithm and the current scheduling strategy employed in the manufacturing facility are compared.

Chapter 4 - Computational Results and Analysis

The purpose of this chapter is to discuss the computational results of the BCS algorithm and provide further analysis into the effects of adjusting certain BCS parameters. The first section explains the data used in the testing of the BCS algorithm. Section 4.2 first discusses the impact of changing certain parameters within the BCS algorithm culminating in recommended values for each. Then, the results of the BCS algorithm are displayed and compared to those obtained using the current process employed in the facility.

4.1 Data

The first data set to be input pertains to the facility parameters. This includes the speed of the cranes, the location of the bridges along the *bayheight* axis, and a standard bridge cross-time. For this manufacturing facility, the cranes move at an average speed of three miles per hour (the speed at which a person walks). There are two sets of three bridges connecting adjacent bays (i.e. three bridges connecting bays A and B, three bridges connecting bays B and C). Each set is evenly spaced along the *bayheight* axis, so their *y*-coordinates are .25, .5, and .75. Whenever a cab is switching bays, one extra minute is added on as the bridge cross-time. The rest of the data pertains to the actual jobs.

Whenever an employee in one of the job-shops places a request for a crane to come pick up a finished item, the time is logged into the facility's system. This time is said to be the release date of the job, and is recorded properly. When the crane actually arrives at the job-shop, the crane operator is supposed to log the time into the system as the start-time of the job, and when the item is dropped off at the next location, the end-time of the job should be recorded.

Due to the high demand of the cranes, operators are more focused on completing jobs rather than logging them into the system properly, especially as the reason for the data collection is beyond the scope of their positions. Thus, most of the time completed jobs are not logged into the system by the crane operators until they have a break in their demand (usually occurring towards the end of the shift). When this occurs the jobs are logged in as started, and then immediately completed, implying the job's process time to be 0-1minute. Other jobs are logged into the system as being started, but are neglected to be recorded as completed until the next shift, resulting in gross overestimates of the job's actual process time.

In speaking with facility management and crane operators, a reasonable range of a job's process time was established to be anywhere from 5 minutes to 4 hours. Any process time outside of the established range (5min.-4 hrs.) is randomly changed to a time falling within the range. Choosing to do this instead of simply throwing the entire job out allows the data to keep the request-time distribution's integrity.

The starting and ending locations of some jobs also had to be manipulated to be usable. The manufacturing facility has a coordinate system laid out over the entire floor. When a crane is requested, the starting and ending locations of the job should be given as an *(x,y)*-coordinate pair. In the data provided, roughly half of the jobs' locations are instead described by a single word, which could pertain to several different areas throughout the facility, each ranging in size and containing multiple coordinate locations. For these jobs, the coordinates were randomly selected while keeping the proportion of cross-bay jobs intact. This strategy is justified by the fact that the manufacturing plant is in the middle of changing its layout of the facility, and therefore currently lacks a consistent flow of materials through the plant.

It is worth noting that the facility is currently installing an active Radio Frequency Identification (RFID) system that is able to track all crane movements and job requests with an accuracy of 10 feet. This RFID system takes the task of logging crane moves and requests into the system away from the crane operators and should greatly improve the accuracy of the data collected. Once this accurate data is collected, a more precise evaluation of the BCS algorithm is possible and should be pursued.

4.2 Results

As explained in previous chapters, the goal of the BCS algorithm is to minimize the non-productive crane time with a reasonable makespan. The requirement of obtaining a reasonable makespan implies the avoidance of overloading one crane, by ensuring all active cranes have similar workloads. This is achieved through the minimization of the Total Crane Time objective, previously defined in Chapter 3. Before the results are discussed, the process of optimizing the parameters within the BCS algorithm is explained, culminating in a recommended value for each. In order to obtain useful results, both the current scheduling process (referred to as Dispatcher Assignment) and the BCS algorithm are used to schedule jobs over 20 simulated 8-hour (one shift) periods. The results displayed in sections 4.2.2 and 4.2.3 are the averages of the 20 shifts. The final section of the chapter comments on the statistical relevance of the comparison between the results of the two strategies used.

