A Genetic Algorithm for Bin Packing and Line Balancing

E.Falkenauer, A.Delchambre

CRIF - Research Centre for Belgian Metalworking Industry
CP 106 - P4
50, av. F.D.Roosevelt
B-1050 Brussels, Belgium

Email: PIERRE_LECOCQ_CRIF@eurokom.ie

Abstract

The bin packing problem can be best described in 'transportation' terms: given a set of boxes of different sizes, how should one pack them all into containers of a given size, in order to use as few containers as possible?

The task of balancing of (robotized) assembly lines is of considerable industrial importance. It consists of assigning operations from a given set to workstations in a production line in such a way that (1) no assembly precedence constraint is violated, (2) no workstation in the line takes longer than a predefined cycle time to perform all the tasks assigned to it, and (3) as few workstations as possible are needed to perform all the tasks in the set.

This paper presents a genetic *grouping* algorithm for the two problems.

We first define the two problems precisely and specify a cost function suitable for the bin packing problem.

Next, we show why the classic genetic algorithm performs poorly on grouping problems and then present an encoding of solutions fitting them.

We present efficient crossover and mutation operators for the bin packing. Then we give the modification necessary to fit these operators for the line balancing.

We follow with results of performance tests on randomly generated data. Especially the line balancing tests largely cover the real-world problem size.

We conclude with a discussion of the results and areas of further research.

Keywords: Genetic algorithms, grouping, problem encoding, bin packing, line balancing.

1. Introduction - the Problems

1.1 The Bin Packing

The bin packing problem (BPP) is defined as follows ([Garey and Johnson, 79]):

Given a finite set O of numbers (the object sizes) and two constants B (the bin size) and N (the number of bins), is it possible to 'pack' all the objects into N bins, i.e. does there exist a partition of O into N or less subsets, such that the sum of elements in any of the subsets doesn't exceed B?

This NP-complete decision problem naturally gives rise to the associated NP-hard optimization problem, the first subject of this paper: what is the *best* packing, i.e. how many bins are necessary to pack all the objects into (what is the *minimum* number of subsets in the above mentioned partition)?

Being NP-hard, there is no known optimal algorithm for BPP running in polynomial time (and there will most probably never be one). However, [Garey and Johnson,79] cite simple heuristics which can be shown to be no worse (but also no better) than a rather small multiplying factor above the optimal number of bins. The idea is straightforward: starting with one empty bin, take the objects one by one and for each of them first search the bins so far used for a space large enough to accommodate it. If such a bin can be found, put the object there, if not, request a new bin. Putting the object into the first available bin found yields the First Fit (FF) heuristic. Searching for the most filled bin still having enough space for the object yields the Best Fit, a seemingly better heuristic, which can, however, be shown to perform as well (as bad) as the FF, while being slower.

1.2 The Line Balancing

The line balancing problem (LBP) can be described as follows: given a set of tasks of various lengths, subject to precedence constraints (i.e. some tasks have as prerequisite the completion of one or more other tasks, see [Sacerdoti,77]), and a time constant called *cycle time*, how should be the tasks distributed over workstations along a production (assembly) line, so that (1) no workstation takes longer than the cycle time to complete all the tasks assigned to it and (2) the precedence constraints are complied with?

In more formal terms, we define the LBP as the following decision problem:

Given a directed acyclic graph G=(T,P) (the nodes T representing the tasks and the arrows P representing the precedence constraints) with a constant L_i (task length) assigned to each node T_i, a constant C (the cycle time) and a constant N, can the nodes T be partitioned into N or less subsets S_j (the j-th station's tasks) in such a way that (1) for each of the subsets, the sum of L_is associated with the nodes in the subset doesn't exceed C, and (2) there exists an ordering of the subsets such that whenever two nodes in distinct subsets are joined by an arrow in G, the arrow goes from a higher-ordered (earlier) to a lower-ordered (later) subset?

It is easy to show that the LBP is NP-complete: it can be reduced to the NP-complete BPP, which it contains as a special case (namely, the set of precedence constraints, the arrow set P, is empty for BPP). Needless to say, the associated optimization problem, where we ask what is the *minimum* number of stations required, the second subject of this paper, is NP-hard.

