An Algorithm for Computing the Optimal Cycle Time of a Printed Circuit Board Assembly Line

Dušan M. Kodek and Marjan Krisper
University of Ljubljana, Faculty of Computer and Information Science
Tržaška 25, 1000 Ljubljana, Slovenia
E-mail: duke@fri.uni-lj.si

Keywords: combinatorial optimization, integer programming, minimax approximation

Received: December 24, 2002

We consider the problem of optimal allocation of components to a printed circuit board (PCB) assembly line which has several nonidentical placement machines in series. The objective is to achieve the highest production throughput by minimizing the cycle time of the assembly line. This problem can be formulated as a minimax approximation integer programming model that belongs to the family of scheduling problems. The difficulty lies in the fact that this model is proven to be NP*-complete. All known algorithms that solve the* NP*-complete problems are exponential and work only if the number of variables is reasonably small. This particular problem, however, has properties that allow the development of a very efficient type of branch-and-bound based optimal algorithm that works for problems with a practically useful number of variables.*

1 Introduction

The problem of optimal allocation of components to placement machines in a printed circuit board (PCB) assembly line is *NP*-complete and is often considered too difficult to solve in practice. This opinion is supported by the experience with the general integer programming programs that are typically very slow and do not produce solutions in a reasonable time. It is therefore not surprising to see many attempts of replacing the optimal solution with a near-optimal one. The reasoning goes as follows: A near-optimal solution is often good enough and is usually obtained in a significantly shorter time than the optimal solution. Although this is true in many cases, it does not hold always. The difficulty with the near-optimal methods is that they, as a rule, do not give an estimate of closeness to the optimal solution. This means that a significantly better optimal solution, about which the user knows nothing, may exist. Given a choice, the user would probably always choose the optimal solution provided that it can be obtained in a reasonable time.

This paper challenges the opinion that the optimal solution is too difficult to compute. An algorithm that takes advantage of the special properties of the minimax approximation optimal allocation problem is developed. This optimal algorithm is much faster than the general integer programming approach mentioned above. The algorithm produces, in most practical cases, the optimal solution in a time that is similar to the time needed for near-optimal methods. Because of its exponential nature, it will of course fail in the cases when the number of variables is large. But it should be noted that the algorithm practically always produces one or more suboptimal solutions which can be used in such cases. These suboptimal solutions are comparable to those obtained by the near-optimal methods like local search, genetic algorithms, or knowledge based systems. Or in other words, the user can only gain if the optimal algorithm is used.

Let us start with an investigation of the PCB assembly line problem. The cycle time T of a PCB assembly line is defined as the maximum time allowed for each machine (or station) in the assembly line to complete its assembly tasks on the board. This time becomes important when the quantity of PCBs is large: A minor reduction in the cycle time can result in a significant cost and time savings. Moreover, a PCB line in the problem has several non-identical placement machines. As a board contains hundreds of surface mounted components in different shapes, sizes, and patterns, different placement machines in the line are installed to cope with different components. The line efficiency depends on the combination of the machine types. Due to the costly placement machines, the optimization of the assembly process can significantly increase the competitiveness of the production.

Many factors affect the efficiency of the PCB assembly, namely customer orders [1], component allocation [2], PCB grouping [3], component sequence [4], and feeder arrangement [5]. Different algorithms have been developed to optimize different factors in PCB assembly [6], [7]). The genetic algorithm technique is one of the heuristic methods that has been used recently to find a near-optimal solution [8].

Machine M_i	Placement times t_{ij} for component type j							Setup time s_i
	1	2	3	4	5	6	7	
1	0.3	0.7	0.7	0.5	∞	∞	∞	11.0
2	0.7	1.2	1.5	1.6	1.5	1.5	2.1	14.7
3	2.3	3.8	3.5	3.5	2.7	3.3	4.3	14.7
Number of type j components per board	324	37	12	5	7	5	4	

Table 1: An example of a PCB assembly line with 3 different placement machines and 7 different component types per board. The placemet times t_{ij} for different components and machines and the setup times s_i are in seconds.

2 Formulation of the problem

When a board is assembled on a production line the board's components are grouped and allocated to appropriate placement machines in order to achieve a high output of the line. The next machine can begin its tasks only after the previous machine has completed the placement of all components that were allocated to it. After the board passes through all the machines, the component placement process is completed. It is clear that the slowest task dictates the performance of the assembly line.

There are two important differences between the traditional assembly line problem and this PCB assembly line problem. First, unlike the traditional assembly line, the precedence of operations in the PCB assembly is not important and can be ignored. The second difference concerns the assembly times for the same component on different machines. Due to various types and configurations of the placement machines, different machines have different times for placement of the same kind of component. The components are usually of a surface mounted type, although this is not important here. An example from Table 1 is used to make the problem easier to understand. This example is the same as the one used in [8] and will allow the comparison of our optimal algorithm to the near-optimal one.

A PCB assembly line with three different placement machines M_1, M_2, M_3 and a board with seven types of components is used in the example. The placemet times t_{ij} for different components and machines are given in the Table 1. If a machine cannot handle a particular type of component, its placement time is assigned to be infinite (∞). The infinity is used here for simplicity of notation only — it is replaced by a large positive number for computation. In addition to the time that is needed to place a component there is also a setup time s_i for each of the machines M_i. The machine needs this time every time a new board arrives for its positioning and placement preparation. Finally, a total number of each type of a component per board c_j is also given.

Obviously, there are many possible ways of allocating the components j to the placement machines M_i. Each of them leads to its own cycle time T. The question is how to allocate the components in such a way that the assembly line has the best performance. The PCB assembly line cycle time T is formally defined as the maximum time needed by one of the machines M_i, $i = 1, 2, \cdots, m$, to complete the placement of the components allocated to it. Clearly, the time interval between two finished boards coming out of the assembly line is equal to T which means that the number of boards produced in a given time span is proportional to $1/T$. This number can be increased by allocating the components to the machines in such way that T is reduced. A mathematical model that describes this situation can now be given.