4.2.1 Optimizing BCS Parameters

There are three parameters needing to be optimized in the BCS algorithm. The first of which is the threshold parameter, d. Recall this is the maximum allowable difference between the sums of the estimated process times of two cranes in the same bay. The other two parameters deal with the Simulated Annealing heuristic. The first is the annealing temperature, A, and the second is the number of iterations, I, when the heuristic completes searching for superior solutions. Because these two parameters directly affect the performance of each other, they are instead optimized simultaneously while keeping the value of the final acceptance probability, A^I, first at 0.0001, and then at 0.001.

In order to optimize these parameters, the BCS algorithm is used to schedule jobs over five simulated 8-hour periods with several different values of each parameter. The values displayed in the tables are the averages over the five shifts. The number of active cranes is always 6, as this is typically the number of cranes in operation at the facility. The number of "fixed" jobs is always equal to 3. When first optimizing *d*, the default values of the annealing temperature, *A*, and the number of annealing iterations, *I*, are 0.912 and 100, respectively. These values correspond to a final acceptance probability of 0.0001. The results of optimizing *d* are displayed below in Table 1.

Table 1 - Threshold parameter, *d*

	Threshold Parameter *d*					
	100	**150**	**200**	**250**	**300**	**350**
Total Delay Time	936.1	1,351.2	825.3	942.8	752.4	975.8
Total Travel Time Between Jobs	1,959.8	1,980.4	1,773.9	1,854.2	1,986.8	1,790.2
C_{max}	1,306.8	1,398.6	1,174.5	1,371.5	1,117.5	1,264.8
Total Crane Time	5,696.9	6,142.1	5,403.2	5,591.6	5,543.3	5,566.4

The optimal value for the threshold parameter *d* is 200. This value provides the best solution, minimizing almost all of the measured outputs. Setting the value to 300 does provide a solution with a C_{max} finishing 57 minutes sooner than the one obtained by setting *d* to 200. However, the savings in the non-productive crane time, and thus the overall crane time, in the solution obtained with *d* = 200, is 140 minutes. Choosing the value of 200 sacrifices an hour's worth of savings on a single crane for over two hours' worth of savings across all the cranes. When optimizing the other parameters, *d* is always set to 200.

The next parameter set to optimize is the annealing temperature and number of annealing iterations. Again, these two are optimized simultaneously while keeping the value of the final acceptance probability, A^I, first equal to 0.0001, and then 0.001. Notice the average run time is now also recorded with the standard output. The run time plays a role in deciding which combination of values to use, as a solution that is only slightly better than another one, but has a much longer computation time may not be considered optimal. This depends on the user's maximum allowable run time. Table 2 displays the results of optimizing the annealing parameters. Notice the value of the annealing temperature, *A*, is displayed in parentheses directly below the corresponding number of iterations, *I*, required to produce the desired final acceptance probability.

Table 2 – Annealing parameters with A^I = 0.0001

	Iterations (Temperature)					
	100 (.912)	**1,000 (.9908)**	**10,000 (.9991)**	**20,000 (.9995)**	**30,000 (.9997)**	**50,000 (.9998)**
Total Delay Time	825.3	448.9	329.6	1,426.2	206.58	354.7
Total Travel Time Between Jobs	1,773.9	1,884.2	1,864.2	1,754.5	1,711.32	1,835.1
C_{max}	1,174.5	1,324.8	1,020.8	1,691.3	1,039.6	1,143.5
Total Crane Time	5,403.2	5,132.9	4,968.9	5,950.5	4,667.5	4,964.8
Run Time	0.67 s	6.00 s	57.83 s	118.69 s	180.75 s	289.4 s

While one generally expects to see an improved solution with a larger amount of iterations, this is not always the case because the annealing temperature is different for each set of iterations. This is a random process, and as the annealing temperatures differ, so to do the navigation choices of the heuristic through the solution space. This is evidenced in the table where the absolute best solution is found after 30,000 iterations. The solution found in 50,000 iterations is very similar to the one found after only 10,000. This leads to the recommendation of setting $A = .9991$ and $I = 10{,}000$. Although a slightly better solution was found with $A = .9997$ and $I = 30000$, the run time more than tripled.

