

Northeastern University

Department of Mechanical and Industrial Engineering

January 01, 2006

Computational complexity of a reverse manufacturing line

Surendra M. Gupta Northeastern University

Seamus M. McGovern Northeastern University

Recommended Citation

Gupta, Surendra M. and McGovern, Seamus M., "Computational complexity of a reverse manufacturing line" (2006). . Paper 30. http://hdl.handle.net/2047/d10009942

This work is available open access, hosted by Northeastern University.



Bibliographic Information

McGovern, S. M. and Gupta, S. M., "Computational Complexity of a Reverse Manufacturing Line", *Proceedings of the SPIE International Conference on Environmentally Conscious Manufacturing VI*, Boston, Massachusetts, pp.1-12, October 1-3, 2006.

Copyright Information

Copyright 2006, Society of Photo-Optical Instrumentation Engineers.

This paper was published in Proceedings of SPIE (Volume 6385) and is made available as an electronic reprint with permission of SPIE. One print or electronic copy may be made for personal use only. Systematic or multiple reproduction, distribution to multiple locations via electronic or other means, duplication of any material in this paper for a fee or for commercial purposes, or modification of the content of the paper are prohibited.

Contact Information

Dr. Surendra M. Gupta, P.E. Professor of Mechanical and Industrial Engineering and Director of Laboratory for Responsible Manufacturing 334 SN, Department of MIE Northeastern University 360 Huntington Avenue Boston, MA 02115, U.S.A.

(617)-373-4846 *Phone* (617)-373-2921 *Fax* gupta@neu.edu *e-mail address*

http://www.coe.neu.edu/~smgupta/ Home Page

Computational Complexity of a Reverse Manufacturing Line

Seamus M. McGovern^a and Surendra M. Gupta^{*b}

^aNational Transportation Systems Center, U.S. DOT, 55 Broadway, Cambridge, MA, 02142 USA; ^bLab. Responsible Manufacturing, Northeastern Univ., 360 Huntington, Boston, MA, 02115 USA

ABSTRACT

Disassembly has recently gained attention in the literature due to its role in product recovery. Disassembly activities take place in various recovery operations including remanufacturing, recycling, and disposal. The disassembly line is the best choice for automated disassembly of returned products. It is therefore important that the disassembly line be designed and balanced so that it works as efficiently as possible. However, finding the optimal balance is computationally intensive with exhaustive search quickly becoming prohibitively large, even for relatively small products, due to exponential growth. In this paper, complexity theory is reviewed and used to prove that the DISASSEMBLY LINE BALANCING PROBLEM is NP-complete, unary NP-complete, and NP-hard, necessitating specialized solution methodologies, including those from the field of combinatorial optimization.

Keywords: Disassembly, disassembly line balancing, combinatorial optimization, complexity theory, product recovery

1. INTRODUCTION

More and more manufacturers are acting to recycle and remanufacture their post-consumed products due to new and more rigid environmental legislation, increased public awareness, and extended manufacturer responsibility. In addition, the economic attractiveness of reusing products, subassemblies and parts instead of disposing of them has further fueled this effort. Recycling is a process performed to retrieve the material content of used and non-functioning products. Remanufacturing, on the other hand, is an industrial process in which worn-out products are restored to like-new conditions. Thus, remanufacturing provides the quality standards of new products with used parts.

In order to minimize the amount of waste sent to landfills, product recovery seeks to obtain materials and parts from old or outdated products through recycling and remanufacturing – this includes the reuse of parts and products. There are many attributes of a product that enhance product recovery; examples include: ease of disassembly, modularity, type and compatibility of materials used, material identification markings, and efficient cross-industrial reuse of common parts/materials. The first crucial step of product recovery is disassembly.

Disassembly is defined as the methodical extraction of valuable parts/subassemblies and materials from discarded products through a series of operations. After disassembly, reusable components are cleaned, refurbished, tested, and directed to inventory for remanufacturing operations. The recyclable materials can be sold to raw-material suppliers, while the residuals are sent to landfills.

This paper first defines the DISASSEMBLY LINE BALANCING PROBLEM (DLBP) mathematically and then provides a review of complexity theory, leading up to three proofs showing that the problem is NP-complete, NP-complete in the strong sense, and NP-hard. All of these are indicative of the requirement to make use of specialized solution techniques, including those from combinatorial optimization. While exhaustive search consistently provides the problem's optimal solution, its time complexity quickly limits its practicality. Combinatorial optimization is an emerging field that combines techniques from applied mathematics, operations research, and computer science to solve optimization problems over discrete structures. These techniques include: greedy algorithms, integer and linear programming, branch-and-bound, divide and conquer, dynamic programming, local optimization, simulated annealing, genetic algorithms, and approximation algorithms.

2. LITERATURE REVIEW

Product recovery involves a number of steps [1], including disassembly [2], [3]. Gungor and Gupta presented the first introduction to the DISASSEMBLY LINE BALANCING PROBLEM [4] and developed an algorithm for solving the

^{*} gupta@neu.edu; phone 1 617 373-4846; fax 1 617 373-2921; URL http://www.coe.neu.edu/~smgupta/

DLBP in the presence of failures with the goal of assigning tasks to workstations in a way that probabilistically minimizes the cost of defective parts [5].

3. NOTATION

The following notation is used in the remainder of the paper:

\leq_P	polynomial-time transformation
\prec	partial order; i.e., x precedes y is written $x \prec y$
CT	cycle time; maximum time available at each workstation
d_k	demand; quantity of part k requested
D	demand rating for a given solution sequence; also, demand bound for the decision version of DLBP
F	measure of balance for a given solution sequence
h_k	binary value; 1 if part k is hazardous, else 0
Н	hazard rating for a given solution sequence; also, hazard bound for the decision version of DLBP
Ι	total idle time for a given solution sequence
j	workstation count (1,, NWS)
k	part identification or sequence position {1,, n}
n	number of part removal tasks
Ν	the set of natural numbers $\{0, 1, 2,\}$
NWS	number of workstations required for a given solution sequence
Р	the set of part removal tasks
PRT_k	part removal time required for kth task
PS_k	k^{th} part in a solution sequence (i.e., for solution $\langle 3, 1, 2 \rangle PS_2 = 1$)
r_k	integer value corresponding to part removal direction
R	direction rating for a given solution sequence; also, direction bound for the decision version of DLBP
R_k	binary value; 0 if part k can be removed in the same direction as part $k + 1$, else 1
ST_j	station time; total processing time requirement in workstation j
V	maximum range for a workstation's idle time
Z	the set of all integers $\{, -2, -1, 0, 1, 2,\}$
\mathbf{Z}^+	the set of positive integers $\{1, 2,\}$

4. THE DLBP MODEL DESCRIPTION

The particular application considered in this paper seeks to fulfill five objectives:

- 1. Minimize the number of disassembly workstations and hence, minimize the total idle time.
 - 2. Ensure the idle times at each workstation are similar.
 - 3. Remove hazardous parts early in the disassembly sequence.
 - 4. Remove high demand parts before low demand parts.
 - 5. Minimize the number of direction changes required for disassembly.