Suppose that there are m non-identical placement machines M_i in a PCB assembly line and that a board with n types of components is to be assembled on this line. It takes t_{ij} units of time to place the component of type j on a machine M_i. In addition, each machine M_i has a setup time s_i. There are exactly c_j components of type j per board. The component allocation problem can be formulated as

$$T_{opt} = \min_{x_{ij}} \max_{i=1,2,\ldots,m} \left(s_i + \sum_{j=1}^{n} t_{ij} x_{ij} \right) , \qquad (1)$$

subject to

$$\sum_{i=1}^{m} x_{ij} \;=\; c_j, \quad j = 1, 2, \ldots, n\,, \qquad (2)$$

$$x_{ij} \;\geq\; 0 \quad \text{and integer}\,. \qquad (3)$$

The solution of this problem is the optimal cycle time T_{opt} and the optimal allocation variables $x_{ij}^{(opt)}$. The variable x_{ij} gives the number of components of type j that are allocated to machine M_i. Constraints (2) ensure that all of the components will be allocated. The components are indivisible and (3) ensures that x_{ij} are positive integers. Note that t_{ij} and s_i are by definition positive[2].

3 Complexity of the problem

The problem (1)–(3) is a combination of assignment and flowshop scheduling problems [9]. It is *NP*-complete for

[2]The times t_{ij} and s_i can be arbitrary positive real numbers. It is easy to reformulate the problem and change t_{ij} and s_i into arbitrary positive integers. This does not change the complexity of the problem.

$n \geq 2$. Proving the *NP*-completness is no too difficult. First, it is trivial to show that the problem is in *P*. Second, it is possible to show that the well known PARTITION problem can be polynomially transformed into (1)–(3) [10]. Since PARTITION is *NP*-complete, so is our problem.

A typical approach to solving this problem is to treat it as a general mixed integer linear programming problem. The minimax problem (1)–(3) is reformulated as

$$\begin{aligned} &\min_{x_{ij}} T_{opt}\,, \\ &T_{opt} - s_i - \sum_{j=1}^{n} t_{ij}x_{ij} \geq 0,\ \ i = 1, 2, \ldots, m, \\ &\sum_{i=1}^{m} x_{ij} = c_j,\ \ j = 1, 2, \ldots, n, \\ &x_{ij} \geq 0 \quad \text{and integer.} \end{aligned} \tag{4}$$

All algorithms that are capable of solving this problem optimally work by starting with the noninteger problem where the variables x_{ij} can be any positive real number. Additional constraints are then gradually introduced into the problem and these constraints eventually force the variables x_{ij} to integer values. Many instances of suitably reformulated subproblems of the form (4) must be solved before the optimal solution is found.

An advantage of the formulation (4) is that general mixed integer programming programs can be used to solve it. Unfortunately, this advantage occurs at the expense of the computation time. The general programs use the simplex algorithm to solve the subproblems. The simplex algorithm is very general and slow since it does not use any of the special properties of the minimax problem. All these properties are lost if the original problem (1)–(3) is converted into the general problem.

The fact that the general problem (4) is so slow has led to the development of suboptimal heuristic algorithms that search for a near-optimal solution. These algorithms are faster and often good enough. The difficulty is that a significantly better optimal solution may exist which these algorithms do not find. It is the purpose of this paper to develop an optimal algorithm that does not use the generalized formulation (4). The algorithm takes advantage of the special properties of the minimax problem (1)–(3). It avoids using the simplex algorithm completely which leads to a much faster solution.

4 The lower bound theorem

The basic idea of our algorithm is to use a lower bound for T_{opt} as a tool that leads to the solution. This lower bound must be computed for each of the subproblems that appear within the branch-and-bound process. It must take into account the fact that some of the subproblem's variables x_{ij} are known integers. To derive it, let us assume that the subproblems's variables $x_{ij}, j = 1, 2, \ldots, k-1$, are known integers for all i. In addition, some, but not all, of the variables x_{ik} may also be known integers. Let I_k be the set of indices i that correspond to the known integers x_{ik}. The subproblem's variables can be formally described as

$$x_{ij} = \begin{cases} x_{ij}^I,\ j = 1, \ldots, k-1,\ i = 1, \ldots, m \\ x_{ij}^I,\ j = k,\ i \in I_k \\ x_{ij},\ j = k,\ i \notin I_k \\ x_{ij},\ j = k+1, \ldots, n,\ i = 1, \ldots, m, \end{cases} \tag{5}$$

where k can be any of the indices $1, 2, \ldots, n$. Notation x_{ij}^I is used to describe the variables that are already known integers. The remaining variables x_{ij} are not yet known. The number of indices in the set I_k lies in the range 0 to $m-2$. If there were $m-1$ known integers x_{ik}^I the constraint (2) gives the remaining variable which contradicts the assumption that not all of the variables x_{ik} are known. The index k changes to $k+1$ when all x_{ik} are known integers.

Definition (5) assumes that a certain rule is used to introduce the constraints which force the variables x_{ij} to integer values. This rule is simple: For every index k it is necessary to constrain x_{ik} to known integers x_{ik}^I for all $i, i = 1, 2, \ldots, m$, before k can change. The rule follows from the structure of constraints given by (2) and is needed to derive the lower bound theorem. There is no problem with this rule because the branch-and-bound method, on which our algorithm is based, allows complete freedom of choosing the variable x_{ij} that is to be constrained next. The indices k can be selected in any order. A simple ascending order $k = 1, 2, \ldots, n$, is used in (5). This also applies to the case when the problem is first reordered along the indices j in a way that gives the fastest rate of lower bound increase. Such a reordering is used in our algorithm.

To simplify the notation, let us first use the known integers x_{ij}^I and redefine s_i into s_i' as

$$s_i' = \begin{cases} s_i + \sum_{j=1}^{k} t_{ij}x_{ij}^I,\ \ i \in I_k \\ s_i + \sum_{j=1}^{k-1} t_{ij}x_{ij}^I,\ \ i \notin I_k\,. \end{cases} \tag{6}$$

Similarly, the known integers x_{ik}^I (if any) are used to redefine c_k into c_k' as

$$c_k' = c_k - \sum_{i \in I_k} x_{ik}^I\,. \tag{7}$$

The lower bound on T_{opt} over all possible not yet known variables x_{ij} is the most important part of our algorithm. It is developed along the lines used in a related integer polynomial minimax approximation problem that appears in a digital filter design [11], [12] and is given in the following theorem.