The next table shows possible value combinations for the annealing parameters, keeping the final acceptance probability equal to 0.001. This larger probability corresponds to increased chances of accepting an improving solution. Table 3 displays the results.

Table 3 - Annealing parameters with $A^I = 0.001$

	Iterations (Temperature)					
	100 (.9333)	**1,000 (.9931)**	**10,000 (.9993)**	**20,000 (.9997)**	**30,000 (.9998)**	**50,000 (.9999)**
Total Delay Time	558.4	664.7	587.8	579	523.7	604.5
Total Travel Time Between Jobs	1,886.6	1,851.5	1,894.2	1,874.4	1,734.76	1,794.4
C_{max}	1,083.8	1,444.7	1,135.4	1,106.9	1,063.6	1,239.7
Total Crane Time	5,241.6	5,310.2	5,249.1	5,208.4	5,031.1	5,162.5
Run Time	0.76 s	6.49 s	57.22 s	121.34 s	260.23 s	292.47 s

All of the solutions found in this table are inferior to the one found by setting $A = .9991$ and $I = 10{,}000$. Therefore, these are optimal values, and are used when comparing the BCS algorithm to the current scheduling strategy.

4.3 Comparison of BCS vs. Current System

Currently, the facility employs a crane dispatcher to decide which crane a job is assigned to and in what order jobs are completed. The dispatcher's main strategy is to assign each job to the closest crane, and have the cranes process these jobs in order of release date. Occasionally, a job is completed out of the release-date order due to a higher priority, but because the jobs are randomly created for this model, there is no loss of modeling accuracy if the jobs are always scheduled using the first-in-first-out (FIFO) technique. The current process is simulated multiple times over 20 8-hour periods using several typical values of the number of active cranes. The results are displayed in Table 4 below.

Table 4 - Dispatcher results

	5 cranes	**6 cranes**	**7 cranes**	**Average**
Total Delay Time	301.8	670.5	1,049.4	673.9
Total Travel Time Between Jobs	2,799.8	2,830.6	2,893.2	2,841.2
C_{max}	1,914.4	1,578.3	1,606.2	1,699.6
Total Crane Time	6,022.1	6,421.7	6,971.9	6,471.9

The exact same data was used as input for the BCS algorithm. When assigning jobs to cranes, the threshold parameter, *d*, has a value of 200 minutes, while the job-transfer iteration limit, *NI*, is set to 40. Within the Simulated Annealing heuristic, the annealing temperature, *A*, has a value of 0.9991, and the number of annealing iterations is set to 10,000. Table 5 shows the results below.

Table 5 - BCS results

	5 cranes	**6 cranes**	**7 cranes**	**Average**
Total Delay Time	264.1	430.9	654.9	450.0
Total Travel Time Between Jobs	1,726.1	1,830.0	1,811.1	1,789.1
C_{max}	1,515.1	1,179.7	1,171.5	1,288.8
Total Crane Time	4,889.4	5,185.0	5,450.6	5,175.0

Table 6 provides a summary of the differences between BCS and the current system. It is evident that BCS is far superior, with an average improvement of 24% in the makespan, 36% in nonproductive time, and 20% in the total crane time. BCS should enable the manufacturer to eliminate at least one crane each shift or increase production by 20%. Since these results are randomly generated, the question remains as to whether or not these results are based off of lucky data or they are statistically significant.

