An Algorithm for Minimising Due Times Violations in Flexible Package Production Scheduling

Francisco S. Ibáñez
Daniel Díaz Araya
Raymundo Q. Forradellas

LISI – Laboratorio Integrado de Sistemas Inteligentes
IdeI - Instituto de Informática – Dpto. de Informática
Universidad Nacional de San Juan
Cereseto y Meglioli – 5400 San Juan – Argentina
Tel: +54 264 426 47 21 - Fax: +54 264 426 51 01
{fibanez, ddiaz, kike}@iinfo.unsj.edu.ar

ABSTRACT

This paper includes part of the strategies used to solve a scheduling problem developed for a company that produces flexible packaging, presented in a quite general form though. In this problem it is necessary to schedule several jobs that involve four process and for each one of them there is a group of machines available (of similar characteristics). Each activity is performed on just one machine.
Besides, for our application, the scheduling must try to verify certain conditions. For each process (and consequently for all the activities that perform this process) there is a list of attributes.
The problem is not only to assign each activity to a starting time and to a specific machine, but also to try to verify conditions that depend on the values of the attributes of the activities. Moreover, there are criteria to choose a particular machine.
An approach to solve this problem was presented first in [1]. As mentioned there, some due dates could not be fulfilled on time. An approach to decrease the quantity of due dates violations was presented in [2]. This approach generates acceptable results for most of the cases in the real application. However, there were some cases in which the Algorithm did not work properly. The present work includes an Algorithm that improves the results generated in [2] for some special cases that arose in the real application.

Keywords: Scheduling Problems, Constrains Satisfaction, Optimization, Production, Flexible Packaging.

1. INTRODUCTION

This paper includes part of the strategies used to solve a scheduling problem developed for a company that produces flexible packaging. The application have been implemented in C++, employing routines of Ilog [3]. In this problem it is necessary to schedule several jobs. These jobs involve four process: Printing, Laminating, Cutting and Packing and for each one of them there is a group of machines available (of similar characteristics). Each job is described by a list of four activities of given processing times, that perform the mentioned processes in that order. Each activity is performed on just one machine. For example, if *a* represents a printing activity and {M1, ...,Mk} represent the set of machines capable of executing the printing process, *a* will be performed by a member of the set {M1, ...,Mk}. For our application the scheduling must also try to verify certain conditions.

For each process (and for all the activities that perform this process) there is a list of attributes. For the printing process, the attributes are: *ink line, duration of the (printing) process*, etc. These attributes are also associated to the machines but their values depend on the time. For each printing machine $M_1, ..., M_k$, the values of the attributes at time *t* are defined as equal to the values of the attributes of the activity that is being performed at time *t.* If no activity is being performed at *t*, these values are set to those of the last activity performed before *t*. For each attribute, there is a condition that must try to satisfy the schedules of the machines $M_1, ..., M_k$.
Given a machine *M* and an activity *a*, each condition associated to *M* is evaluated at time *t,* as a function of the value of the corresponding attribute of *M* at time *t,* and the value of the same attribute of *a.* For example, for the attribute *ink line*, (corresponding to the printing process) the condition is *to preserve the ink line*. If the activity *a* uses machine *M* and is scheduled starting at time *t,* the condition *to preserve the ink line* holds at time *t*, if the value of the attribute *ink line* for *M* at time *t* is equal to the value of the attribute *ink line* of the activity *a*. In the practical application, the verification of this condition represents the fact that the activity *a* and the previous one use the same ink line.
The problem is to assign each activity to a starting time and to a specific machine trying to verify the conditions. This problem can be considered as a "Multi Objective COmbinatorial" (MOCO) problems where the objectives are determined by the conditions. In the bibliography that we have found about MOCO problems, the multi-objective functions are evaluated after finding a solution (see [4] & [5]).
In our problem, the objectives to be fulfilled have a very peculiar characteristic: The conditions (i.e. *to preserve ink line*, etc.) that must be verified, are associated with pairs of activities scheduled consecutively in one machine; whereas [4] & [5] need all the activities to be scheduled to evaluate the objective functions. As a result, our algorithm can evaluate the objectives in each step that leads to a solution, as opposed to evaluating the multi-objective function after the whole solution was found, as it is done in [4] & [5]. A comparison of these approaches would be deceptive since we take advantage of particular features of our problem that allows us to guide our search for solutions whereas the other approaches are much more general. The problem has been initially modeled in [1], using alternative resource sets [3].
From now on alternative resource sets will be referred as AltResSets. An AltResSet is a compound resource that contains two or more equivalent resources, called alternative resources, to which activities