Theorem 1 *Let T_{opt} be the minimum cycle time corresponding to the optimal solution of the problem (1)–(3) in which some of the variables are known integers defined by*

(5). Then T_{opt} is bounded by

$$T_{opt} \geq \max_{j=k+1,\ldots,n} \left(\frac{c_j + \sum_{i=1}^{m} \frac{s'_i}{t_{ij}} + p_j + q_j}{\sum_{i=1}^{m} \frac{1}{t_{ij}}} \right) \tag{8}$$

where

$$\begin{aligned} p_j &= \sum_{\substack{r=k+1 \\ r\neq j}}^{n} c_r \min_{i=1,2,\ldots,m} \left(\frac{t_{ir}}{t_{ij}} \right), \\ q_j &= c'_k \min_{i\notin I_k} \left(\frac{t_{ik}}{t_{ij}} \right), \quad j = k+1,\ldots,n\,. \end{aligned} \tag{9}$$

Proof: Let h be a number that satisfies

$$h \geq \begin{cases} s_i + \sum_{j=1}^{k-1} t_{ij} x^I_{ij} + \sum_{j=k+1}^{n} t_{ij} x_{ij}, & i \in I_k \\ s_i + \sum_{j=1}^{n} t_{ij} x^I_{ij} + \sum_{j=k}^{n} t_{ij} x_{ij}, & i \notin I_k\,. \end{cases} \tag{10}$$

Note that h is a lower bound for T_{opt} if we can prove that (10) holds over all possible not yet known values x_{ij}. Using (6) eq. (10) is simplified

$$h \geq \begin{cases} s'_i + \sum_{j=k+1}^{n} t_{ij} x_{ij}, & i \in I_k \\ s'_i + \sum_{j=k}^{n} t_{ij} x_{ij}, & i \notin I_k\,. \end{cases} \tag{11}$$

It follows from (11) that variables x_{ij} can be expressed as

$$\begin{aligned} x_{ij} &\leq \frac{h}{t_{ij}} - \frac{s'_i}{t_{ij}} - \sum_{\substack{r=k+1 \\ r\neq j}}^{n} \frac{t_{ir}}{t_{ij}} x_{ir},\ i \in I_k, j = k+1,\ldots,n, \\ x_{ij} &\leq \frac{h}{t_{ij}} - \frac{s'_i}{t_{ij}} - \sum_{\substack{r=k \\ r\neq j}}^{n} \frac{t_{ir}}{t_{ij}} x_{ir},\ i \notin I_k,\ j = k,\ldots,n. \end{aligned} \tag{12}$$

Adding all x_{ij} by index i and using (2) and (7) gives

$$c_j \leq \sum_{i=1}^{m} \frac{h}{t_{ij}} - \sum_{i=1}^{m} \frac{s'_i}{t_{ij}} - \sum_{\substack{r=k+1 \\ r\neq j}}^{n} \sum_{i=1}^{m} \frac{t_{ir}}{t_{ij}} x_{ir} - \sum_{i\notin I_k} \frac{t_{ik}}{t_{ij}} x_{ik}, \quad j = k+1,\ldots,n\,, \tag{13}$$

and the lower bound for h can now be written as

$$h \geq \frac{c_j + \sum_{i=1}^{m} \frac{s'_i}{t_{ij}} + \sum_{\substack{r=k+1 \\ r\neq j}}^{n} \sum_{i=1}^{m} \frac{t_{ir}}{t_{ij}} x_{ir} + \sum_{i\notin I_k} \frac{t_{ik}}{t_{ij}} x_{ik}}{\sum_{i=1}^{m} \frac{1}{t_{ij}}}, \quad j = k+1,\ldots,n\,. \tag{14}$$

All the terms in (14) are positive. This means that h is a lower bound over all variables if the lowest possible values of the terms containing variables x_{ir} and x_{ik} are used. The variables x_{ir} are subject to

$$\sum_{i=1}^{m} x_{ir} = c_r\,, \quad r = k+1,\ldots,n\,. \tag{15}$$

It is quite easy to see that the sum containing x_{ir} is bounded by

$$\sum_{\substack{r=k+1 \\ r\neq j}}^{n} \sum_{i=1}^{m} \frac{t_{ir}}{t_{ij}} x_{ir} \geq \sum_{\substack{r=k+1 \\ r\neq j}}^{n} c_r \min_{i=1,2,\ldots,m} \left(\frac{t_{ir}}{t_{ij}} \right) = p_j, \quad j = k+1,\ldots,n, \tag{16}$$

since it is obvious that a minimum is obtained if x_{ir} is given the value c_r for index i that corresponds to the lowest of the factors t_{ir}/t_{ij} while all other x_{ir} are set to zero. Similarly, the variables x_{ik} are subject to

$$\sum_{i\notin I_k} x_{ik} = c'_k\,, \tag{17}$$

and the sum containing x_{ik} is bounded by

$$\sum_{i\notin I_k} \frac{t_{ik}}{t_{ij}} x_{ik} \geq c'_k \min_{i\notin I_k} \left(\frac{t_{ik}}{t_{ij}} \right) = q_j\,, \quad j = k+1,\ldots,n\,. \tag{18}$$

Equations (16) and (18) are used in the definitions (9) and this completes the proof. □

Note that the Theorem 2 does not include the lower bound for the case $k = n$. The following trivial lower bound, which holds for all k, can be used in this case

$$T_{opt} \geq \max_{i\notin I_k} \left(s'_i + t_{ik} x_{ik} \right), \quad k = 1,\ldots,n. \tag{19}$$

Note also that index $j = k$ was not used in the derivation of the Theorem 1. The equivalent of (13) for $j = k$ is

$$c'_k \leq \sum_{i\notin I_k} \frac{h}{t_{ik}} - \sum_{i\notin I_k} \frac{s'_i}{t_{ik}} - \sum_{r=k+1}^{n} \sum_{i\notin I_k} \frac{t_{ir}}{t_{ik}} x_{ir}\,. \tag{20}$$

When I_k is not empty all x_{ir} in the sum over $i \notin I_k$ can be zero and still satisfy (15). The lowest possible sum containing x_{ir} is obviously zero in this case. This gives an additional lower bound

$$T_{opt} \geq \frac{c'_k + \sum_{i\notin I_k} \frac{s'_i}{t_{ik}}}{\sum_{i\notin I_k} \frac{1}{t_{ik}}}\,. \tag{21}$$

This lower bound is almost always much lower than the one given by (8). It can included in the algorithm to bring a small decrease in computing time which is on the order of 1%.