A major constraint is the requirement to provide a feasible disassembly sequence for the product being investigated. The result is an integer, deterministic, multiple-criteria decision making problem with an exponential search space.

Line balancing seeks to achieve "perfect balance" (all idle times equal to zero). When this is not achievable, either Line Efficiency (LE) or the Smoothness Index (SI) is often used as a performance evaluation tool [6]. We use a measure of balance that combines the two and is easier to calculate. SI rewards similar idle times at each workstation, but at the expense of allowing for a large (sub-optimal) number of workstations. This is because SI compares workstation elapsed times to the largest ST_j instead of to CT. LE rewards the minimum number of workstations, but allows unlimited variance in idle times between workstations because no comparison is made between ST_j s. This paper makes use of the balancing method developed by McGovern and Gupta [7]. The balancing method simultaneously minimizes the number of workstations while aggressively ensuring that idle times at each workstation are similar, though at the expense of the generation of a nonlinear objective function. The method is computed based on the number of workstations required as well as the sum of the square of the idle times at all of the workstations. This penalizes solutions where, even though the number of workstations may be minimized, one or more have an exorbitant amount of idle time when compared to the other workstations. It provides for leveling the workload between different workstations on the disassembly line.

Therefore, a resulting minimum performance value is the more desirable solution, indicating both a minimum number of workstations and similar idle times across all workstations. The measure of balance is represented as

$$F = \sum_{j=1}^{NWS} (CT - ST_j)^2$$
(1)

with perfect balance indicated by F = 0. A hazard measure has been developed and is represented as

$$H = \sum_{k=1}^{n} (k \cdot h_{PS_{k}}), \quad h_{PS_{k}} = \begin{cases} 1, hazardous \\ 0, otherwise \end{cases}$$
(2)

The demand measure can be represented as

$$D = \sum_{k=1}^{n} (k \cdot d_{PS_k}), \ d_{PS_k} \in \mathbb{N}, \forall PS_k$$
⁽³⁾

A measure based on a count of the direction changes has also been developed. Part removal directions are expressed as

$$r_{PS_{k}} = \begin{cases} +1, direction : +x \\ -1, direction : -x \\ +2, direction : +y \\ -2, direction : -y \\ +3, direction : +z \\ -3, direction : -z \end{cases}$$
(4)

The direction measure is then represented as

$$R = \sum_{k=1}^{n-1} R_k, \ R_k = \begin{cases} 1, r_{PS_k} \neq r_{PS_{k+1}} \\ 0, otherwise \end{cases}$$
(5)

The overall goal is to minimize each of these measures, with a priority of *F*, then *H*, then *D*, then *R*. McGovern and Gupta [7] provide further explanation of formulae (1) through (5). In DLBP, a solution consists of an ordered sequence of work elements (i.e., tasks, components, or parts). For example, if an ordered sequence of elements consisted of the 8-tuple $\langle 5, 2, 8, 1, 4, 7, 6, 3 \rangle$, then part 5 would be removed first, followed by part 2, then part 8, and so on. Problem assumptions include the following:

- Part removal times are deterministic, constant, and integer (or able to be converted to an integer format).
- Each product undergoes complete disassembly.
- All products contain all parts with no additions, deletions, or modifications.
- Each task is assigned to one and only one workstation.
- The sum of the part removal times of all the parts assigned to a workstation must not exceed CT.
- The precedence relationships among the parts must not be violated.

5. COMPLEXITY THEORY BACKGROUND

In order to attempt to solve the DLBP, its structure and complexity first must be established and understood. It should be noted that a great deal of the explanations in Sections 5 through 9 (with the exception of the three proofs) are not the original work of this paper's authors and can be attributed to Garey and Johnson [8], Karp [9], Papadimitriou and Steiglitz [10], and Tovey [11]. Because of the importance of NP-completeness theory, its occasional misrepresentations in texts and papers, and the seemingly cryptic and extremely terse illustrations traditional in the precise field of mathematics, the bulk of the information presented in the remainder of this paper is offered here exclusively as supplementary information to allow a more full understanding of the three proofs. This is especially necessary since the three proofs presented here are structured in that same succinct format, requiring some degree of interpretation of various details that are often considered to be "obvious" and are typically left to the reader. In fact, this paper uses the

notoriously compact proof format used by Garey and Johnson [8] (which is actually longer and more detailed than that used by Karp [9] in his seminal paper). Even so, additional information is still provided in some of the proofs including the listing of analogous elements in DLBP and MULTIPROCESSOR SCHEDULING prior to *Theorem 2* and the discussion showing DLBP \in NP in *Theorem 2*.

Computational complexity is the measure of how much work – typically measured in time or computer memory – is required to solve a given problem. If the problem is easy, it can probably be solved as a linear program (LP), network model, or with some similar method. If a problem is hard, finding an exact solution is apt to be time intensive or even impractical, requiring the use of enumerative methods or accepting an approximate solution obtained with heuristics.

Though similar in name to the ASSEMBLY LINE BALANCING problem (and more specifically, the simple assembly line-balancing type I problem, a claimed NP-hard combinatorial optimization problem), the DISASSEMBLY LINE BALANCING PROBLEM contains a variety of complexity-increasing enhancements. The general ASSEMBLY LINE BALANCING problem can be formulated as a scheduling problem, specifically as a flow shop. In its most basic form, the goal of DLBP is to minimize the idle times found at each flow shop machine (workstation). Since finding the optimal solution requires investigating all permutations of the sequence of *n* part removal times, there are *n*! possible solutions. The time complexity of searching this space is O(n!). This is known as *factorial complexity* and referred to as *exponential time*. Though a possible solution to an instance of this problem can be verified in polynomial time, it cannot always be optimally solved. This problem is therefore classified as *nondeterministic polynomial* (NP); if the problem could be solved in polynomial time it would then be classified as *polynomial* (P). (More precisely, Karp [9] defines P to be the class of languages *recognizable* – recognition in languages is equivalent to solving in decision problems – in polynomial time by *one-tape deterministic Turing machines* and NP to be the class of languages recognizable in polynomial time by *one-tape nondeterministic* – guessing – *Turing machines*.)