As far as an available algorithm for the LBP is concerned, we are not aware of any polynomial approximation similar to those known for the BPP. However, the tight connection we just pointed out, between the two problems, will enable us to treat them in a very similar way.

1.3 The Cost Function

Let us concentrate on the BPP and try to define a suitable cost function to optimize.

The objective being to find the minimum number of bins required, the first cost function which comes to mind is simply the number of bins used to 'pack' all the objects. This is correct from the strictly mathematical point of view, but is unusable in practice. Indeed, such a cost function leads to an extremely 'unfriendly' landscape of the search space: a very small number of optimal points in the space are lost in a sea of points where this purported cost function is just one unit above the optimum. More importantly, all those slightly suboptimal points yield the *same* cost. The trouble is that such a cost function lacks any capacity of *guiding* an algorithm in the search.

Consider the extreme case where just one arrangement of the objects yields the optimum of, say, N bins. The number of possible arrangements yielding N+1 bins grows exponentially with N and is thus very large even for small problem sizes. Nevertheless, all these points in the search space yield the same cost of N+1 and thus appear to be absolutely equal in terms of merit for searching their surroundings. In other words, an algorithm would have to run into the optimal solution by mere chance. That would be impractical, to say the least.

We thus have to find a cost function which assigns similar (but not equal) values to similar solutions, while having the same optima as the function above. In other words, we have to identify the smallest natural piece of a solution which is meaningful enough to convey information about the expected quality of the solution it's part of.

Fortunately, this is easy to do in the case of BPP: the bin points itself out as the natural 'information quantum'. We simply realize that the better *each* of the bins is used, the fewer bins one needs to pack all the objects in. Or, conversely, a bad use of bins' capacity leads to the necessity of supplementary bins, in order to contain the objects not packed into the wasted space.

In order to champion the bin, rather than the overall performance of all the bins together, we also have to account for the following: if we take two bins and shuffle their contents among them, the situation where one of the bins is nearly full (leaving the other one nearly empty) is better than when the two bins are about equally filled. This is because the nearly empty bin will more easily accommodate additional objects which could otherwise be too big to fit into either of the half-filled bins.

We thus settled for the following cost function for the BPP: maximize

$$f_{\mathrm{BPP}} = \frac{\Sigma_{i=1..N}(\mathrm{fill}_i / C)^k}{N}$$

with N being the number of bins used,
fill_i the sum of sizes of the objects in the bin i,
C the bin capacity and
k a constant, k>1.

In other words, the cost function to maximize is the average, over all bins, of the k-th power of 'bin efficiency' measuring the exploitation of a bin's capacity.

The constant k expresses our concentration on the well-filled 'elite' bins in comparison to the less filled ones. Should k=1, only the total number of bins used would matter, contrary to the remark above. The larger k is, the more we prefer the 'extremists', as opposed to a collection of about equally filled bins. We have experimented with several values of k and found out that k=2 gives good results. Larger values of k seem to lead to premature convergence of the algorithm, as the local optima, due to a few well-filled bins, are too hard to escape.

1.4 Context of the Line Balancing

Figure 1 presents the global architecture of an *Integrated Control of a flexible assembly cell*, under development in our lab, which the line balancing program is a part of.

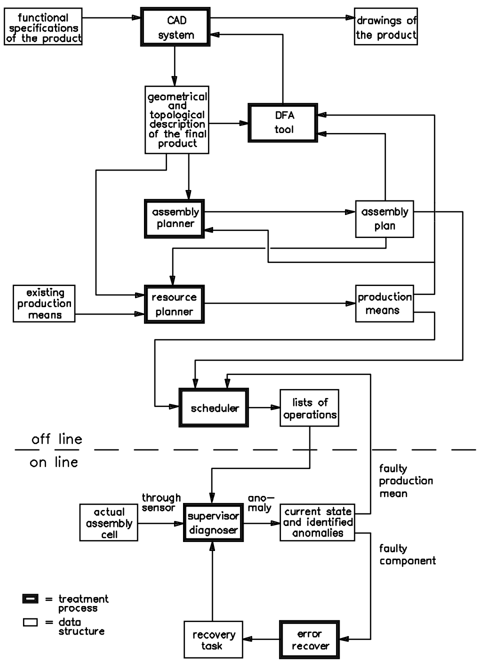

Figure 1: Overall architecture of the Integrated Control.