By choosing $k = 0$ one can use (8)–(9) to compute the lower bound over all possible values of variables x_{ij}. Applying this to the example from the Table 1 gives $T_{opt} \geq 96.084$. But there is more — the theorem plays a central role in our algorithm because it eliminates the need to use the simplex algorithm for solving the subproblems within the branch-and-bound process.

5 Application of the lower bound theorem

The usefulness of the Theorem 1 is based on the following observation: The problem of finding the all-integer solution that gives the lowest cycle time T_{opt} can be replaced by the problem of finding the all-integer solution that has the lowest lower bound for T_{opt}. Both approaches obviously lead to the same solution since T_{opt} equals its lower bound when all variables x_{ij} are integers.

This observation, however, is not enough. A new constraint must be introduced on one of the variables $x_{ik}, i \notin I_k$, at each branch-and-bound iteration. This constraint cannot be made on the basis of the Theorem 1 alone and requires additional elaboration.

To see how the lower bound depends on x_{ik} let us define the parameters $T_L(j,k)$ as

$$T_L(j,k) = \frac{c_j + \sum_{i=1}^{m} \frac{s'_i}{t_{ij}} + p_j + \sum_{i \notin I_k} \frac{t_{ik}}{t_{ij}} x_{ik}}{\sum_{i=1}^{m} \frac{1}{t_{ij}}}, \qquad (22)$$

where $j = k+1, \ldots, n$, and $k = 1, \ldots, n-1$. The $T_L(j,k)$ are simply (14) rewritten in a slightly different way. The Theorem 1 lower bound (8) in which the variables x_{ik} are left is now equal to

$$T_{opt} \geq \max_{j=k+1,\ldots,n} T_L(j,k)\,. \qquad (23)$$

This lower bound does not include the case $k = n$. This is easily corrected if (19) is included. To simplify notation we first define parameters $T_I(i,k)$ as

$$T_I(i,k) = s'_i + t_{ik} x_{ik}, \quad k = 1, \ldots, n, \qquad (24)$$

and define the new lower bound $T_{opt} \geq T_{LB}(k)$

$$T_{LB}(k) = \max\left(\max_{i \notin I_k} T_I(i,k), \max_{j=k+1,\ldots,n} T_L(j,k) \right). \qquad (25)$$

The $T_{LB}(k)$ are defined for $k = 1, \ldots, n$ (where $T_L(j,n) = 0$). They include $T_I(i,k)$ for all k even if it is strictly needed only for $k = n$. There is a good reason for that because the T_I lower bound sometimes exceeds the T_L lower bound. This can occur when the values of t_{ij} differ by several orders of magnitude as is the case in the example from Table 1 where a large positive t_{ij} is used instead of ∞. Although the algorithm works if T_I is used for $k = n$ only, experiments show that it is usually faster if it is used for all k.

The lower bound $T_{LB}(k)$ (25) is the basis of our algorithm. It is a linear function of the variables $x_{ik}, i \notin I_k$, and, as mentioned before, a new constraint must be introduced on one of them at each branch-and- bound iteration.

Let $i_c, i_c \notin I_k$, be the index of the variable $x_{i_c k}$ that is selected for constraining. Selection of the index i_c is simple — any of the indices $i, i \notin I_k$, can be used as i_c. It is more difficult to find the value $x^*_{i_c k}$ that will be used in the branch-and-bound iteration to constrain the selected variable to integers $x^I_{i_c k}$ which are the nearest lower and upper neighbours of $x^*_{i_c k}$. The $x^*_{i_c k}$ must be a number that gives the lowest possible lower bound $T_{LB}(k)$ over all possible values of the not yet known variables $x_{ik}, i \notin I_k$, and $x_{ij}, i = 1, \ldots, m, j = k+1, \ldots, n$. Or in other words, the $x^*_{i_c k}$ must be at the global minimum of $T_{LB}(k)$.

It is important to understand why $x^*_{i_c k}$ must be at the global minimum of $T_{LB}(k)$. It is because our algorithm uses the property that $T_{LB}(k)$ is a linear function of the variables x_{ik} and is therefore also convex. The convex property is crucial for the success of our algorithm since it ensures that every local optimum is also global. The algorithm uses this property by stopping the search along a variable in the branch-and-bound process when $T_{LB}(k)$ exceeds the current best solution T_u. This, however, can be used only if $x^*_{i_c k}$ is such that $T_{LB}(k)$ does not decrease when an arbitrary integer is added to $x^*_{i_c k}$. The $x^*_{i_c k}$ at the global minimum certainly satisfies this condition.

A great advantage of using the lower bound comes from the fact that the lower bound $T_{LB}(k)$ in (25) depends only on the variables $x_{ik}, i \notin I_k$, and is independent of the remaining variables $x_{ij}, i = 1, \ldots, m, j = k+1, \ldots, n$. This means that the number of variables is significantly reduced in comparison with the general approach (4). Solution of the minimax problem

$$\min_{\substack{x_{ik} \\ i \notin I_k}} \max\left(\max_{i \notin I_k} T_I(i,k), \max_{j=k+1,\ldots,n} T_L(j,k) \right), \qquad (26)$$

$$\sum_{i \notin I_k} x_{ik} = c'_k\,, \qquad x_{ik} \geq 0\,, \qquad (27)$$

gives the nonnegative numbers x^*_{ik} that give the global minimum of $T_{LB}(k)$ for a given k.