Combinatorial optimization is an emerging field that combines techniques from applied mathematics, operations research, and computer science to solve optimization problems over discrete structures. Combinatorial optimization problems are optimization problems where some c (a cost function that maps feasible points to the real numbers) is optimized over a finite set F (the set of feasible points). The goal is to find $f \in F$ such that c(f) is maximized, minimized, or equal to some value; that is, find an optimal f. Note that there can also be more than one optimal solution, or none. Usually, F is considered to be the set of solutions, and in all cases F is finite. Combinatorial optimization problems are those that can be put in form (F, c), for example the TRAVELING SALESMAN PROBLEM (TSP; given a finite set of cities, edges, and weights, the lowest cost *tour* is desired) is a combinatorial optimization problem where F is the set of the Hamiltonian cycles (i.e., cycles that visit every node exactly once) and c is the total cost of a cycle (which is to be minimized). To solve one of these problems, one must consider every solution, compute their cost functions, and select the best one. This technique is known as the *exhaustive* approach to solving a problem. However, exhaustive search takes n! time, which makes this solution technique impractical for all but the smallest problems.

The basic distinction made between solution techniques is whether they are easy (they take polynomial time $O(x^a)$). where x is indicative of the problem size and a is some constant; this group includes n, $n\log n$, n^2 , n^3 , $10^8 n^4$, etc.) or they are hard (taking exponential time $O(a^x)$; this group includes 2^n , 10^n , $n^{\log n}$, n!, n^n , etc. [10]). (A function g(x) is O(h(x))) whenever there exists a constant a such that $|g(x) \le a \cdot |h(x)|$ for all $x \ge 0$.) Complexity theory is formulated from the theory of *recursive functions* and the definitions and applications of *recursively enumerable sets*. The terminology includes the following. A specific numeric case or class of cases having the same form is known as an instance (I). An instance of a problem is a particular case of the problem with all parameters quantitatively defined. The term *problem* (Π : also L, a reference to its basis in languages from *recursive enumeration*) refers to a class of instances and is defined as a general question to be answered that usually contains variables. It includes a broad description of all parameters and of what properties the answer (solution) must satisfy. A problem is then a general description of the properties of an optimal solution. Therefore, an instance can be thought of as a particular, specific version of the problem. For example, TSP is a problem while an actual list of cities and weights is an instance. An *algorithm* is a general, step-by-step process for solving the problem. An algorithm can solve a problem if it can produce a solution for any instance of that problem. In order to solve a problem, an algorithm must always find the same, optimum (minimum or maximum) solution for a given instance [8]. (This strict definition of an algorithm from the field of mathematics is always providing the same result and one that is always the optimal result – techniques that include exhaustive search, the Simplex algorithm for LP, and dynamic programming for PARTITION - is given here for completeness but is not used throughout the remainder of this paper in favor of the more prevalent definition from the field of computer science of an algorithm as being simply a step-by-step process for solving a problem; that is, not necessarily an optimal solution and not necessarily

a deterministic process.) It is generally of interest to find an algorithm that gives a solution in the most efficient way, where the most efficient algorithm is generally considered to be the fastest. This is due to the fact that the time requirements are most often the measure of whether or not a given algorithm is of any practical use since the time is reflective of the problem size. For every input size, the time complexity function for an algorithm gives the largest amount of time needed by that algorithm to solve an instance of that size. *Time complexity* is defined to be the same size as the worst-case runtime, though in this paper time complexity is generically used to refer to any time study. A *polynomial time algorithm* is one whose time complexity function is O(p(x)) for some polynomial function p with input of size x; this definition includes constant time, that is, O(1). If it cannot be bounded this way, it is called (excluding certain non-polynomial time complexity functions) an exponential time algorithm. As an example of the time differences between the two types of algorithms, a computer that can perform one million operations per second would require 13 minutes to process an instance of size x = 60 for a polynomial time algorithm operating at $O(x^5)$, while taking 366 centuries to process the same size instance but using an exponential time algorithm operating at $O(2^x)$ [8]. For all but the smallest problems, polynomial time algorithms are much more desirable than exponential time algorithms which (it is believed) render a problem *intractable* (where a problem cannot be solved by any known polynomial time algorithm). Exceptions include algorithms that are worst-case exponential time but tend to run much faster in average case, such as the Simplex algorithm for LINEAR PROGRAMMING and branch-and-bound for the KNAPSACK problem. There are two causes for intractability: problems so hard they require an exponential amount of time to obtain a solution, and problems whose solutions are exponential in size (this is typically indicative of an unrealistically defined problem since the solution cannot practically be used due to its size). These definitions assume a *reasonable machine*. Reasonable machines include: personal computers, workstations, supercomputers, etc. They also include Turing machines (a one-tape deterministic Turing machine makes use of a list of n-tuples, entering at the first non-blank position on the tape using the information in an (the) initial state *n*-tuple that also corresponds to the 0, 1, or blank in the initial position on the tape, updates the machine's state, writes a 0, 1, or a blank, and moves right or left on the tape – all per that *n*-tuple's data – one position, then repeats until reaching a point where no *n*-tuple corresponds to the state required and the 0, 1, or blank currently in the position on the tape at which point the machine halts and the newly written sequence on the tape is the final tape output). Machines that are not reasonable include: a machine that can do an infinite number of things at once, a parallel computer with an unlimited number of nodes, quantum computers (ultimate laptop, black hole computer), etc. Subsequent definitions also require reasonable encodings. Reasonable encodings are concise; they do not contain extra information. An example of an encoding that is not reasonable would include an input to TSP that was a listing of all cycles in the graph and their costs. Each cycle could then be checked to see if it is Hamiltonian and then the one with the smallest cost could be selected. The number of cycles is the size of a TSP instance. Therefore, an algorithm can be found that solves TSP in polynomial time with respect to the number of cycles, so TSP would be tractable. However, this encoding is unreasonable – it is exponentially large with respect to the number of vertices. In all, polynomial time complexity assumes: input size x is large enough, constant factors are ignored, instance encoding is reasonable, and the machine is reasonable.