can be assigned. An AltResSet is defined for each process. Each AltResSet represents a set of machines such as $\{M_1, ..., M_k\}$ and contains k alternative resources that represent the machines $M_1, ..., M_k$.
The present work includes an Algorithm that improves the results generated in [2] for some special cases that arose in the real application (see 2.3).

2. SOLVING THE PROBLEM

In order to take into account the due dates, we define two attributes associated to the activities: *PriorityWeight* and *MaxEnd*.
Each job *J* has a due date, referred as *dueDate(J)*. The values of the attribute *MaxEnd* are set by executing the following pre-processing:

```
For each job J
 {
   Let a1, a2, a3 and a4 be the activities belonging to the job J
   (Printing, Laminating, Cutting and Packing, respectively)
   a4.MaxEnd = dueTime(J)
   for i = 3 down to 1{ai.MaxEnd = ai+1.MaxEnd – duration(ai+1)}
 }
```

For each activity *a*, *a.MaxEnd* represent the maximum time in which the activity *a* can finish. This value does not change during the execution of the Algorithm, whereas *a. PriorityWeight* is initially set to 0 and it increases its value every time that *a.End* > *a.MaxEnd* in the reached solution (*a.End* represents the end of the activity *a*). It has been assumed that each activity requires only one *AltResSet*.
Let *AltResSets*, *AltResources*, and *Conditions* represent: all the *AltResSets*, all the alternative resources, and all the conditions, respectively. Below we included the functions involved in the Algorithms.

StartMin: takes as argument an activity not scheduled, and returns the minimal possible start time.
AltResSet: takes as argument an activity, and returns the AltResSet required by this activity.
Verify: takes as arguments an activity *act*, an alternative resource *altRest*, and a condition *cond*, and returns 1 if *act* verify the condition *cond* at the time *StartMin(act)* with respect to the alternative resource *altRest.* Otherwise the function returns 0.
Conds: takes as argument an AltResSet, and returns the set of conditions associated with the argument.
Possible: takes as arguments, an activity *act*, and an alternative resource *altRes*, and returns 1 if it is possible to assign *altRes* to *act* at the time *StartMin(act)*. Otherwise it returns 0.
Weight: takes a condition and returns a value that represents the degree of importance of that condition.
AltRes: takes an AltResSet and returns the set of alternative resources that are part of the AltResSet.
AltResPreference: takes an activity and an alternative resource, and returns a non negative integer number, whose value is set according to the convenience of assigning the alternative resource to the activity.

Given,
an activity *a*,
an AltResSet *altResSet*,
an Alternative Resource *altRes*∈*AltRes(altResSet),*
and *conds = Conds(AltResSet)*,

the functions *AltConvenience, AltResSetConvenience* and *ActivityConvenience* are defined as follows:

AltConvenience(a, altRes, conds) =
*Possible(a, altRes) * (AltResPreference(a, altRes)*
+ $\sum_{c\in conds}$ *Verify(a, altRes,c) * Weight(c))*
+ a.PriorityWeight
AltResSetConvenience(act, altResSet) =
$Max_{recAlt\in AltRes(altResSet)}$
AltConvenience(act, altRes, Conds(altResSet))
ActivityConvenience(act) =
AltResSetConvenience(act, AltResSet(act))