A complication arises when k changes to $k+1$ because the solution of (26)–(27) for $k+1$ depends not only on x^*_{ik+1} but also on x^*_{ik} (through s'_i). The problem is that x^*_{ik} are not at the global minimum of $T_{LB}(k+1)$. It is possible that the minimum of (26) for $k+1$ decreases if different x^*_{ik} are used. An error can occur if this is ignored because the algorithm stops the search along a variable if the minimum is $> T_u$ when in fact a lower value for $T_{LB}(k+1)$ exists. It is obvious that this error cannot occur if the minimum $T_{LB}(k+1) \leq T_{LB}(k)$.

The following corrective procedure is used in the algorithm when the minimum $T_{LB}(k+1) >$ minimum $T_{LB}(k)$. It consists of adding +1 and/or -1 to the $x^*_{i_c k}$ that was used

as the last constraint. Using the new $x^*_{i_c k}$ we simply recompute $T_{LB}(k)$ and solve again (26)–(27) for $k+1$. If $\max(T_{LB}(k), T_{LB}(k+1))$ decreases we continue in that direction until it stops decreasing or until (27) is violated ($T_{LB}(k)$ increases when the original $x^*_{i_c k}$ changes). The corrected $x^*_{i_c k}$ is a solution of

$$\min_{x_{i_c k},\, x_{ik+1}} \max\left(T_{LB}(k), T_{LB}(k+1)\right) . \qquad (28)$$

It is used to replace the original and this eliminates the possibility of error. Note that it is not necessary to correct the remaining variables x^I_{ik} even if they were not derived from the global minimum of $T_{LB}(k+1)$. This is because the branch-and-bound process ensures that all values of x^I_{ik} will be tried as long as their $T_{LB}(k)$ is lower than T_u. Additional details about the implementation of (28) are given in step 6 of the algorithm in section 7.

The minimax problem (26)–(27) must be solved many times within the branch-and-bound process and it is extremely important to have an efficient method that gives its solution. Most of the computing time in our algorithm is spent on solving this problem. The method that is used to solve it is worth a detailed description.

6 Solving the discrete linear minimax problem

The number of variables x_{ik} in (26)–(27) is equal to the number of indices $i, i \notin I_k$. Let $m', 1 \le m' \le m$, be this number and let $R(i), i = 1, \ldots, m'$, be the indices not in I_k. Equation (26) contains m' terms T_I and $n-k$ terms T_L. The total number of terms n' is equal to

$$n' = n + m' - k\,, \qquad m' \le n' \le n + m' . \qquad (29)$$

It helps to rewrite (26) using a new index v

$$\min_{x_{R(i)k}} \max\left(\max_{v=1,\ldots,m'} T_I(R(v),k), \max_{v=m'+1,\ldots,n'} T_L(v',k) \right), \qquad (30)$$

where $v' = v + k - m'$. Because of the sum constraint in (27) there are only $m'-1$ independent variables; the m'-th variable can be expressed as

$$x_{R(m')k} = c'_k - \sum_{i=1}^{m'-1} x_{R(i)k}. \qquad (31)$$

The minimax problem (26)–(27) can now be reformulated into a more general form

$$\min_{x_{R(i)k}} \max_{v=1,\ldots,n'} \left(f_v + \sum_{i=1}^{m'-1} \Phi_{vi} x_{R(i)k} \right), \qquad (32)$$

$$\sum_{i=1}^{m'-1} x_{R(i)k} \le c'_k\,, \quad x_{R(i)k} \ge 0\,. \qquad (33)$$

Definitions of terms f_v and Φ_{vi} are somewhat tedious though they follow directly from (22) and (24)

$$f_v = \begin{cases} s'_{R(v)}\,, & v = 1, \ldots, m'-1 \\ s'_{R(v)} + t_{R(v)k} c'_k\,, & v = m' \\ \dfrac{c_{v'} + \sum_{r=1}^{m} \frac{s'_r}{t_{rv'}} + p_{v'} + \frac{t_{R(m')k}}{t_{R(m')v'}} c'_k}{\sum_{r=1}^{m} \frac{1}{t_{rv'}}}\,, & v > m' \end{cases} \qquad (34)$$

$$\Phi_{vi} = \begin{cases} t_{R(i)k} \text{ if } i = v,\ 0 \text{ if } i \ne v, & v = 1,\ldots,m'-1 \\ -t_{R(m')k}\,, & i = 1,\ldots,m'-1,\ v = m' \\ \dfrac{\frac{t_{R(i)k}}{t_{R(i)v'}} - \frac{t_{R(m')k}}{t_{R(m')v'}}}{\sum_{r=1}^{m} \frac{1}{t_{rv'}}}, & i = 1,\ldots,m'-1,\ v > m' \end{cases} \qquad (35)$$

for $v = 1, \ldots, n'$ and $i = 1, \ldots, m'-1$.

The process of solving (32)–(33) is simplified greatly by the theorem that gives the necessary and sufficient conditions for the variables $x^*_{R(i)k}, i = 1, \ldots, m'-1$, that minimize (32). The general version of the theorem is given in [15]. It is repeated here in the form that applies to our problem.

Theorem 2 *The variables $x^*_{R(i)k}, i = 1, \ldots, m'-1$, are the optimal solution of the minimax problem (32)–(33) if and only if the following holds*

$$\min_{z_i} \max_{v \in V_{max}(x^*)} \sum_{i=1}^{m'-1} \Phi_{vi}(z_i - x^*_{R(i)k}) = 0\,, \qquad (36)$$

over all numbers $z_i, i = 1, \ldots, m'-1$, that satisfy

$$\sum_{i=1}^{m'-1} z_i \le c'_k\,, \quad z_i \ge 0\,. \qquad (37)$$

The set $V_{max}(x^*)$ contains those of the indices $v, v = 1, \ldots, n'$, at which the maximum is obtained. That is

$$\max_{v=1,\ldots,n'} \left(f_v + \sum_{i=1}^{m'-1} \Phi_{vi} x^*_{R(i)k} \right) = f_v + \sum_{i=1}^{m'-1} \Phi_{vi} x^*_{R(i)k}\,, \quad v \in V_{max}(x^*)\,. \qquad (38)$$