Easy problems include LP, MINIMUM COST NETWORK FLOW, MATCHING, MINIMUM SPANNING TREE, and sorting. Hard problems include INTEGER PROGRAMMING (IP), TSP, and JOB-SHOP SCHEDULING [11]. P denotes the class of easy problems; NP-hard is a larger class than NP-complete and includes NP-complete. Being NP-hard or NP-complete effectively means that it takes a long time to exactly solve all cases of sufficiently large size [11]. The formality of NP-completeness is of special interest since, as put by Garey and Johnson [8], there is a certain theoretical satisfaction in precisely locating the complexity of a problem via a "completeness" result that cannot be obtained with a "hardness" result. Adding a threshold value to an optimization problem converts it to "yes-no" (*decision*) form, which is the standard format used in complexity analysis and NP-completeness proofs. This is due to complexity theory having its basis in recursive enumeration, which includes a function g that only takes two values (which are assumed to be 0 and 1). The yes-no version of a problem is analyzed to classify its optimization version. Therefore, hard problems should stay hard and easy problems should stay easy when converted to yes-no form. That is, the yes-no form, the yes-no version will never be more difficult to solve than the original version. In the class NP, if lucky guesswork is allowed, then the problems can be solved in polynomial time. Most reasonable and practical problems are in NP.

6. NP-COMPLETENESS

Cook provided the foundations for NP-completeness in 1971 [12] by showing that a *polynomial time reduction* (\leq_P) from one problem to another ensures any polynomial time algorithm used on one can be used to solve the other. In addition, Cook emphasized the class of NP decision problems that can be solved on a nondeterministic computer (a fictitious computer that operates in a manner that is not predictable; that is, a computer that guesses solutions). The final significant contribution of that paper was the proof that every problem in NP can be reduced to one particular problem in NP (SATISFIABILITY, or SAT) and the suggestion that other problems in NP may share SAT's property of being the hardest problem in NP. This group of the hardest problems in NP is the class of *NP-complete problems*. (The term "complete" in reference to NP-complete problems was first used by Karp [9] and comes from recursive enumerability and the notion of a language being "complete" for a class with respect to a given type of reducibility if it belongs to the class and every other member reduces to it; Karp defines the term to mean "equivalent" and used it to describe problems are intractable, this has never been proven or disproved. Once determined and proven, the question "is P = NP?" will effect every problem that is NP-complete. If and when answered, it will be known that no NP-complete problem will ever be solved in polynomial time or that all NP-complete problems can be optimally solved.

The theory of NP-completeness is applied only to *decision problems*. A decision problem Π consists of a set *D* of decision instances and a subset $Y \subseteq D$ of yes-instances along with any additional problem structure. Specifying a problem consists of two parts: describing a generic instance (INSTANCE) and stating the yes-no question (QUESTION) asked in terms of the generic instance. While the optimization problem asks for a structure of a certain type that has a minimum (or maximum) cost among all such structures, associating a numerical threshold as an additional parameter and asking whether there exists a structure of the required type having cost no more (less) than that threshold creates the desired decision problem. As long as the cost function is relatively easy to evaluate, the decision problem can be no harder than the corresponding optimization problem. Therefore, even though the theory of NP-completeness is restricted to decision problems, it is known that the optimization version is at least as hard and hence, we can extend the implications of the theory. (Note that many decision problems – including TSP – can also be shown to be no easier than their corresponding optimization problems.) The reason that this theory is restricted to decision problems is due to their formal mathematical correspondence to languages (and ultimately, recursive enumeration) and use of encoding schemes.

Originally formalized with application to combinatorial problems in Karp [9], the language framework is now defined. An *alphabet* Σ is a finite set of symbols (e.g., $\Sigma = \{0, 1\}$). A *string* is a sequence made up of the symbols in the alphabet (e.g., 01101). A language L over an alphabet Σ is any set of strings over Σ (e.g., $L = \{1, 10, 100, 010, 1001, \ldots\}$). The *empty string* is written ε while the set containing all possible strings from Σ is written Σ^* . It is assumed that any problem instance can be encoded as one of the strings in Σ^* , that is, as a binary string. Languages provide a way to easily list all the instances of a problem whose answer is "yes." Languages and problems are closely related. A language is equivalent to a problem; a string is equivalent to an instance; 1s and 0s can be used to encode real numbers where a 1 is equivalent to the answer "yes" and a 0 is equivalent to the answer "no." An encoding scheme is what is used when specifying problem instances. Karp [9] provided a mathematical definition where an encoding $e: D \to \Sigma^*$ of a set $D' \subseteq$ D is recognizable in polynomial time if $e(D') \in \mathbb{P}$. Languages allow for the decision problem to be represented to and solved by a deterministic one-tape Turing machine (and ultimately a nondeterministic one-tape Turing machine). The class NP is intended to isolate decision problems in polynomial time. Given a problem instance I, a nondeterministic algorithm has two stages: a guessing stage where some structure S is proposed, and a checking stage where I and S are used to terminate with "yes," terminate with "no," or compute without halting (the latter two are effectively indistinguishable). The class of NP is then the class of all decision problems that (under reasonable encoding schemes) can be solved by polynomial time nondeterministic algorithms (with the understanding that "solve" refers to verification of a "yes" or "no" answer). As a result, note that $P \subset NP$ though it is generally accepted (but never proven) that $P \neq NP$. Applied from the mathematics of languages, a *polynomial transformation* (\leq_P ; i.e., a polynomial time reduction) implies a function exists that can map the solution space of one problem to another. Karp [9] in one of his definitions explains how the reduction from Π' to $\Pi(\Pi' \leq_P \Pi)$ is actually the process of converting Π to Π' through the use of some function. Formally, $\Pi' \leq_P \Pi$ is a function $g: D' \to D$ where g is computable by a polynomial time algorithm and $\forall I \in D', I \in Y' \Leftrightarrow$ $g(I) \in Y$. $\Pi' \leq_P \Pi$ can be interpreted as meaning that Π is at least as hard as Π' . Also, if Π can be solved by a polynomial time algorithm, so can Π' ; if Π' is intractable, so is Π . \leq_P is an equivalence relation and poses a partial order on the resulting classes of languages (i.e., decision problems). As a result, the class P forms the lowest equivalence class in this partial ordering with the decision problems viewed as the easiest problems computationally. The class of NP-complete

languages (problems) forms another class, the one having the hardest languages (problems) in NP. A language *L* is NPcomplete if (i) $L \in NP$ and (ii) for all other languages *L'* such that $L' \in NP$, $L' \leq_P L$. This prompts the identification of the NP-complete problems as the hardest problems in NP (it is property (ii) by itself that causes problems to be in the class NP-hard). If any NP-complete problem can be solved in polynomial time, then every problem in NP can be solved in polynomial time; if any problem in NP is intractable, then every problem in NP-complete is intractable. (Note that if $P \neq NP$ then there must be problems in NP that are neither solvable in polynomial time nor are they NP-complete, but these are generally not of interest.)

