A hybrid genetic algorithm for a loading problem in flexible manufacturing systems

C. BASNET
Department of Management Systems
The University of Waikato, Hamilton, New Zealand

Abstract
One of the operating decisions involved in the scheduling of flexible manufacturing systems (FMS) is that of loading the FMS. Given a pool of jobs, which can be processed on alternate machines and alternate tools, the scheduler has to decide on the allocation of tools and machines to the different jobs. Various versions of this problem have appeared in the literature. We consider the version where jobs are selected for processing in a FMS in a planning horizon, operations for these jobs are assigned to machines, and corresponding tools are allocated to the slots in the machines. The objective is to minimise system unbalance. A hybrid genetic algorithm is presented that addresses this problem. Computational comparison between the genetic algorithm and previous algorithms is presented.

Keywords: Flexible manufacturing systems; Genetic algorithm; Planning and scheduling; Machine loading.

Correspondence:
Chuda Basnet
Department of Management Systems
The University of Waikato
Private Bag 3105
Hamilton, New Zealand
Phone: + 64 7 838 4562
Fax: + 64 7 838 4270
Email: chuda@waikato.ac.nz
A hybrid genetic algorithm for a loading problem in flexible manufacturing systems

1. Introduction

Flexible manufacturing systems (FMS) essentially comprise machine tools with computer numerical control (CNC) and automatic transportation devices, which are controlled by a supervisory computer. This combination permits mass customisation at the low costs associated with mass manufacturing. As FMSs are adopted more widely by manufacturers, scheduling and control of FMS has become the focus of attention of many researchers. There are many problems to address, at different levels of management of FMSs. Apart from the planning and design problems to be solved at the start-up phase, there are the problems of allocating work to a FMS and to the individual machines in the FMS. Other problems have to do with sequencing the allocated work and control of the detailed movement and processing of parts.

One of the operating decisions associated with the scheduling of a Flexible manufacturing systems (FMS) is that of loading the FMS. Given a pool of jobs, which can be processed on alternate machines and alternate tools, the scheduler has to decide on the allocation of tools and machines to the different jobs. The objective is to minimise system unbalance, thus maximising throughput. In this paper, we present a hybrid genetic algorithm that addresses this FMS loading problem. Computational comparison between the genetic algorithm and previous algorithms is presented. The contribution of the paper is two-fold: we present a new algorithm for this problem; and we carry out computational studies involving large problem sizes, which has not been done before.
2. Literature review

Various models have appeared in the literature to help the manager in the planning of operations in a FMS. There are models to select the part types to be assigned to a FMS, to select and group the machines in a FMS, to choose the parts and tools to be loaded on the machines in a FMS, and to schedule the processing of parts.

Stecke (1983) suggested large non-linear integer models to solve the grouping and loading problems of FMSs. These problems could be solved with different objectives: balancing the assigned machine processing times, minimizing the number of movements, balancing the workload per machine, and unbalancing the workload per machine for a system of groups of pooled machines of unequal sizes, filling the tool magazines, and maximizing the sum of operation priorities.

Sarin and Chen (1987) approach the loading problem from the viewpoint of machining cost. In their formulation, the machines have limitations in their processing times as well as in their tool magazine capacities. The tools also have tool life limits. Since the operations can be assigned to different machine tool combinations, the problem is to select a particular combination for each operation so as to minimize the operating cost. Computational methodologies to solve the integer programming formulation are proposed. Ram et al. (1990) refine this problem, and present it as a discrete generalized network and propose a branch and bound procedure. Basnet (1996) presented a tabu search approach to this problem.

Pitts and Ventura (2009) developed a two-stage tabu search procedure for scheduling a flexible manufacturing cell with alternate routings and makespan objective. However tool magazine capacity was not considered. Similarly Moon, Lee, and Bae (2008) presented a mixed-integer linear programming formulation for a general job shop scheduling with alternate routings. Wang, Zhou, and Xi (2008) developed a filtered-beam-search-based heuristic for the same scenario. Near-optimal results with significantly less time (compared to optimal solutions) were reported. Chan, Chung, and Chan (2008) considered travel times as well as alternate routings in their formulation.

Chan, Chung, and Chan (2006) proposed a genetic algorithm to minimise the
makespan of a FMS schedule encompassing multiple factories and alternate routings. Sharafali, Co, and Goh (2004) considered production scheduling of FMS, where both the demand and processing times are stochastic.