Only the indices $v, v \in V_{max}(x^*)$, that give the extremal values of the function (38) are used in the Theorem 2. The theorem says that $x^*_{R(i)k}$ is the optimal solution if there are no numbers z_i for which (36) is lower than zero. To show how this can be used to solve (32)–(33) let us assume that we have a set of numbers $x^*_{R(i)k}$ and would like to check if they are optimal. Depending on $V_{max}(x^*)$ and Φ_{vi} there are two mutually exclusive cases:

1. The set $V_{max}(x^*)$ contains at least two indices v_1 and v_2 for which the following holds

$$\Phi_{v_1 i}\Phi_{v_2 i} \leq 0\,, \quad i = 1, \ldots, m' - 1\,. \tag{39}$$

It is easy to see that the numbers z_i that give (36) lower than zero cannot exist. This is because of the opposite signs of $\Phi_{v_1 i}$ and $\Phi_{v_2 i}$ for all i. Any set of numbers z_i that is different from $x^*_{R(i)k}$ makes (36) greater than zero for at least $v = v_1$ or $v = v_2$. Thus, according to the Theorem 2, $x^*_{R(i)k}$ are optimal.

2. The set $V_{max}(x^*)$ does not contain two indices v_1 and v_2 for which (39) holds (this is always true if $V_{max}(x^*)$ contains only one index v). This means that there exists a set of indices I_p, containing at least one index i, for which

$$\Phi_{v_1 i}\Phi_{v_2 i} > 0,\ i \in I_p,\ v_1, v_2 \in V_{max}(x^*), \tag{40}$$

holds for any pair of indices v from $V_{max}(x^*)$. Or in other words, for each $i \in I_p$ the Φ_{vi} are nonzero and have the same signs for all $v \in V_{max}(x^*)$. Let us assume that there are numbers $z_i, i \in I_p$, that satisfy (37) and give

$$\sum_{i \in I_p} \Phi_{vi} z_i < \sum_{i \in I_p} \Phi_{vi} x^*_{R(i)k},\ v \in V_{max}(x^*). \tag{41}$$

These numbers, together with $z_i = x_{R(i)k}$ for $i \notin I_p$, obviously make (36) lower than zero. The numbers $x^*_{R(i)k}$ are therefore not optimal if such z_i exist. They exist almost always — the only exception occurs if the following holds

$$\sum_{i \in I_p} \Phi_{vi} x^*_{R(i)k} = \min_{z_i} \sum_{i \in I_p} \Phi_{vi} z_i\,, \tag{42}$$

for some $v, v \in V_{max}(x^*)$. It is clear that (41) cannot be satisfied in this case because the $x^*_{R(i)k}$ sum is already the lowest possible. The lowest possible sum in (42) is easy to compute by using $z_i = 0$ for $\Phi_{vi} > 0$ and $z_i = c'_k$ for the most negative of $\Phi_{vi} < 0$. This means that it is also easy to check if $x^*_{R(i)k}$ are optimal.

Using (39)–(42) it becomes straightforward to solve (32)–(33). A starting solution for $x^*_{R(i)k}$ is selected and checked as described above. If it is found optimal, we have a solution. If not, one of the variables $x^*_{R(i_1)k}, i_1 \in I_p$, is tried; if it can change towards zero (if $\Phi_{vi_1} > 0$) or towards c'_k (if $\Phi_{vi_1} < 0$) without violating (33), it leads to an improved solution. It is ignored otherwise and a new variable is tried. The set I_p always contains at least one index i that leads to an improved solution.

The new value of $x^*_{R(i_1)k}$ is computed by trying all $v_1, v_1 \notin V_{max}(x^*)$, and solving

$$f'_{v_1} + \Phi_{v_1 i_1} x^*_{R(i_1)k} = f'_v + \Phi_{v i_1} x^*_{R(i_1)k},\ v \in V_{max}(x^*), \tag{43}$$

where f'_v are defined as

$$f'_v = f_v + \sum_{\substack{i=1 \\ i \neq i_1}}^{m'-1} \Phi_{vi} x^*_{R(i)k},\ v = 1, \ldots, n'. \tag{44}$$

Each of the equations (43) gives a possible new value for $x^*_{R(i_1)k}$. The one that is the least different from the current value must be used because the set $V_{max}(x^*)$ changes at that value. The new $x^*_{R(i_1)k}$ must of course also satisfy (33). Replacing $x^*_{R(i_1)k}$ with the new value gives a new solution $x^*_{R(i)k}, i = 1, \ldots, m' - 1$, for which the whole process is repeated until the optimal solution is found.

Selecting a good starting solution is important because it reduces the number of iterations. Our algorithm uses a solution that is found by choosing $x^*_{R(i)k} = c'_k$ (the remaining $x^*_{R(i)k}$ are zero) for $i = 1, \ldots, m'$, and computing the lower bound $T_{LB}(k)$ for each of them. The choice that gives the lowest $T_{LB}(k)$ is the starting solution. This starting solution is often optimal; when it is not, it usually takes only one or two iterations to find the optimum. Note that the search for the optimal $x^*_{R(i)k}$ is not necessary if the starting $T_{LB}(k)$ is lower than the lower bound (21). In such cases the algorithm simply uses the starting solution.

Having the optimal variables $x^*_{R(i)k}, i = 1, \ldots, m' - 1$, it remains to select the one that will be used as the new constraint. This is done by computing the products

$$t_{R(i)k} x^*_{R(i)k}, \quad i = 1, \ldots, m'\,, \tag{45}$$

where (31) is used to compute the remaining variable $x^*_{R(m')k}$. The index $R(i)$ that gives the highest product is selected as i_c. The reasons for this choice is obvious: The highest of products (45) is most likely to give the largest increase of the lower bound $T_{LB}(k)$.

7 The algorithm

The algorithm is based on the well known branch-and-bound method which is described in detail in many textbooks (see, for example, [13] or [14]). We assume that the reader is familiar with this method and continue with the description of the algorithm.