The figure shows the various functions that this control must fulfil and the interactions between them.

The system includes two essential parts :

- an off-line part
- an on-line part.

1.4.1 Off-line Programming

The off-line programming includes four functions :

- design for assembly,
- assembly planning,
- resource planning,
- scheduling.

1.4.1.1 Design for assembly

This module evaluates a first proposal of product design. It helps the user to improve the design of this product in order to make its assembly process more efficient. This tool uses Design for Assembly (DFA), Design for Manufacturing (DFM) and Design for Quality (DFQ) techniques.

1.4.1.2 Assembly planning

Assembly planning generates assembly plans based on three types of description: geometrical (dimensions of components and their relative positions in the final assembly), physical (the components' physical characteristics), and topological (relations between parts of the assembly). An assembly plan is a graph which nodes represent assembly operations and links represent precedence constraints. General knowledge of the assembly cell makes it possible to choose among the various plans proposed. Except for these general criteria, the plan is independent of the production means used.

This is a task-level programming system. Ideally, the operator specifies the type of object to be assembled and the system generates the assembly plan. The data elements composing these plans are the assembly actions or operations, in the form of object-level instructions :

- insert peg a in hole b,
- place part c on part d,
- screw part e onto part f.

These are the operations required to construct the product. They can be performed serially and/or in parallel. A plan thus usually lists a set of sequential series of actions called assembly sequences.

1.4.1.3 Resource planning

From the chosen assembly plan and using a database of existing production means, a design tool helps the user to design both general production means (assembly methods : manual, automated or robotized; cell layout) and specific production means (robots, fixtures, feeding devices, grippers,...).

1.4.1.4 Scheduling

On the basis of the chosen assembly plan or plans and of a detailed description of the production means available (robots, fixtures, grippers, feeding devices, ...), scheduling is needed to answer the question : "Who does what and when ?". In other words, it synchronises assembly operations and distributes them to the cell's various actors.

It's at this stage that collision-free paths are generated.

1.4.2 On-line Control

The on-line programming system controls the execution of assembly operations. It includes two main functions :

- assembly supervision and error diagnosis,
- error recovery.

1.4.2.1 Assembly supervision and error diagnosis

This module monitors assembly execution, detects errors and identifies the causes of an anomaly: one or several defective parts, a broken-down tool, ...

It makes intensive use of various sensors (proximity, force, vision,...).

1.4.2.2 Error Recovery

This module plans and controls the correction operations, so as to return the cell to a state in which normal assembly may continue. This step's essential criterion is to eliminate as few components as possible.

The case of faulty tools requires rescheduling of operations, as shown in figure 1.

This paper deals with the first step of resource planning: the line balancing problem. It minimizes the number of workstations using cycle time and operation execution time constraints.

2. Genetic Algorithms and Grouping Problems

2.1 Generalities

The Genetic Algorithms (GAs) are an optimization technique for functions defined over finite (discrete) domains. Since their introduction some 15 years ago ([Holland,75]) they have been extensively studied and applied to a wide variety of problems, including machine learning and NP-hard optimization problems ([Goldberg, 89], [Holland et al.,86], [Grefenstette,85], [Grefenstette, 87], [Schaffer,89]).

The reader not familiar with the GA paradigm can consult the recent literature on the topic (e.g. [Morrow,91]) - we won't introduce it here for space reasons. A short description can be found, for example, in [Falkenauer and Bouffouix,91], while [Goldberg,89], for instance, offers an excellent presentation of much of the GA technique.

Let us, however, recall that the main concept underlying the GA mechanics are *schemata* - portions of chromosomes which map onto subspaces of the search space. In fact, the efficiency of GAs (or, more precisely, of the crossover operator) stems from the fact that the algorithms, while manipulating the chromosomes, implicitely manipulate large numbers of schemata ([Holland,75]).

In this section we will examine the effects of the classic genetic operators on the structures relevant to the *grouping* problems. These are the optimization problems where the aim is to group members of a set into a small number of families, in order to optimize a cost function, while complying with some hard constraints (which, in particular, prevent the grouping of *all* the objects into the same unique family). A problem is a grouping when the cost function to maximize increases with the size and decreases with the number of families created.