2.1. Obtaining a Solution

The next Algorithm produces a solution in which the number of due dates violation depend on the value of the attribute *PriorityWeight* assigned to each activity. *Activities* represent the set of all the activities that have to be scheduled.

```
repeat
   Min = Min act∈Activities StartMin(act)
   (Get the minimum time in which it is possible to schedule an
   activity)
   MinSet = {act∈Activities : StartMin(act) = Min}

   (Get the set of activities with minimum start time Min)

   MaxConvenience=Max act∈MinSet ActivityConvenience(act)
   Pairs =
     {
      (a, altRes): a∈MinSet, r = AltResSet(a),
       altRes ∈ AltRes(r), conds = Conds(r),
       AltConvenience(a, altRes, conds) = MaxConvenience
     }
     (Get the set of pairs Activity-AlternativeResource that
     maximise the function AltConvenience).
   Select an element of the set Pairs. Let's say (a, altRes).
   Schedule the activity a at time Min assigning the alternative
   resource altRes.
until All the activities are scheduled
```

Algorithm 1. Algorithm to obtain a solution

2.2 Reducing due dates violation. First Version

The Algorithm is similar to the one presented in [2] and is based on repeatedly solving the scheduling while trying to verify as many conditions as possible (initially completely disregarding due dates) and calculating the lateness of the activities with respect to the maximum times in which the activities can finish.
This information is used in the Algorithm in the following iterations so that the delayed activities tend to be scheduled earlier. *n* represent the maximum quantity of iterations.

```
iter = 0;
for each a∈Activities { a.PriorityWeight = 0}
(initially due dates will be disregarded)

   repeat
     execute Algorithm 1
```

```
for each a∈ Activities
  {a.lateness = a.End – a.MaxEnd
        if a.lateness > 0
          then
            a.PriorityWeight = a.PriorityWeight + a.lateness * Step
  }
 iter = iter + 1
until (a.lateness <= 0 for all a∈ Activities) or (iter>n)
```

Algorithm 2. Algorithm to obtain a solution minimizing due dates violation

The greater the lateness is for an activity the greater its priority to be chosen will be in the next iteration. *Step* determines how fast the delayed activities increase will their priorities.

2.3 Reducing due dates violation. Second Version

Algorithm 2 generates acceptable results for most of the cases in the real application. However, there were some cases in which the Algorithm did not work properly. We can summarize the found drawbacks in the following issues:

1. The value of Step is not automatically set and has to be carefully chosen. An inadequate value for Step can produce bad results. There are two cases.

1.a. In each iteration, the weights and the preferences of the alternative resources compete with the latenesses of activities. If we choose too high a value for Step, we take the risk that the weights and the preferences of the alternative resources have no influence whatsoever. In this case, the Algorithm will blindly first schedule all the activities with lateness.

1. b. Conversely, if the value of Step is too low, the lateness will exert insignificant influence and the scheduling will mainly be driven by the weights and the preferences of the alternative resources. So the performance of the Algorithm is strongly dependent on the value chosen for Step.

2. Even by choosing a suitable value for Step in order to avoid the problem pointed out previously, problems still may arise in some cases. Consider two altResSets r1 and r2 such that the sum of the weight of r1 is much lower than the sum of the weights of r2. A low value for Step is suitable for r1 and too low for r2. Conversely, A high value for Step is suitable for r2 and too high for r1.

The *Algorithm 3* improves the *Algorithm 2*, (and the one presented in [2]) for special cases that arose in the real application. Cases in which there are too many weights and therefore a suitable value for Step is nor easy to find, and cases in which the situation pointed out in 2 happens.

To overcome the problems previously mentioned, we propose the *Algorithm 3* based on the following idea:

For each activity *a* that requires the AltResSet *r*, such that *a.Lateness* is greater than zero, *a.PriorityWeight* is calculated taking into account the lateness of *a*, the maximum lateness of the activities that require *r*, the weights of *r*, the preferences of using one or another alternative resource of *r*, and the number of the current iteration.