The process of devising an NP-completeness proof for a decision problem Π consists of four steps:

- 1. show that $\Pi \in NP$,
- 2. select a known NP-complete problem Π' ,
- 3. construct a transform g from Π' to Π , and
- 4. prove that *g* is a polynomial transformation.

The main idea in an NP-completeness proof is to model a known NP-complete problem as the problem under consideration (not the reverse, which is what might be done to solve a problem). This is done using a transformation – essentially a computer program – that has an INPUT (an instance of Π' , a known NP-complete problem) and an OUTPUT (an instance of Π , the problem under consideration). The transformation must: map "yes" instances of Π' to "yes" instances of Π , and it must be fast (run in polynomial time in the length of the input instance).

Cook provided the first NP-complete problem (SAT [12]). The six other basic NP-complete problems include: 3-SATISFIABILITY, 3-DIMENSIONAL MATCHING, VERTEX COVER, CLIQUE, HAMILTONIAN CIRCUIT (HC), and PARTITION [8]. These six problems are commonly referred to as the "known NP-complete problems." Also, they are the most commonly used problems for NP-completeness proofs and serve as a basic core of known NP-complete problems. In addition, they all appeared in the original list of 21 NP-complete problems in Karp [9]. Some techniques for proving NP-completeness include restriction, local replacement, and component design [8].

An NP-completeness proof by *restriction* for a given problem $\Pi \in NP$ consists simply of showing that Π contains a known NP-complete problem Π' as a special case. It requires that additional restrictions be placed on the instances of Π so that the resulting restricted problem will be identical to Π' (not necessarily exact duplicates of one another but rather an obvious one-to-one correspondence between their instances that preserves "yes" and "no" answers). That is, considering a problem $\Pi = (D, Y)$ (the set *D* of all instances and the set *Y* of all "yes" instances), $\Pi' = (D', Y')$ is a subproblem of Π if

1. $D' \subseteq D$, and

 $2. Y' = Y \cap D'.$

Instead of transforming a known NP-complete problem to the target problem, an attempt is made to restrict inessential aspects of the target problem until a known NP-complete problem appears (many problems that arise in practice are simply more complicated versions of known NP-complete problems).

With *local replacement*, some aspect of the structure of the known NP-complete problem instance is used to make up a collection of basic units. The corresponding instance of the target problem is obtained by uniformly replacing each basic unit with a different structure, each replacement constituting only a local modification of that structure. This is occasionally augmented by a limited amount of additional structure, known as an *enforcer*, which imposes certain additional restrictions on the ways in which a "yes" answer to the target instance can be obtained. The enforcer acts to limit the target problems instances to those that reflect the known NP-complete problem instance.

Component design proofs are any in which the constructed instance can be viewed as a collection of components, each performing some function in terms of the given instance. The basic idea is to use the constituents of the target problem instance to design certain components that can be combined to realize instances of the known NP-complete problem. Components are joined together in a target instance in such a way that the choices are communicated to property testers, and the property testers then check whether the choices made satisfy the required constraints. Interactions between components occur both through direct connections and through global constraints.

The decision version of DLBP is NP-complete. This can be shown through a proof by restriction to one of the known NP-complete problems (PARTITION). PARTITION is described by:

INSTANCE: A finite set *A* of tasks, a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

QUESTION: Is there a partition $A' \subseteq A$ into two disjoint sets such that $\sum_{a \in A'} s(a) = \sum_{a \in A-A'} s(a)$?

Theorem 1 The decision version of DLBP is NP-complete (verification is omitted for brevity; the decision version of DLBP is easily seen to be solvable in polynomial time by a nondeterministic algorithm, see *Theorem 2*).

INSTANCE: A finite set *P* of tasks, partial order \prec on *P*, task time $PRT_k \in \mathbb{Z}^+$, hazardous part binary value $h_k \in \{0, 1\}$, part demand $d_k \in \mathbb{N}$, and part removal direction $r_k \in \mathbb{Z}$ for each $k \in P$, workstation capacity $CT \in \mathbb{Z}^+$, number $NWS \in \mathbb{Z}^+$ of workstations, difference between largest and smallest idle time $V \in \mathbb{N}$, hazard measure $H \in \mathbb{N}$, demand measure $D \in \mathbb{N}$, and direction change measure $R \in \mathbb{N}$.

QUESTION: Is there a partition of *P* into disjoint sets P_A , P_B ,..., P_{NWS} such that the sum of the sizes of the tasks in each P_X is *CT* or less, the difference between largest and smallest idle times is *V* or less, the sum of the hazardous part binary values multiplied by their sequence position is *H* or less, the sum of the demanded part values multiplied by their sequence position is *D* or less, the sum of the number of part removal direction changes is *R* or less, and it obeys the precedence constraints?

Proof: Clearly the decision version of DLBP is in NP (see *Theorem 2* for details).

PARTITION \leq_P DLBP. Restrict to PARTITION by allowing only instances in which $CT = \frac{\sum_{k=1}^{n} PRT_k}{2} \in \mathbb{Z}^+, V = 0, \prec$ is

 \square

empty, and
$$h_x = h_y$$
, $d_x = d_y$, $r_x = r_y \forall x, y \in P$.

Therefore, the decision version of DLBP is NP-complete.

Informally, PARTITION is a subset of DLBP; that is, some DLBP instances look like PARTITION instances. It can also be viewed that DLBP contains PARTITION within it. Other problems that are formulated similarly to DLBP substantiate the suspicion that this problem is NP-complete. In addition, these similar problems can be easily used to prove DLBP belongs to the class NP-complete. They include problems from the areas of: storage and retrieval (BIN PACKING), mathematical programming (KNAPSACK), sets and partitions (3-PARTITION), and sequencing and scheduling (MULTIPROCESSOR SCHEDULING, PRECEDENCE CONSTRAINED SCHEDULING).