The BPP (and hence LBP) can clearly be seen to be a grouping problem: the aim is to group the objects into families (bins) and the cost function f_{BPP} above indeed grows with the size (fill) and decreases with the number (explicitly via N, and implicitly via fill) of the families.

For the sake of clarity, we will consider the bin packing problem in the rest of this section, and assume that the hard constraint, i.e. the maximum bin capacity, is always complied with.

2.2 The Crossover

Let us see how the significant (strong) schemata relevant to the problem of grouping are transferred from parents to offspring under the standard crossover.

Let's assume the most straightforward encoding scheme, namely one gene per object. For example, the chromosome

ADBCEB

would encode the solution where the first object is in the group (bin) A, the second in the group D, third in B, fourth in C, fifth in E and sixth in the group B. Note that the third and the sixth objects are in the group B - the objects are thus *grouped* in the same group.

Since, by definition of the problem, the cost function increases with the size of groups, the grouping of the two objects into one group constitutes a gain and should be thus transmitted to the next generations.

However, the two genes are positioned too far from each other on the chromosome to be safe against disruption during the crossover. Indeed, the probability that a crossing site will fall between the two of them grows with their distance.

We must thus find a way to shorten this kind of schema. The standard way to do this is through *inversion*. By storing the loci together with the alleles, the same chromosome, written this time

123456 (the number of the object)
ADBCEB (the group the object is in),

could be arranged by (possibly successive) inversion(s) into

123654
ADBBEC.

This chromosome has the interesting genes tightly together. This time the group of Bs has good chances to survive a crossover.

So far so good. The problems begin when the groups become larger, that is, incidentally, when the optimization process starts to develop a good solution of the grouping problem. Indeed, in a chromosome like

123654
ACBBBB,

the probability of disruption of the very promising group of four Bs is as large as it was for the group of two Bs without inversion. We thus see that inversion cannot help in assuring good survival rates for schemata relevant to the grouping problem. This is because the *good* schemata for this problem are, by definition, *long* schemata.

In other words, while the classic crossover (fitted with inversion or not) might converge to a better solution in the beginning of the genetic search, once a good candidate has been found, instead of improving this solution, it *works against its own progress* towards destruction of the good schemata. The result is, of course, an algorithm stagnating on poor, never improving solutions.

2.3 The Mutation

Let's again consider the standard encoding and see the effects of the standard mutation, i.e. a random modification of a randomly chosen allele, in the case of grouping.

Consider for example the following chromosome :

ABDBAC.

A mutation of this individual could yield

ABDEAC,

which could be beneficial, for the allele E, perhaps missing from the population, appears in the genetic pool.

The troubles begin, once again, as the algorithm approaches a good solution, developing large groups of identical alleles. The standard mutation of

AAABBB

would lead, for example, to

AAEBBB.

On the one hand, the allele E appears in the population - a possibly beneficial effect. On the other hand, the new chromosome contains a 'group' of just one element. Since grouping of objects accounts for a gain, this mutated individual will most probably show a *steep loss* of fitness in comparison with the other non-mutated individuals. Consequently, this individual will be eliminated with high probability from the population on the very next step of the algorithm, yielding hardly any benefit for the genetic search.

In other words, the classic mutation is *too destructive* once the GA begins to reach a good solution of the grouping problem.

3. Operators for the two Problems

3.1 The Encoding

As we have seen, the standard genetic operators are not suitable for the grouping problems. Unlike with the deceptive problems of [Goldberg, 87], the strong schemata are not misleading. Rather, they do not survive the very genetic search supposed to improve them.

The main reason is that the structure of the simple chromosome (which the above operators work with) is much too *object* oriented, instead of being *group* (i.e. bin) oriented. In short, the above encoding is not adapted to the cost function to optimize. Indeed, f_{BPP} champions the promising bins, but there is no structural counterpart for them in the chromosome above. That is a serious drawback: the fact that the *object* i is in the bin j is meaningless - it is the fact that the *bin* j is full or empty that is important. Of course, given the distribution of the objects to the bins, one can always compute the state of the bins, but such an information is far *too indirect* for the GA (i.e. the operators) to be taken into account efficiently.