An important part of the branch-and-bound method is the branch-and-bound tree. Each node in the tree represents a subproblem that has some of the variables constrained to integers. Information that is stored at each node must contain the following: The node's lower bound $T_{LB}(k)$, index k, the size of set I_k (it is equal to $m - m'$), the indices i in I_k, integer variables $x^I_{ij}, j = 1, \ldots, k$, and the noninteger variable $x^*_{i_c k}$ that will be used as the next constraint (together with the index i_c). The efficient organization of the tree is important. It does not, however, influence the results of the algorithm and will not be discussed here. The algorithm is described in the following steps:

1. Set $k = 0$ and use (8)–(9) to compute

$$T_L(j,0) = \frac{c_j + \sum_{i=1}^{m} \frac{s'_i}{t_{ij}} + p_j + q_j}{\sum_{i=1}^{m} \frac{1}{t_{ij}}}, \qquad (46)$$

for $j = 1, 2, \ldots, m$. Sort the lower bounds $T_L(j,0)$ in the ascending order. The problem parameters t_{ij} and c_j are reordered accordingly. It is assumed from here on that $j = 1$ corresponds to the lowest $T_L(j,0)$, $j = 2$ to the next higher $T_L(j,0)$, and so on. The reasons for this reformulation of the problem are simple: We wish to eliminate the indices j that give the lowest contribution to the total lower bound $T_{LB}(k)$ and at the same time keep the indices that give the highest contribution to the total lower bound. Several other strategies for selecting the indices j were tested; none performed better over a large class of problems.

2. Set the current best solution T_u to ∞ (a large positive number). The corresponding variables $x_{ij}^{(u)}$ can be set to anything — they will be replaced by one of the solutions quickly. The index u indicates that T_u is an upper bound on T_{opt}. The alternative is to use some heuristic construction and compute a near-optimal starting solution T_u. We found that this is not really necessary because the algorithm quickly produces good near-optimal solutions.

3. Create the root node. This is done by making $k = 1$, $m' = m$ (this makes the set I_k empty), and solving the problem (32)–(33) as described by (36)–(45). The resulting information is stored in the branch-and-bound tree. Initialize the branching counter N to zero.

4. Choose the branching node by searching through the nodes of the branch-and-bound tree. Go to step 8 if no nodes with $T_{LB}(k) < T_u$ are found or if the tree is empty. Add 1 to the branching counter N and choose the branching node according to the following rule: If N is odd, choose the node with the lowest $T_{LB}(k)$, otherwise choose only among the nodes that contain the largest number of integer variables x_{ij}^I and select the one that has the lowest $T_{LB}(k)$. This branching strategy is a combination of the *lowest lower bound* and *depth first* strategies and is used to get many of the near-optimal solutions as fast as possible. This is especially important for large problems with several hundred variables x_{ij}.

5. Two subproblems are created from the branching node by fixing the node's variable $x_{i_c k}^*$ to integers

$$x_{i_c k}^I = \lfloor x_{i_c k}^* \rfloor, \qquad (47)$$

$$x_{i_c k}^I = \lfloor x_{i_c k}^* \rfloor + 1, \qquad (48)$$

where $\lfloor x_{i_c k}^* \rfloor$ denotes the nearest lower integer to $x_{i_c k}^*$. The integers $x_{i_c k}^I$ must of course conform to (27). If $x_{i_c k}^I$ in (48) does not, discard this subproblem (subproblem (47) is never discarded because $x_{i_c k}^*$ satisfies (33)). The number of noninteger variables x_{ik} is reduced by 1

$$m' \leftarrow m' - 1. \qquad (49)$$

If $m' \geq 2$ go to step 6. Otherwise there is only one noninteger variable x_{ik} left. Its integer value is already determined because (27) gives

$$x_{i_c k}^I + x_{ik}^I = c'_k, \qquad (50)$$

and x_{ik}^I is easily computed. All variables x_{ik} are known integers x_{ik}^I, $i = 1, 2, \ldots, m$. Because of this the index k is incremented as described by the definition (5)

$$k \leftarrow k + 1, \qquad (51)$$

The new set I_k is made empty ($m' = m$). If $k \leq n$, go to step 6. Otherwise we have a case where all of the subproblem's variables x_{ij} are integer. This is a *complete integer solution* and the cycle time T is simply computed as

$$T = \max_{i=1,2,\ldots,m} \left(s_i + \sum_{j=1}^{n} t_{ij} x_{ij}^I \right). \qquad (52)$$

If $T < T_u$, we have a new best solution; the current T_u is set to T and the current best solution $x_{ij}^{(u)}$ is replaced by x_{ij}^I. The branch-and-bound tree is searched and all nodes with $T_{LB}(k) \geq T_u$ are removed from the tree. Go to step 7.

6. Each of the non-discarded subproblems from step 5 is solved. The already known integers are taken into account by computing s'_i and c'_k using (6) and (7). Equations (34) and (35) are used next to compute f_v and Φ_{vi} and the problem (32)–(33) is solved as described by (36)–(45). The results are $T_{LB}(k)$ and $x_{i_c k}^*$. If $T_{LB}(k) \geq T_u$ ignore this subproblem since it obviously cannot lead to a solution that is better than the current best T_u. Otherwise if $m' = 2$ and $k < n$ do the corrective procedure (28) and replace $x_{i_c k}^*$ and $T_{LB}(k)$ with the new values. The newly computed $T_{LB}(k)$ will in most cases be greater than that of the branching node. This growth is not monotone and it is possible that the new $T_{LB}(k)$ is lower. Since the lower bound cannot decrease we use the branching node's $T_{LB}(k)$ as the subproblem's $T_{LB}(k)$ in such cases. The subproblem information containing $x_{i_c k}^*$ and $T_{LB}(k)$ is stored as a new node in the branch-and-bound tree.