Note that the addition of precedence constraints (i.e., \prec not empty) can both reduce the search space and may provide promise for an effective pseudo-polynomial time algorithm. This observation may be primarily academic since, unless it is a process similar to branch-and-bound or some type of pre-processing is performed on the instance, many search techniques (including the ones in this paper) first consider a solution, then determine if it is feasible. Also, it is the requirement to check every possibility that causes many problems having partial orders (e.g., PARTIALLY ORDERED KNAPSACK) to be NP-complete. In addition, precedence constraints invariably add a layer of complexity to the search, with problems possessing precedence constraints still being found to be NP-complete (primarily sequencing and scheduling problems such as the previously mentioned MULTIPROCESSOR SCHEDULING and PRECEDENCE CONSTRAINED SCHEDULING problems). Note that just determining if a feasible solution to DLBP exists can be seen to be the same problem as DIRECTED HAMILTONIAN CIRCUIT, an NP-complete problem (e.g., transform HC to DIRECTED HAMILTONIAN CIRCUIT by replacing each edge [p, q] in the undirected graph with the two arcs (p, q) and (q, p)). Finally, with DLBP able to be modeled both as a flow shop and as a multi-criteria version of an assembly line, it should come as no surprise that DLBP is NP-complete since it is obviously in NP and the SALB-I problem is regularly described as NP-hard; also, FLOW-SHOP SCHEDULING has been proven to be NP-complete [8].

7. NP-COMPLETENESS IN THE STRONG SENSE

NP-completeness provides strong evidence that a problem cannot be solved with a polynomial time algorithm (the theory of NP-completeness is applied only to decision problems). Many problems that are polynomial solvable differ only slightly from other problems that are NP-complete. 3-SATISFIABILITY and 3-DIMENSIONAL MATCHING are NP-complete while the related 2-SATISFIABILITY and 2-DIMENSIONAL MATCHING problems can be solved in polynomial time. If a problem is NP-complete, some subproblem (created by additional restrictions being placed on the allowed instances) may be solvable in polynomial time. Certain encoding schemes and mathematical techniques can be used on some size-limited cases of NP-complete problems (an example is a dynamic programming approach to PARTITION that results in a search space size equal to the size of the instance multiplied by the log of the sum of each

number in the instance) enabling the solution by what is know as a "pseudo-polynomial time algorithm." In the PARTITION case, this results in a polynomial time solution as long as extremely large input numbers are not found in the instance. Problems for which no pseudo-polynomial time algorithm exists (assuming $P \neq NP$) are known as "NP-complete in the strong sense" (also, *unary* NP-complete – referring to allowance for a non-concise or "unary" encoding scheme notation where a string of *n* 1s represents the number *n* – with an alternative being *binary* NP-complete). Typical limiting factors for NP-complete problems to have a pseudo-polynomial time algorithm include the requirement that the problem be a number problem (vs. a graph problem, logic problem, etc.) and be size constrained, typically in the number of instance elements (*n* in DLBP) and the size of the instance elements (*PRT_k* in DLBP). Formally, an algorithm that solves a problem Π will be called a *pseudo-polynomial time algorithm* for Π if its time complexity function is bounded above as a function of the instance *I* by a polynomial function of the two variables Max[*I*] (problem size) and Length[*I*] (data length, the number of digits required to write each number – note that this can be enormous). A general NP-completeness result simply implies the problem cannot be solved in polynomial time in all the chosen parameters.

KNAPSACK has been shown to be solvable by a pseudo-polynomial time algorithm in a dynamic programming fashion similar to PARTITION. MULTIPROCESSOR SCHEDULING and SEQUENCING WITHIN INTERVALS have both been shown to be NP-complete in the strong sense. The same is true for 3-PARTITION which is similar to PARTITION but has *m* (instead of 2) disjoint sets and each of the disjoint sets has exactly 3 elements. By considering subproblems created by placing restrictions on one or more of the natural problem parameters, useful information about what types of algorithms are possible for use with the general problem can be gleaned.

The decision version of DLBP is NP-complete in the strong sense. This can be shown through a proof by restriction to MULTIPROCESSOR SCHEDULING; note that some authors (e.g., Papadimitriou and Steiglitz [10]) define the problem differently (e.g., they include precedence constraints and require task lengths to be equal). Garey and Johnson [8] describe MULTIPROCESSOR SCHEDULING as:

INSTANCE: A finite set A of tasks, a length $l(a) \in \mathbb{Z}^+$ for each $a \in A$, a number $m \in \mathbb{Z}^+$ of processors, and a deadline $B \in \mathbb{Z}^+$.

QUESTION: Is there a partition $A = A_1 \cup A_2 \cup \ldots \cup A_m$ of A into m disjoint sets such that $\max\left\{\sum_{a \in A_i} l(a) : 1 \le i \le m\right\} \le B$?

In DLBP, NWS is equivalent to m in MULTIPROCESSOR SCHEDULING, P is equivalent to A, CT is equivalent to B.

The following theorem provides the proof:

Theorem 2 The decision version of DLBP is NP-complete in the strong sense.

INSTANCE: A finite set *P* of tasks, partial order \prec on *P*, task time $PRT_k \in \mathbb{Z}^+$, hazardous part binary value $h_k \in \{0, 1\}$, part demand $d_k \in \mathbb{N}$, and part removal direction $r_k \in \mathbb{Z}$ for each $k \in P$, workstation capacity $CT \in \mathbb{Z}^+$, number $NWS \in \mathbb{Z}^+$ of workstations, difference between largest and smallest idle time $V \in \mathbb{N}$, hazard measure $H \in \mathbb{N}$, demand measure $D \in \mathbb{N}$, and direction change measure $R \in \mathbb{N}$.

QUESTION: Is there a partition of *P* into disjoint sets P_A , P_B ,..., P_{NWS} such that the sum of the sizes of the tasks in each P_X is *CT* or less, the difference between largest and smallest idle times is *V* or less, the sum of the hazardous part binary values multiplied by their sequence position is *H* or less, the sum of the demanded part values multiplied by their sequence position is *D* or less, the sum of the number of part removal direction changes is *R* or less, and it obeys the precedence constraints?