Das and Canel (2005) consider a particular configuration of FMS where it consists of cells of similar machines, and the setup of machines is sequence dependent. Their objective is to minimise the makespan. An efficient branch and bound approach is presented. Seo and Egbelu (1999) have presented another model with a makespan objective, which combines the decisions on machine assignment, job-sequencing on machines, and guided vehicle guide-path layout in one model. Ho and Hsieh (2005) presented a part and tool assignment method for an FMS whose machines are identical. Since the machines are identical, they propose the objective of tool-shortage occurrences in the above assignment.

Özpeynirci and Azizoglu (2009) investigated the problem of selecting jobs for processing in a FMS. Their formulation considered the machines as interchangeable, but had limited tool slots. If a job was assigned to a machine, all the tools for that job were to be loaded on that machine. The objective was to select jobs within available machine capacity so as to maximise some value associated with jobs. Some upper bounds were developed for this problem. Bilgin and Azizoglu (2006) considered a similar knapsack-type problem with limitation on number of tools available.

It is clear from the above that there is no unique “FMS scheduling” problem. Authors have chosen to highlight particular facets of the FMS scenario. In this paper, we address the problem of loading in a “random” FMS. In a so-called random FMS, jobs are not preselected, and new jobs keep arriving at the FMS. The tool magazine has capacity limits and it may not be possible to allocate all the arriving jobs in one tooling set-up. This gives rise to a selection problem. Out of the pool of waiting jobs, jobs are selected to be processed in the next planning period. The selected parts are then sequenced. The process is repeated period by period. In this approach, it is assumed that at the beginning of each planning period all the tools are reassigned and replaced in the tool magazine. Shanker and Tzen (1985) propose a part selection problem for random FMS, where random jobs arrive at the FMS. Their mathematical programming approach selects jobs to be processed in the current planning
horizon considering machine capacities and tool magazine capacities. Their approach is similar to Stecke (1983) who addressed dedicated FMS. Stecke assumes the part ratio as given and the planning horizon as indefinite whereas Shanker and Tzen consider individual parts and a fixed planning horizon. They have a constraint on the tool magazine capacity which is very similar to Stecke (1983). They constrain the model to find a unique routing for each part type (in contrast to Stecke). Two objectives are considered: 1) Balancing the workload, and 2) Balancing the workload and minimizing the number of late jobs. Here balancing the workload means assigning workload (processing time) to machines so that the overload and the underload is minimised. The resulting problems presented by Shanker and Tzen are non-linear integer problems. Even after linearization, the problems are computationally too expensive, and they further propose two heuristics corresponding to the two objectives. Their experience showed that the analytical formulations would be too formidable to be of practical use. Shanker and Srinivasulu (1989) suggested a branch and backtrack algorithm and some heuristics for this problem.

Mukhopadhyay et al. (1992) considered the same problem, with the objective of minimising the system unbalance and maximising the throughput, and suggested a heuristic based on an “essentiality ratio” concept. This concept attempts to differentiate between optional operations that have multiple options for machine assignments and essential operations that have no flexibility (they can be assigned only to one particular machine). In assigning machines to optional operations, those machines which have smaller total loads from essential operations are given priority. Nagarjuna et al. (2006) have presented another heuristic for this problem. They, like Mukhopadhyay et al. (1992), exploit the distinction between essential and optional operations. In assigning operations to machines, Nagarjuna et al. (2006) give preference to the machines with more optional time requirements, more remaining processing times, and less essential time requirements.

Further works on this problem have considered the twin objectives of maximising throughput and minimising system unbalance (Vidyarthi and Tiwari, 2001; Kumar et al., 2004; Srinivas et al., 2004; Swarnkar, and Tiwari, 2004; Nagarjuna et al., 2006; Prakash et al., 2008; Kumar et al., 2006; Yogeswaran et al., 2009).
In this paper, we present a new genetic algorithm based approach for the above-discussed problem with the objective of minimising the system unbalance. We also present results of computational tests on the algorithm. Next section presents the problem discussed in this paper. After that, a description of the genetic algorithm is presented. Then computational comparisons of the heuristics are given. Finally, concluding remarks are made.