Table 6 - Comparison of averages

	Dispatcher	BCS
Total Delay Time	673.9	450.0
Total Travel Time Between Jobs	2,841.2	1,789.1
C_{max}	1,699.6	1,288.8
Total Crane Time	6,471.9	5,175.0

A paired *t*-test was applied to each set of simulations. The null hypothesis is that the means of the makespan objective and Total Crane Time objective are equal between BCS and the dispatcher for each of the 5, 6 and 7-crane simulations. This involves a total of 6 statistical tests. Setting α to .01 resulted in rejection of the null hypothesis in each case. Furthermore, no *p* value is greater than 1×10^{-6} as shown in Table 7. Thus, I can statistically reject the null hypothesis and state that there is strong statistical evidence that these means are not equal. Consequently, it can be assumed that BCS provides substantially better solutions and should be implemented immediately at the facility and utilized by other practitioners seeking to improve overhead crane systems.

Table 7 - *p* values

	Total Crane Time	C_{max}
5 cranes	1.94×10^{-14}	9.99×10^{-14}
6 cranes	3.57×10^{-11}	2.04×10^{-8}
7 cranes	9.86×10^{-10}	2.56×10^{-7}

Chapter 5 - Conclusion

This thesis presents the BCS problem. To the extent of the author's knowledge, this is the first OCS problem where cranes are operating across multiple adjacent bays, and are allowed access to each through specific portals. In addition, BCS allows jobs access to multiple cranes. The BCS algorithm developed to optimize the crane schedules takes a unique approach to creating crane schedules through the use of an initial job-assignment heuristic, two continuous time simulations, and a SA improvement heuristic, all working in conjunction to minimize the non-productive crane time while balancing the workload across all active cranes.

Most previous studies and optimization attempts in OCS focus on the use of solution-based heuristics communicating with discrete event simulations. While these strategies proved to be somewhat successful, they are applied to OCS problems that are much less complex than the BCS problem. Previous OCS problems involve a fixed number of cranes with known and consistent move-patterns. The jobs being assigned to cranes in these more basic problems can be classified into a relatively small number of categories based on their consistent beginning and ending locations. This allows for discrete event simulations to provide fairly accurate models of the OC system.

In the BCS problem, jobs do not follow consistent patterns, due to the sheer size of the facility and its ever-changing production requirements. The number of active cranes also varies from shift to shift. For these reasons, discrete event simulation would not provide a useable model for the BCS problem. By instead employing continuous time simulation, the BCS algorithm is able to assess the quality of potential solutions with an accuracy of one tenth of a minute, and it does so with an average computation time of less than one hundredth of a second. The flexibility inherent in continuous time simulations also makes the BCS algorithm easy to apply to other OCS problems representing different manufacturing facilities' overhead crane systems.

When testing the BCS algorithm against the current scheduling strategy employed in the manufacturing facility for which the algorithm was developed, BCS provided a 20% reduction in the total crane time. This is equivalent to completing the same set of jobs in the same amount of time required by the current scheduling strategy, but with one less crane. Furthermore, BCS is shown to be statistically superior to the current policy.

5.1 Future Research

Given the novelty of the BCS problem and subsequent algorithm, there are many opportunities for future research to help expand the functionality and improve the performance of this algorithm. This section discusses some of these topics.

As previously stated, the manufacturing facility around which the BCS problem is modeled is in the process of implementing an active RFID system that should greatly increase the accuracy of the data collected. Once this system is installed, the performance of the BCS algorithm should be reevaluated in comparison to the facility's current scheduling strategy using the improved data. Also, the parameters whose values were optimized in section 4.2.1 should then be adjusted to ensure the best values are used.

An interesting sub-problem arising out of BCS is calculating the value of an additional crane. That is, by how much may the potential production rate of the facility increase with each new crane added to the operation. Certainly each new crane's value is different, leading to the conclusion that there exist an optimal number of cranes for a given set of jobs, in accordance with some overall objective function.