Given, an activity *a*, an AltResSet *altResSet*, and an Alternative Resource *ar*∈ *AltRes(altResSet),* we define the following functions in order to calculate the value of *a.PriorityWeight* if *a.Lateness* is greater than zero.

$$RequiredActivities(altResSet) = \{a \in Activities: AltResSet(a) = altResSet\}$$

$$MaxWeight(altResSet) = \sum_{c \in Conds(altResSet)} Weight(c)$$

$$MaxAltResPreference(altResSet) = Max_{a \in RequiredActivities(altResSet), ar \in AltRes(altResSet)} AltResPreference(a, ar)$$

$$Max(altResSet) = MaxWeight(altResSet) + MaxAltResPreference(altResSet);$$

$$MaxLateness(altResSet) = Max_{a \in RequiredActivities(altResSet)} (a.End - a.MaxEnd)$$

(*a.End – a.MaxEnd* represents the Lateness of activity *a*)

The *Algorithm 3* works as follows. As a consequence of the first line, *Algorithm 1* is initially executed disregarding due dates. The solution initially found is dedicated to verify as many conditions as possible.

The Algorithm then iterates *n* times or stops if no lateness is found. In each iteration, after executing the *Algorithm 1*, values for *a.Lateness* are determined and the values of *a.PriorityWeight* are evaluated for each activity *a* in order to be used in the next iteration.

The value of *n* has to be high enough to produce good results as will be explained later on.

```
iter = 0;
for each a∈ Activities {a.PriorityWeight = 0}
//(initially due dates will be disregarded)

repeat
   execute Algorithm 1;
   //updates a.PriorityWeight for all activity
   for each r∈ AltResSets
    {
    maxLateness = MaxLateness(r);
    max = Max(r);
    for each a∈ RequiredActivities(r)
     {
       a.Lateness = a.End – a.MaxEnd;
       if (a.Lateness > 0)
         then
           a.PriorityWeight =
                (i/n) * max * (1 + a.Lateness /maxLateness)
         else
           a.PriorityWeight =0;
     };
    };
   iter = iter + 1
until (a.lateness <= 0 for all a∈ Activities) or (iter > n)
```

Algorithm 3. Improved Algorithm to obtain a solution minimizing due dates violation

If at least one of the activities violates the due date in the last iteration (*iter* = *n*), we can deduce that

*a.PriorityWeight = max * (1 + a.Lateness /maxLateness)*

for some *a* such that *a.Lateness > 0*

It can be proven that for this iteration the Algorithm will first schedule all the activities that violate due dates, avoiding the risk pointed out in 1.b.

Proof:

Given an AltResSet *altResSet*,
if *altRes*∈ *AltRes(altResSet)*,
conds = Conds(altResSet), a∈ *RequiredActivities(altResSet)*
and *a.Lateness > 0*,
we can ensure that *maxLateness > 0*

Consequently

a.PriorityWeight > Max(altResSet),

and therefore

AltConvenience(a, altRes , conds) > Max(altResSet),

since

Possible(a, altRes) >=0,

AltResPreference(a, altRes) >= 0, and

$\sum_{c \in conds}$ *Verify(a′, rAlt,c)*Weight(c) >=0*

Let's consider now the activities scheduled on time. For all activity *a′∈RequiredActivities(altResSet),* such that *a′.Lateness = 0*, the following holds:

for all *altRes′∈AltRes(altResSet),*

AltConvenience(a′, altRest′ , conds) =

*Possible(a′, altRes′) * (AltResPreference(a′, altRes′)*

+ $\sum_{c \in conds}$ *Verify(a′, altRes′,c)*Weight(c)) + a′.PriorityWeight*;

We can infer that

AltConvenience(a′, altRest′ , conds) <= Max(altResSet)

since

Possible(a′, altRes′) <= 1,

AltResPreference(a′,altRes′) <= MaxAltResPreference(altResSet)

$\sum_{c \in conds}$ *Verify(a′, altRes′,c)*Weight(c) <= MaxWeight(altResSet)*

and

a′.PriorityWeight = 0.