Proof: DLBP \in NP. Given an instance, it can be verified in polynomial time if the answer is "yes" by: counting the number of disjoint sets and showing that they are NWS or less, summing the sizes of the tasks in each disjoint set and showing that they are CT or less, examining each of the disjoint sets and noting the largest and the smallest idle times then subtracting the smallest from the largest and showing that this value is V or less, summing the hazardous part binary values multiplied by their sequence position and showing this is H or less, summing the demanded part values multiplied by their sequence position and showing that this is D or less, summing the number of changes in part removal direction and showing that this is R or less, and checking that each task has no predecessors listed after it in the sequence.

MULTIPROCESSOR SCHEDULING \leq_P DLBP. Restrict to MULTIPROCESSOR SCHEDULING by allowing only instances in which V = CT, \prec is empty, and $h_x = h_y$, $d_x = d_y$, $r_x = r_y \forall x, y \in P$.

 \square

Therefore, the decision version of DLBP is NP-complete in the strong sense.

It is easy to see that the BIN-PACKING problem is also similar to DLBP. The BIN-PACKING problem is NP-hard in the strong sense (it includes 3-PARTITION as a special case) which indicates that there is little hope in finding even a pseudo-polynomial time optimization algorithm for it.

8. NP-HARDNESS

The techniques used for proving NP-completeness can also be used for proving problems outside of NP are hard. Any decision problem Π , whether a member of NP or not, to which an NP-complete problem can be transformed will have the property that it cannot be solved in polynomial time unless P = NP. Such a problem Π is NP-hard, since it is at least as hard as the NP-complete problems. Similarly, a search (optimization) problem Π , whether a member of NP or not, is NP-hard if there exists some NP-complete problem Π' that reduces to Π . Furthermore, by the previous association of languages with string relations and decision problems with search problems, it can be said then that all NP-complete languages (i.e., all NP-complete problems) are NP-hard. So NP-complete \subseteq NP and NP-complete \subseteq NP-hard; that is, NP-complete = NP-hard \cap NP. Whenever it is shown that a polynomial time algorithm for the optimization problem can be used to solve the corresponding decision problem in polynomial time, a reduction between them is being made, and hence an NP-completeness result for the decision problem can be translated into an NP-hardness result for the optimization problem. That is, showing an NP-complete decision problem is no harder than its optimization problem, along with the fact that the decision problem is NP-complete, constitutes a proof that the optimization problem is NP-hard. NP-hard is interpreted as meaning "as hard as the hardest problem in NP" though the term has differing meanings in some of the literature (the definition of NP-complete, however, is generally standardized and universally understood).

DLBP is NP-hard. This can be shown in two steps [8] by showing

1. the NP-complete decision problem is no harder than its optimization problem, and

2. the decision problem is NP-complete.

This constitutes a proof that the optimization problem is NP-hard. Note that SALB-I is claimed to be NP-hard [6].

Theorem 3 DLBP is NP-hard.

Proof: Shown in two steps:

The decision version of DLBP is no harder than its optimization version. Both versions require preservation of 1) the precedence constraints. The optimization version of DLBP asks for a sequence that has the minimum difference between largest and smallest idle times among all disjoint sets, the minimum sum of hazardous part binary values multiplied by their sequence position, the minimum sum of demanded part values multiplied by their sequence position, and the minimum number of part removal direction changes. The decision version includes numerical bounds V, H, D, and R as additional parameters and asks whether there exists NWS disjoint sets with a difference between largest and smallest idle times no more than V, a sum of hazardous part binary values multiplied by their sequence position no more than H, a sum of demanded part values multiplied by their sequence position no more than D, and a number of part removal direction changes no more than R. So long as the difference between largest and smallest idle times, the sum of hazardous part binary values multiplied by their sequence position, the sum of demanded part values multiplied by their sequence position, and the number of part removal direction changes is relatively easy to evaluate, the decision problem can be no harder than the corresponding optimization problem. If a minimum difference between the largest and smallest idle times, a minimum sum of hazardous part binary values multiplied by their sequence position, a minimum sum of demanded part values multiplied by their sequence position, and a minimum number of part removal direction changes could be found for the DLBP optimization problem in polynomial time, then the associated decision problem could be solved in polynomial time. This would require finding the minimum idle time and the maximum idle time from all NWS subsets, compute the difference, and compare that difference between largest and smallest idle times to the given bound V_{i} sum all the hazardous part binary values multiplied by their sequence position and compare that sum to the given bound H; sum all the demanded part values multiplied by their sequence position and compare that sum to the given bound D; and sum all of the part removal direction changes and compare that sum to the given bound R. Therefore, the decision version of DLBP is no harder than its optimization version.

2) From *Theorems 1* and 2, the decision version of DLBP is NP-complete.

Therefore, from 1) and 2), DLBP is NP-hard.

Things that make a problem hard include: dividing work or resources up evenly, making sequencing decisions that depend on current and past positions, splitting a set of objects into subsets where each must satisfy a constraint, finding a

largest substructure that satisfies some property, maximizing or minimizing intersections or unions of sets, interactions among three or more objects or classes of constraints [11] (the BIN-PACKING problem is NP-hard, though there are fast, approximate solutions such as First-Fit, First-Fit-Decreasing, and Best-Fit [8]). Problems that are not susceptible to dynamic programming are called unary NP-hard or NP-hard in the strong sense (defined similarly to strong NPcompleteness). For example, dynamic programming works well for 2-MACHINE LINE BALANCING; it works but is slower by a factor of *n* for 3-MACHINE LINE BALANCING; it is impractical for 5-MACHINE LINE BALANCING. Theoretically, if *m* is allowed to vary and get large, the *m*-machine problem is like 3-PARTITION or BIN PACKING and is NP-hard in the strong sense; although the problem is not NP-hard in the strong sense for any fixed value of m, m =5 is large, practically speaking. To summarize the previously described classes informally, a problem is in the class NP if a guessed answer can be checked against the instance's threshold(s) in polynomial time. A problem is in the class NPhard if a known NP-complete problem can be transformed to it in polynomial time. A problem is in the class NPcomplete if both these requirements are met.

9. SOLVING NP-COMPLETE PROBLEMS

There are normally considered to be two general categories of techniques for addressing NP-complete problems [8].