Finally, the BCS algorithm should be evaluated against algorithms developed for the more basic OCS problem. To accomplish this, a set of standard data should first be developed for the OCS problem. Then the BCS algorithm can easily be used to schedule these jobs to cranes operating in a single bay (as is the case in the OCS problem). It would be interesting to see if the combination of a neighborhood based heuristic and continuous time simulation, as found in the BCS algorithm, outperforms the combination of solution-based heuristics with discrete event simulation.

As computers become more powerful, continuous time simulations can be completed in less time, making them useful tools for the evaluation of optimization techniques. Thus, the application of continuous time simulation should be expanded to other classical logistical problems. Some obvious candidate problems are the Meet/Pass Problem in the railroad industry, and the scheduling of Automated Guided Vehicles in large facilities. In both of these problems, knowing the exact location of the vehicles at all times is crucial to successfully optimizing the process.

References

Ahuja, R., Jha, and Liu. (2007). *Innovative railroad blocking optimizer*Innovative Scheduling.

Allen, G. and Dytham, C. (2009). An efficient method for stochastic simulation of biological populations in continuous time. *BioSystems,* (98), 37-42.

Andrews, S. and Bray, D. (2004). Stochastic simulation of chemical reactions with spatial resolutionand single molecule detail. *Institute of Physics Publishing, 1*, 137-151.

Battiti, R. and Tecchiolli, G. (1994). The reactive tabu search. *ORSA Journal on Computing, 6*(2)

Baesens, B., De Backer, M., Haesen, R., Martens, D., Snoeck, M., and Vanthienen, J. (2007). Classification with ant colony optimization. *IEEE Transactions on Evolutionary Computation, 11*(5), 651.

Barricelli, N. Esempi numerici di processi di evoluzione. *Methodos,* , 45-68.

Barricelli, N. (1957). Symbiogenetic evolution processes realized by artificial methods. *Methodos,* , 143-182.

Belew, R., Goodsell, D., Halliday, R., Hart, W., Huey, R., and Morris, G. (1998). Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry, 19*(14), 1639-1662.

Blauwkamp, R. and Hawley, P. (2010). Six-degree-of-freedom digital simulations for missile guidance,navigation,and control. *Johns Hopkins APL Technical Digest, 29*(1), 71.

Capara, A., Galli, L., Monaci, M., and Toth, P. (2006). Train platforming problem. *Informs Annual Meeting: Celebrating the Renaissance of Operations Research,* Pittsburgh, PA.

Chen, S., Evans, J., Newman, T., and Wyglinski, A. (2010). *Genetic algorithm-based optimization for cognitive radio networks*. Worcester, MA: Worcester Polytechnic Institute.

Coolidge, F. and Wynn, T. (2009). *The rise of homo sapiens: The evolution of modern thinking*. Malden, MA: Wiley-Blackwell.

Cordeau, J., Laporte, G., Ropke, S., and Valentini, M. (2008). *Modeling and solving a multimodal routing problem with timetables and time windows*. Italy: University of Calabria.

Crosby, J. (1973). *Computer simulation in genetics*. London: John Wiley & Sons.

Dantzig, G. (1947). Maximization of a linear function of variables subject to linear inequalities. In T. Koopman (Ed.), *Activity analysis of production and allocation* (pp. 339-347). New York: Wiley & Chapman-Hall.

D'Ariano, A., Hansen, I., Pacciarelli, D., and Pranzo, M. (2006). Modeling reordering and local rerouting strategies to solve train conflicts during rail operations. *Informs Annual Meeting: Celebrating the Renaissance of Operations Research*, Pittsburgh, PA.

Deng, G. (2007). Simulation-based optimization. (Doctor of Philosophy (Mathematics and Computation in Engineering), University of Wisconsin-Madison).

Di Salvo, A. and Perozzi, E. (2008). Novel spaceways for reaching the moon: An assessment for exploration. *Celestial Mechanics Dynamic Astrology, 102*, 207.