7. The subproblem in the branching node from step 4 is modified (the root node is an exception — it is simply removed from the branch-and-bound tree and we go to step 4). The branching subproblem is modified by

Machine M_i	Allocation x_{ij} of components							Assembly time on machine M_i
	1	2	3	4	5	6	7	
1	274	0	2	5	0	0	0	97.1
2	50	37	2	0	0	0	0	97.1
3	0	0	8	0	7	5	4	95.3
Number of type j components per board	324	37	12	5	7	5	4	

Table 2: One of the 10 equivalent optimal solutions of the cycle time problem given in example Table 1. The solution was obtained with the algorithm described in this paper.

changing the integer variable x^I_{lk} that was created last. The modification is equal to

$$x^I_{lk} \leftarrow \begin{cases} x^I_{lk} - 1 \text{ if } x^I_{lk} \text{ was created by (47)} \\ x^I_{lk} + 1 \text{ if } x^I_{lk} \text{ was created by (48).} \end{cases} \quad (53)$$

This of course means that each node in the branch-and-bound tree must also contain information about the integer variable that was created last and about the way it was created (either by (47) or (48)). The branching node is removed from the tree if the new $x^I_{lk} < 0$ or if $x^I_{lk} > c'_k$ and we go to step 4. Otherwise the modified subproblem is solved exactly as in step 6. Note that k and m' remain unchanged and that this subproblem can never be a *complete integer solution*. If $T_{LB}(k) < T_u$ the modified subproblem is stored back into the tree, otherwise it is removed from the tree. Go to step 4.

8. The current best solution is the optimal solution. The optimal cycle time T_{opt} is equal to T_u and the optimal variables $x^{(opt)}_{ij}$ are equal to $x^{(u)}_{ij}$. Stop.

8 Experimental results and conclusions

The algorithm was implemented in a program and tested on many different cases. It is typical for the problem (1)–(3) that there are often many equivalent optimal solutions. One of the 10 optimal solutions of the example given in the Table 1 is presented in the Table 2. It took less than 0.01 seconds of computer time (on a 2.4 GHz Pentium 4) to find all 10 optimal solutions.

The computing time depends not only on the number of variables x_{ij} but also on the problem parameters t_{ij} and especially s_i. The lower values of s_i obviously make the search space smaller and reduce the computation time. Experiments have shown that for the problem parameters similar to those in the Table 1 all optimal solutions are typically found within a minute of computing time if the number of variables x_{ij} is 60 or fewer. For example, the 3-machine case from the Table 1 in which the number of different component types per board is increased to 20, takes less than a second to find all optimal solutions. This time increases to almost 2 hours if an additional machine is added (giving a problem with $4 \times 20 = 80$ variables x_{ij}). It should be noted, however, that for this example a suboptimal solution that is within 0.1% of the optimum was found after less than 0.1 second. This behaviour is typical for the branch-and-bound based algorithms where a great amount of time is often needed to prove the optimality of a solution that was found early in the process.

The algorithm was also tested on problems with a much larger number of variables x_{ij}. Cases with up to 10 machines and up to 100 different component types per board (giving up to 1000 variables x_{ij}) were tried. Because of the exponential nature of the algorithm the optimal solution is not found and/or proved optimal in a reasonable computing time for problems this large. But the algorithm is useful even in such cases — the branching strategy ensures that many good near-optimal solution are obtained. In addition, the algorithm gives a global lower bound on the optimal solution which allows the user to determine how close to the best possible solution a near-optimal solution is. The global lower bound on the optimal solution is the lowest of the $T_{LB}(k)$ in the branch-and-bound tree and is obtained in step 4 of the algorithm. It can be used to decide if a near-optimal solution is sufficiently close to the optimum and also if it is worth trying the longer computing time.

Acknowledgment

The authors would like to thank Prof. B. Vilfan for providing the formal proof of *NP*-completeness for the problem (1)–(3).

References

[1] P. Ji, Y.S. Wong, H.T. Loh, L.C. Lee, "SMT production scheduling: A generalized transportation approach," *International Journal of Production Research*, vol.32 (10), pp.2323–2333, 1994.

[2] J.C Ammons, M. Carlyle, L. Cranmer, G. Depuy, K. Ellis, L.F. Mcginnis, C.A. Tovey, H. Xu, "Component allocation to balance workload in printed circuit

card assembly system," *IIE Transactions*, vol.29 (4), pp.265–275, 1997.

[3] A. Schtub, O.Z. Maimon, "Role of similarity measures in PCB grouping procedure," *International Journal of Production Research*, vol.30 (5), pp.973–983, 1992.

[4] J. Sohn, S. Park, "Efficient operation of a surface mounting machine with a multihead turret," *International Journal of Production Research*, vol.34 (4), pp.1131–1143, 1996.

[5] Z. Ji, M.C. Leu, H. Wong, "Application of linear assignment model for planning of robotic printed circuit board assembly," *ASME Manufacturing Processes and Material Challenges in Microelectronics Packaging*, vol.ADM-v131/EEP-v1, pp.35–41, 1991.

[6] M. Sadiq, T.L. Landers, G. Taylor, "A heuristic algorithm for minimizing total production time for a sequence of jobs on a surface mount placement machine," *International Journal of Production Research*, vol.31 (6), pp.1327–1341, 1993.

[7] Y.D. Kim, H.G. Lim, M.W. Park, "Search heuristics for a flowshop scheduling problem in a printed circuit board assembly process," *European Journal of Operational Research*, vol.91 (1), pp.124–143, 1996.

[8] P. Ji, M.T. Sze, W.B. Lee, "A genetic algorithm of determining cycle time for printed circuit board assembly lines," *European Journal of Operational Research*, vol.128 (3), pp.175–184, 2001.

[9] P. Brucker, "*Scheduling algorithms*," Second Ed., Springer, pp.274–307, 1998.

[10] B. Vilfan, "NP-completeness of a certain scheduling problem," (in Slovenian) *Internal report*, University of Ljubljana, Faculty of Computer and Information Science, June 2002.

[11] D.M. Kodek, "A theoretical limit for finite wordlength FIR digital filters," *Proc. of the 1998 CISS Conference*, vol. II, pp.836–841, Princeton, March 20-22, 1998.

[12] D.M. Kodek, "An approximation error lower bound for integer polynomial minimax approximation," *Electrotechnical Review*, vol.69 (5), pp.266–272, 2002.

[13] C.H. Papadimitrou and K. Steiglitz, "*Combinatorial optimization*," Prentice-Hall, pp.433–453, 1982.

[14] E. Horowitz, S. Sahni, "*Fundamentals of computer algorithms*," Computer Science Press, pp.370–421, 1978.

[15] V.F. Demyanov, V.N Malozemov, "*Introduction to minimax*," Dover, pp.113–115, 1990.