The first category includes approaches that attempt to improve upon exhaustive search as much as possible. If realistic cases that are modeled as IPs are not very large, commercial math-programming software may be effective. Among the most widely used approaches to reducing the search effort of problems with exponential time complexity are those based on branch-and-bound or implicit enumeration techniques. These generate "partial solutions" within a tree-structured search format and utilize powerful bounding methods to recognize partial solutions that cannot possibly be extended to actual solutions, thereby eliminating entire branches of the search in a single step. Other approaches include dynamic programming, cutting plane methods, and Lagrangian techniques. Dynamic programming can solve KNAPSACK, PARTITION, and similar problems as long as the instance is not too large. However, other NP-hard problems including 3-PARTITION and BIN PACKING are not susceptible to quick solution by dynamic programming. Dynamic programming of a KNAPSACK problem that is made up of integers and having a knapsack size of *a* uses a table that is *x* by *a* + 1 in size, requiring O(ax) work. This algorithm is pseudo-polynomial time because it is only fast when *a* is small. As discussed previously, if the dynamic programming algorithm is polynomial in the parameters of problem size and data length (note, however, that if *a* were an 80-digit number, the required table could not fit into the combined disk memory of all Internet-linked computers [11]), the algorithm is pseudo-polynomial time. In practice, dynamic programming algorithms are apt to run out of memory before they bog down with CPU usage [11].

The second category allows for the attainment of sub-optimal (though ideally, near-optimal or optimal) solutions – to optimization problems only – in a reasonable amount of time through the use of heuristics. These are especially effective when one is unsure of the model (for example, when the objective can only be approximately quantified), when the data is not accurate, when an exact solution is not required, when the problem is transient or unstable and robustness in the solution method becomes important, when the instances are enormous or the problem is very difficult in practice (for example, JOB SHOP SCHEDULING or the QUADRATIC ASSIGNMENT PROBLEM), when solution speed is critical, or when the necessary expertise is not available. Heuristic approaches are frequently based on intuitive rules-of-thumb. These methods tend to be problem specific. The most widely applied technique is that of a "neighborhood search" where a pre-selected set of local operations is used to repeatedly improve an initial solution, continuing until no further local improvements can be made and a "locally optimum" solution has been obtained. Heuristic methods designed via this and other approaches have often proved successful in practice, although a considerable amount of problem-specific fine-tuning is usually required in order to achieve satisfactory performance. As a consequence, it is rarely possible to predict how well such methods will perform by formally analyzing them beforehand. Instead, these heuristics are usually evaluated and compared through a combination of empirical studies and common-sense arguments.

Papadimitriou and Steiglitz [10] further expand on these two categories to allow for six alternatives to address NP-complete problems. In order to solve an exponential time problem, they propose one of the following approaches:

Approximation: application of an algorithm that quickly finds a sub-optimal solution that is within a certain (known) range or percentage of the optimal solution,

Probabilistic: application of an algorithm that provably yields good average runtime behavior for a distribution of the problem instances,

Special cases: application of an algorithm that is provably fast if the problem instances belong to a certain special case,

Exponential: strictly speaking, pseudo-polynomial time algorithms are exponential; also, many effective search techniques (e.g., branch-and-bound, Simplex) have exponential worst-case complexity,

Local Search: recognized to be one of the most effective search techniques for hard combinatorial optimization problems, local (or *neighborhood*) search is the discrete analog of "hill climbing,"

Heuristic: application of an algorithm that works reasonably well on many cases, but cannot be proven to always be fast or optimal.

McGovern and Gupta [13], [14], [15], [16] demonstrate solutions to the DLBP using promising search techniques from several of these approach categories using some of the few DLBP instances available at this time.

10. CONCLUSIONS

In this paper, relevant aspects of complexity theory are reviewed and used to prove that the DISASSEMBLY LINE BALANCING PROBLEM is NP-complete, NP-complete in the strong sense, and NP-hard, necessitating specialized solution methodologies, including those from the field of combinatorial optimization.

REFERENCES

- A. Gungor and S. M. Gupta, "Issues in Environmentally Conscious Manufacturing and Product Recovery: A Survey," *Computers and Industrial Engineering*, 36(4), 811-853, 1999.
- [2] L. Brennan, S. M. Gupta, and K. N. Taleb, "Operations Planning Issues in an Assembly/Disassembly Environment," *International Journal of Operations and Production Planning*, 14(9), 57-67, 1994.
- [3] S. M. Gupta and K. N. Taleb, "Scheduling Disassembly," International Journal of Production Research, 32, 1857-1866, 1994.
- [4] A. Gungor and S. M. Gupta, "Disassembly Line in Product Recovery," *International Journal of Production Research*, 40(11), 2569-2589, 2002.
- [5] A. Gungor and S. M. Gupta, "A Solution Approach to the Disassembly Line Problem in the Presence of Task Failures," *International Journal of Production Research*, 39(7), 1427-1467, 2001.
- [6] E. A. Elsayed and T. O. Boucher, *Analysis and Control of Production Systems*, Prentice Hall, Upper Saddle River, NJ, 1994.
- [7] S. M. McGovern and S. M. Gupta, "Performance Metrics for End-Of-Life Product Processing," *Proceedings of the* 17th Annual Production & Operations Management Conference, CD-ROM, Boston, MA, 2006.
- [8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, NY, 1979.
- [9] R. M. Karp, "Reducibility Among Combinatorial Problems," Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, 85-103, Plenum Press, New York, NY, 1972.
- [10] C. H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Dover Publications, Mineola, NY, 1998.
- [11] C. A. Tovey, "Tutorial on Computational Complexity," Interfaces, 32(3), 30-61, 2002.
- [12] S. A. Cook, "The Complexity of Theorem-Proving Proceedures," Proceedings 3rd Annual Association for Computing Machinery Symposium on Theory of Computing, 151-158, New York, NY, 1971.
- [13] S. M. McGovern and S. M. Gupta, "Ant Colony Optimization for Disassembly Sequencing With Multiple Objectives," *The International Journal of Advanced Manufacturing Technology*, in press.
- [14] S. M. McGovern and S. M. Gupta, "A Balancing Method and Genetic Algorithm for Disassembly Line Balancing," *European Journal of Operational Research*, in press.
- [15] S. M. McGovern and S. M. Gupta, "Local Search Heuristics and Greedy Algorithm for Balancing the Disassembly Line," *The International Journal of Operations and Quantitative Management*, 11(2), 91-114, 2005.
- [16] S. M. McGovern and S. M. Gupta, "Deterministic Hybrid and Stochastic Combinatorial Optimization Treatments of an Electronic Product Disassembly Line," *Applications of Management Science*, K. D. Lawrence, G. R. Reeves, and R. Klimberg, 12, Elsevier Science, North-Holland, Amsterdam, 2006.