Domonkos, T. (2010). Computer simulation as a tool for analyzing and optimizing real-life processes. *Management Information Systems, 5*(1), 13-18.

Dorigo, M. and Maniezzo, V. (1991). Distributed optimization by ant colonies. *The First European Conference on Artificial Intelligence,* Paris, France. pp. 134-142.

Fraser, A. (1957). Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Science, 10*, 484.

Fraser, A. and Burnell, D. (1970). *Computer models in genetics*. New York: McGraw-Hill.

Ganapathy, L., Lal, P., Sambandam, N., and Vachajitpan, P. (2009). Heuristic methods for capacitated vehicle routing problem. *Thai VCML, 52,* 19.

Garey, M. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research, 1*(2), 117.

Ge, P., Jin, M., and Wang, J. (2011). An efficient heuristic algorithm for overgead cranes scheduling operations in workshop. *Applied Mathematics & Information Sciences, 6*(3), 1087-1094.

Gelatt, C., Kirkpatrick, S., and Vecchi, M. (1983). Optimization by simulated annealing. *Science, 220*(4598), 671-680.

Glover, F. (1989). Tabu search - part 1. *ORSA Journal on Computing, 1*(103) .

Glover, F. and McMillan, C. (1986). The general employee scheduling problem: An integration of MS and AI. *Computers and Operations Research.*

Goodwin, G., Henriksen, S., Shook, A., and Wang, L. (2009). *Application of a genetic algorithm in crnae movement scheduling*. University of Newcastle: Center for Integrated Dynamics and Control.

Kantorovich, L. (1940). A new method of solving some classes of external problems. *Doklady Akad Sci USSR*, (28), 211-214.

Khoshraftar, M., Seyedabadi, M., and Yaghini, M. (2012). *A population-based algorithm for the railroad blocking problem*. Tehran, Iran: School of Railway Engineering, Iran University of science and Technology.

Lan, W., Ting, C., and Wu, K. (2007). Ant colony system based approaches to the air-express courier's routing problem. *Proceedings of the Eastern Asia Society for Transportation Studies, 6.*

Lieberman, R. and Turksen, I. (2007). *Two-operation crane scheduling problems*. Ontario, Canada: University of Toronto.

Lourenco, R. and Serra, D. (2002). Adaptive search heuristics for the generalized assignment problem. *Mathware & Soft Computing, 9*(2).

Minaei-Bidgoli, B., & Punch, W. (2008). *Using genetic algorithms for data mining optimization in an educational web-based system*. East Lansing,MI: Michigan State University.

Mrotzek, M. and Ossimitz, G. (2008). *The basics of system dynamics: Discrete vs. continuous modeling of time*. Athens,Greece: International Systems Dynamics Conference.

Nowicki, E. "A Fast Taboo Search Algorithm for the Job Shop Problem." *Management Science* (1996): 797.

Ozdamar, L. and Yi, W. (2007). A dynamic logistics coordination model for evacuation and support. *European Journal of Operational Research, 179*, 1177-1193.

Panettier, J., Bassas-alsina, J., and Caignaert, V. (1990). Prediction of crystal structures from crystal chemistry rules by simulated annealing. *Nature, 346*, 343.

Selvi, V. and Umarani, R. (2010). Comparative analysis of ant colony and particle swarm optimization techniques. *International Journal of Computer Applications, 5*(4).

Shah, V. (2008). Time dependent truck routing and driver scheduling problem with hours of service regulations. (Operations Research Master of Science, Northeastern University).

Taha, H. (2007). *Operations research an introduction*. Upper Saddle River,NJ: Pearson Prentice Hall.

Toth, P. and Vigo, D. (2002). Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics, 123*, 487.

Wang, C. (1999). Aircraft autopilot design using a sampled-data gain scheduling technique. (Master of Science, Ohio State University).

Wang, C., Quan, H., and Xu, X. (1999). Optimal design of multiproduct batch chemical process using tabu search. *Comp. Chemical Engineering, 23*, 427.