Note that these remarks are nothing more than a call for compliance with the *Thesis of Good Building Blocks*, central to the GA paradigm. Note also that since we did not refer explicitly to the BPP or LBP, the above conclusions apply to all grouping problems. Indeed, for instance [Falkenauer,91] applies similar conclusions in a successful treatment of a classification problem, which turns out to be a grouping.

To remedy the above problems, we have chosen the following encoding scheme: the standard chromosome above is augmented with a *group part*, encoding the bins on a one gene for one bin basis. For example, the first and the next to last chromosomes above would be encoded as follows:

ADBCEB:BECDA
AAABBB:AB,

with the group part written after the semicolon.

The important point is that the genetic operators will work with the group part of the chromosomes, the standard object part of the chromosomes merely serving to identify which objects actually form which group. Note in particular that this implies that the operators will have to handle chromosomes of *variable length*.

3.2 Apport of Known Heuristics

In constructing a solution of the BPP, we can make the following observation: in a good solution, *most* of the bins are well filled. This is important, because it shows that in constructing a new solution of the problem, it usually pays to look for well filled bins.

On the other hand, the FF heuristic (see section 1.1 above) is attractive as a means of placing objects into bins for three reasons.

First, it places the objects one by one, i.e. its strategy is independent of the whole of the set of objects to pack. That means the heuristic does make sense in case a *subset* of the objects is to be placed into bins.

Second, it does not need to start each time from the scratch, i.e. it can start with a set of partially filled bins (i.e. with a *partial solution*) and still pack the supplementary objects in a reasonable way.

Third, it is *complete*, i.e. it can generate the optimal solution. Indeed, there is a leeway in its function: the solution it produces depends on the order in which the objects are presented to the heuristic (and thus placed into the bins), and at least one permutation of the objects leads to the optimum. Note that this is rather exceptional for a heuristic. For example, another heuristic for BPP, the First Fit Descending (FFD) in [Garey and Johnson, 79], first sorts the objects in decreasing order of their sizes before applying the FF strategy. While performing slightly better than FF, it always produces a unique solution of a problem.

Given the three points above, we took advantage of the FF and FFD heuristics in conceiving the genetic operators for the BPP.

3.3 Generating the First Population

The genetic algorithm starts from an initial population which is usually generated at random. This initial 'seed' cannot, however, contain invalid individuals, which means that we must find a way to generate a solution which is both random and compliant with the constraint(s).

The FF heuristic serves well as the initial solution-generator. When presented with the objects in a random order, it generates a solution which is reasonable (i.e. not unacceptably bad), yet still random to a large extent. More importantly, the modification introduced in the section 3.7 below will make it generate valid individuals for the line balancing as well.

3.4 BPCX, the Bin Packing Crossover

A crossover's job consists in producing offspring out of two parents in such a way that the children inherit as much of the meaningful information from *both* parents as possible. Since it is the bin that conveys important information in BPP, we must find a way to transmit bins from the parents onto the children. This is done as follows.

Consider the following *group parts* of the chromosomes to cross (recall that there is one gene per bin):

ABCDEF	(first parent)
abcd	(second parent).

First, copies are made of the two parents (in order not to destroy the parents) and two crossing sites are chosen at random in each of them, yielding for example

A|BCD|EF and
ab|cd|.

Next, the bins between the crossing sites in the second chromosome are *injected* into the first, at the first crossing site, yielding

AcdBCDEF.

Now some of the objects appear twice in the solution and must be thus eliminated. Suppose the objects injected with the bins c and d also appear in the bins C, E and F. We eliminate those bins, leaving

AcdBD.

With the elimination of those three bins we have, however, most probably eliminated objects which were not injected with the bins c and d. Those objects are thus missing from the solution. To fix this last problem, we apply the FFD heuristic to reinsert them, yielding, say

AcdBD*x*,

where *x* are one or more bins formed of the reinserted objects.

As can easily be seen, the child just constructed indeed inherited important information from *both* parents, namely the bins A, B and D from the first and c and d from the second. Note, however, that the bins A, B and D might not be exactly the original ones found in the first parent, because the FFD might have filled them up with some of the objects reinserted in the last stage of the BPCX. Nonetheless, this is actually beneficial, since it leads to bins even better filled than in the parent.