As a result,

AltConvenience(a, altRes , conds) > AltConvenience(a′, altRest′ , conds)

for any pair

altRes, altRes′∈AltRes(altResSet),

and for any pair *a*, *a′,*

such that *a.Lateness > 0*

and *a′.Lateness = 0.*

Thus, at the last iteration, the Algorithm will first schedule all the activities that violate due date.

On the other hand, if we take a .high enough value of *n*, *a.PriorityWeight* will be very low for the first iterations and then the function *AltConvenience* will strongly depend on the weights and on the preference of the alternative resources, avoiding the risk cited out in 1.a.

Finally, the risk pointed out in 2, is clearly avoided since the values of *a.PriorityWeight* depend on the weights of the particular AltResSet that is required by *a*.

3. OBTAINED RESULTS

In our application, we do not use an objective function to minimize, but rather we provide different measures to evaluate the quality of the results. Between these measures are, the percentage of conditions that are verified, and the measures related to the violations of due date. It is difficult to obtain an average behavior in terms of execution time or in terms of percentage of conditions verified, due to the fact that the output is strongly dependent on the particular input data.

Unfortunately the industrial application is too complex to include input data and results. However, we can comment on the relevant problems that arose. In spite of an acceptable percentage of conditions verified, some due dates could not be reached.

The solution adopted in [1] to overcome these problems was to divide the set of activities into clusters, scheduling them independently. As it was shown in [1], this solution generates idle periods of time for the machines.

The solution found in [2] reduces the quantity of due date violations without generating idle periods of time for the machines, but these reductions, as described in this paper, depend on the data.

The approach presented in [2] showed acceptable results for most of the data used at that time, but showed poor performance for some particular cases which arose later on in the factory. The present Algorithm is mainly focussed on generating acceptable results for these cases, while keeping acceptable results for the previous cases.

The execution time of the Algorithm presented here is roughly the time required to execute one iteration (see [1]) multiplied by the number of iterations.

4. CONCLUSION

In this work, an Algorithm for solving a Scheduling for Flexible Package Production minimizing Due Times violations has been examined. This paper presents an Algorithm that improves the results generated in [2] for some particular cases.

That is, mainly, cases in which there are many conditions associated with the resources and also the weights of the resources are very different among them. Typically, the performance of the Algorithm improves as the number of iterations grows, but of course the execution time increases as well.

Although the results obtained up to now with the Algorithm presented here are better than those obtained in [2] for the mentioned cases, an exhaustive evaluation on both Algorithms has to be done on a large variety of data and this is the task that is being carried out at the present moment.

5. REFERENCES

[1] Ibañez F., Diaz D., Forradellas R.,"Scheduling for flexible package production", Proceedings IEPM'2001. Vol. 1, 385-400, Quebec, Canada, 2001. Selected work for the International Journal of Production Economics (IJPE) topic "Operation Management"

[2] Ibañez F., Diaz D., Forradellas R.," Scheduling for Flexible Package Production Minimising Due Times Violations", Eighth International Workshop on Project Management and Scheduling, EURO Working Group, (PMS 2002), www.adeit.uv.es/pms2002/, Valencia, Spain, 2002.

[3] "Ilog Schedule- Reference Manual Version 4.4", Ilog, France, 1999.

[4] Teghem J., Tuyttens D., Ulungu E.L., "An interactive heuristic method for multiobjective combinatorial optimization". Computers and Operations Research , Vol. 27. 621-634(2000).

[5] Teghem J., Ph. Fortemps, Tuyttens D., T. Loukil "Solving multi-objective production scheduling problems using metaheuristics", Proceedings IEPM'2001. Vol. 1, 385-400, 2001.