3.5 The Mutation

The mutation operator for the BPP is very simple: given a chromosome, we select at random a few bins and eliminate them. The objects which composed those bins are thus missing from the solution and we use the FF to insert them back in a *random order*.

In order to improve the chances of the mutation to improve the current solution, we follow two rules: the emptiest bin is always among the eliminated ones, and we always eliminate and subsequently reinsert at least three bins (the number of used bins cannot be improved with less).

Note that, as with the BPCX, even some of the bins not selected for elimination might be filled up with objects from the eliminated bins.

3.6 The Inversion

The role of an inversion operator is to change the positions of some of the genes on the chromosome in order to assist the crossover in transmitting good schemata to the offspring (see section 2.2). Unlike the other two operators, it thus only affects the representation of the genetic contents of a chromosome, not the information itself.

The inversion we use for the BPP is the classic one, the only difference being that it operates only on the group (bin) part of the chromosome. Since the membership of objects in bins is not affected by a change of bins' positions on the chromosome, the object part of the chromosome remains unchanged. For example, the chromosome

ADBCEB:BECDA

could be inverted into

ADBCEB:CEBDA.

The example illustrates the utility of this operator: should B and D be promising genes (i.e. well filled bins), the probability of transmitting *both* of them during the next crossover is improved after the inversion, since they are now closer together, i.e. safer against disruption. That in turn makes the proliferation of the good schemata easier.

3.7 Modification for the Line Balancing

Given the operators above and the similarity between the bin packing and line balancing pointed out in section 1.2, we can easily construct the crossover and mutation operators for the latter problem. An inversion doesn't change the information in a chromosome, so the one above can be used for both problems.

Since the only difference between BPP and LBP lies with the supplementary precedence constraint for the latter, all we have to do to adapt the BPP operators for LBP, is to ensure that none of them will violate this constraint.

This is easily done if we note the following: (1) eliminating groups (i.e. workstations) from a valid solution doesn't spoil the constraint, i.e. the new (partial) solution is again a valid one, and (2) each time we complete a solution by placing objects into bins, we use the FF or FFD heuristic. Clearly, making sure the heuristics never violate the precedence constraints yields valid operators for the line balancing problem.

Reverting to the LBP definition above, we note that the precedence constraint is violated when there is a *cycle* in the graph obtained from the original graph G by merging the nodes belonging to the same group (workstation) into one node, preserving all the arrows except the ones that make a node point to itself.

Indeed, with a cycle in the above graph, there is no way we could say whether a node in the cycle precedes or follows another node in the cycle, i.e. the required ordering of the subsets cannot be found. Conversely, we can always compute the degrees of nodes in an acyclic graph, i.e. such an ordering can be found.

In order to prevent cycles in the 'group graph', we modify the heuristics as follows: each time we are about to insert an object into a group (a task onto a workstation, or, in BPP terms, an object into a bin), we compute the length (in terms of the number of arrows involved) of the longest precedence path leading from the object to all the groups already in place. We than restrict the heuristic to put the object into a group which is either not connected at all with the object, or is connected via exactly one arrow.

This cycling prevention for the heuristics insures the validity of the mutation operator, since all we do there is to eliminate and then reinsert objects from a solution. The BPCX crossover is a bit more tricky: the injected bins (workstations) can be incompatible (i.e. induce cycles) with the ones already in place.

Hence we proceed as follows: having injected groups and eliminated those containing double occurrences of objects from the chromosome, we check for the presence of cycles between the groups left in the chromosome. If there is none, we proceed as usual, with the constrained FFD. If so, in order to modify the inherited schemata as little as possible, we repeatedly eliminate *one object*, chosen at random from those which form the cycle, and check again for the presence of a cycle. As soon as the cycle is broken, we have a valid partial solution, and can proceed as above.

4. Experimental Results

4.1 The Bin Packing

Since the First Fit Descending can be shown to be a good heuristic for the BPP, it constituted a benchmark in tests of performance of our GA approach.

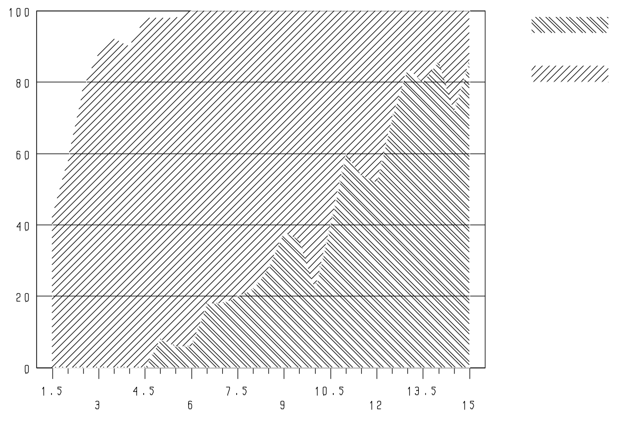

Figure 2: Relative BPP performance of the GA and FFD.

We constructed the test data as follows: we first generated objects of random sizes admitting a *perfect packing* (i.e. $f_{BPP}=1$), and then subtracted from randomly chosen objects a total of LEEWAY percentage of the size of one bin. For example, with LEEWAY set to 3% and the bin size of 255, the total size of the objects was 7.65 less than the total capacity of the bins in the perfect packing. Thus the test reflected the ability of the algorithm to get as *close* as possible to the optimum, rather than its eventual ability to find *the* optimum perfect packing. Indeed, the latter test would be a test of optimality, which would be equivalent to asking whether we can solve in a reasonable time an NP-complete decision problem - something the algorithm wasn't and couldn't be designed to.

Since the total size of the objects was only a fraction of the bin size less than the total capacity of the bins in the perfect packing, the optimum number of bins hasn't changed. Hence we observed the ability of the algorithm to reach that number of bins, compared to the FFD heuristic. However, in order to account for a practical use of the GA, we required it to reach the optimum number of bins in at most 5000 generations.

The results are summarized in figure 2. It shows the average proportion, over 50 successive runs, of test cases of 64 objects successfully optimized by the GA and FFD, function of the LEEWAY (1.5 through 15% of the bin size). The chart shows the net superiority of the GA in 'tough' conditions, i.e. when the space for the objects to pack is tight.

The running times of the GA were of the order of a minute on a 4D35 Silicon Graphics (33 MIPS).

4.2 The Line Balancing

For the line balancing, the precedence constraints were generated for a set of tasks obtained as above, by sorting them 'in time' and making precede 'later' tasks by randomly chosen 'earlier' tasks. In short, we made sure that the optimal packing also complied with the precedence constraints.

Due to the precedence constraints, we cannot use the FFD as a benchmark anymore. Indeed, when modified as indicated in section 3.7 above, its performance deteriorates badly. Nevertheless, a figure similar to the figure 2 gives an idea of the performance of the algorithm. It was obtained as the previous one, except that the algorithm was allowed to run for a maximum of 10000 generations.

The figure demonstrates the GA's ability to find

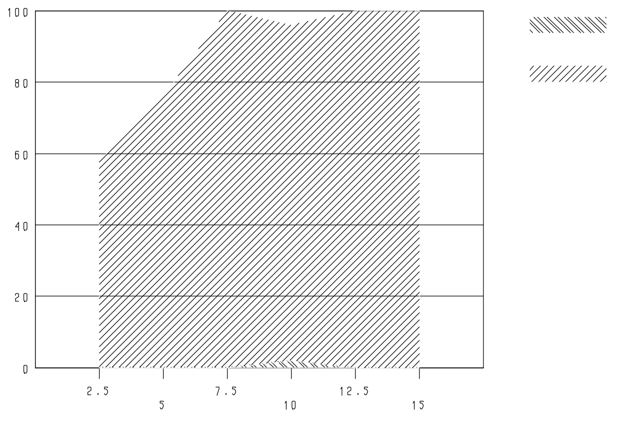

Figure 3: Relative LBP performance of the GA and FFD.

the optimal number of workstations with only a small LEEWAY (say 5% of the cycle time). The average running time was of the order of 5 minutes.

The experimental LBP results compare favorably with the industrial reality :

- The size of 64 operations comfortably covers the majority of industry's needs. Nevertheless, should that not be enough, preliminary tests suggest that problems of up to two hundred tasks could still be handled in a reasonable time.

- The cycle time being a parameter set by economic considerations, it is extremely rare to require a *perfect* line balancing (i.e. the sum of task lengths be a multiple of the cycle time), which means that a LEEWAY always exists. With that in mind, the 5% of cycle time, compared to the total length of *all* tasks, don't represent a major strain. In any case, when necessary, allowing the algorithm to run for more generations increases the probability of an optimal balancing.

The GA's performance also represents a very good alternative in comparison to most enumeration techniques. This is due mainly to its capacity to handle cases with very sparse precedence constraints, thanks to its bin packing 'ancestor'. The enumeration techniques require *much* denser precedence constraints, not common in practice, to guide them in the search, if they are to solve problems of comparable size in a reasonable time and memory space. For instance, a sophisticated dynamic programming technique used by [Peng,91] requires 60% nonempty entries in the precedence matrix in order to solve problems with 25 tasks reasonably fast.

5. Discussion

5.1 Conclusions

We have presented an efficient genetic algorithm for two NP-hard problems, the bin packing and the line balancing. The latter being an important industrial problem, we demonstrated the algorithm's ability to handle real-world data sizes.

The two problems being of the grouping type, we have justified the use of new encoding and operators by showing that the classic GA cannot perform well on problems of this kind.

5.2 Suggestions for Further Development

Let us concentrate on the line balancing problem. The algorithm is ready for industrial use in conditions given by the LBP definition above. However, in order to enhance its applicability, at least the following improvements can be suggested:

- enable the user to specify preferences of certain tasks to be or not to be performed on the same workstation. This preference could be considered as either a hard (inviolable) or a soft constraint;
- take into account the cost of each workstation, given the tasks assigned to it. For example, a station with very dissimilar tasks, i.e. a very flexible one, would be more expensive than a specialized one.

The first of these is currently under way in our lab.

6. Acknowledgement

The research described in this paper was supported in part by the EEC under the ESPRIT II Project 2637 - ARMS.

7. References

[Davis,87] Davis Lawrence (Ed) *Genetic Algorithms and Simulated Annealing*, Pitman Publishing, London.

[Falkenauer,91] Falkenauer Emanuel *A Genetic algorithm for Grouping*, in "Procs of the Fifth Int'l Symposium on Applied Stochastic Models and Data Analysis", Granada, Spain, April 23-26, 1991, World Scientific Publishing Co. Pte. Ltd.

[Falkenauer and Bouffouix,91] Falkenauer Emanuel and Bouffouix Stéphane *A Genetic Algorithm for Job Shop*, in "Procs of the 1991 IEEE Int'l Conference on Robotics and Automation", Sacramento, CA, April 1991.

[Garey and Johnson,79] Garey Michael R. and Johnson David S. *Computers and Intractability - A Guide to the Theory of NP-completeness*, W.H.Freeman Co., San Francisco.

[Goldberg,87] Goldberg David E. *Simple Genetic Algorithms and the Minimal, Deceptive Problem*, in [Davis,87].

[Goldberg,89] Goldberg David E. *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wessley Publishing Company, Inc.

[Grefenstette,85] Grefenstette John J. (Ed) *Procs of the First Int'l Conference on Genetic Algorithms and their Applications*, Carnegie-Mellon University, Pittsburgh, PA, July 24-26, 1985, Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ.

[Grefenstette,87] Grefenstette John J. (Ed) *Genetic Algorithms and their Applications: Procs of the Second Int'l Conference on Genetic Algorithms*, MIT, Cambridge, MA, July 28-31, 1987, Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ.[Holland,75] Holland John H. *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

[Holland et al.,86] Holland John H., Holyoak Keith J., Nisbett Richard E. and Thagard Paul A. *Induction: Processes of Inference, Learning and Discovery*, The MIT Press, Cambridge.

[Morrow,91] Morrow M. *Genetic Algorithms - A new class of searching algorithms*, Dr.Dobb's Journal, April 1991.

[Peng,91] Peng Yanming *The Algorithms for the Assembly Line Balancing Problem*, CRIF Internal Report, August 1991.

[Sacerdoti,77] Sacerdoti Earl D. *A structure for plans and behavior*, Stanford Research Institute, Elsevier North-Holland Inc.

[Schaffer,89] Schaffer David J. (Ed) *Procs of the Third Int'l Conference on Genetic Algorithms*, George Mason University, June 4-7, 1989, Morgan Kaufmann Publishers, Inc., San Mateo, CA.