3. FMS loading problem

The problem considered in this paper can be stated briefly as follows. There is a pool of jobs waiting to be processed in a FMS. Each of these jobs consists of processing multiple identical parts (a batch), with one or more operations. Each operation has a choice of one or more alternate machines within the FMS that can carry out that operation. The machines are non-identical, but for each operation there exists a list (possibly a singleton) of alternate machines. An operation requires some processing time on the machine to which it is assigned. An operation also takes up a certain number of tool slots in that machine, to accommodate the required tool. The problem is to select jobs from the job pool to process in a planning horizon, possibly a shift, and to assign the operations of the job to particular machines. Any jobs not selected in a selection cycle are candidates for the next cycle (together with any new jobs that have arrived in the interim). Some overload or underload of the machines is allowed. The objective is to minimise this overload/underload (or unbalance). Precedence of operations is not considered, because sequencing is not of concern here – the only concern is job selection and allocation. The following notation will be used.
**Notation**

**Indices:**

\( i \) = Subscript for jobs, 1 ... ... \( N \)

\( O_i \) = Number of operations for job \( i \)

\( j \) = Subscript for operations, 1 ... ... \( O_i \)

\( k \) = Subscript for machines, 1 ... ... \( M \)

**Parameters:**

\( H \) = Length of the planning horizon

\( T_k \) = Number of tool slots available on machine \( k \)

\( B(i, j) \) = Set of machines on which operation \( j \) of job \( i \) can be processed

\( p_{ijk} \) = Processing time of job \( i \), operation \( j \), on machine \( k \) (this is the total processing time for the entire batch of identical parts in this job)

\( b_i \) = Batch size of job \( i \) (number of parts in the job)

\( t_{ijk} \) = Number of tool slots needed for processing job \( i \), operation \( j \), on machine \( k \)

**Decision variables:**

\( x_{ijk} \) = 1, if operation \( j \) of job \( i \) is assigned to machine \( k \)

= 0, otherwise

\( x_i \) = 1 if job \( i \) is selected for processing in the planning horizon

= 0, otherwise

With the above notation, the problem may be stated as:

Minimise system unbalance, \( SU = \sum_k \left| H - \sum_y x_{yjk} p_{ijk} \right| \)

where \( |x| \) indicates the absolute value of \( x \).

s.t.
\[ \sum_{j} x_{ijk} t_{ijk} \leq T_{k}, \quad k = 1, \ldots, M \]

The assigned machines must have enough slots for the requisite tools.

\[ \sum_{j \in \mathcal{A}(i,j)} x_{ijk} \leq 1, \quad i = 1, \ldots, N; \quad j = 1, \ldots, O_{i} \]

An operation can be assigned to only one machine on its alternate route.

\[ x_{ijk} = 0, \quad k \notin B(i,j); \quad i = 1, \ldots, N; \quad j = 1, \ldots, O_{i} \]

An operation cannot be assigned to a machine that is not on its alternate route.

\[ a_{x_{i}} = \sum_{k} \sum_{j} x_{ijk}, \quad i = 1, \ldots, N \]

All the operations of a job must be assigned if the job is selected for assignment.

\[ x_{i} = 0 \text{ or } 1, \quad i = 1 \ldots N \]

\[ x_{ijk} = 0 \text{ or } 1, \quad i = 1 \ldots N, \quad j = 1 \ldots O_{i}, \quad k = 1 \ldots M \]

The decision variables are 0-1 integers.

The literature on this problem differs on the objective considered. In this paper, the objective function to be minimised is:

System unbalance (SU), defined as \[ \sum_{k} \left| H - \sum_{j} x_{ijk} p_{jk} \right| \]

System unbalance (SU_{1}) is also defined in many articles as \[ \sum_{k} \left( H - \sum_{j} x_{ijk} p_{jk} \right), \] with the proviso that SU is positive. This proviso makes the practical use of this objective function questionable. Since the model does allow overloading of machines, it would seem reasonable to allow the system unbalance (SU_{1}) to be negative or positive so long as the absolute value is close to zero. Thus the objective of loading the machines close to capacity is more closely reflected in the SU objective than in the SU_{1} objective.

Throughput (TH), defined as \[ \sum_{j} b_{j} x_{i} \], i.e., the total number of assigned parts, is another objective function often used in the literature. Obviously, throughput is maximised. It can be
argued that the objective of a scheduler in practice is to maximise the throughput of work accomplished in the FMS, i.e., \( \sum_{ijk} x_{ijk} p_{ijk} \), not necessarily to increase the throughput (TH), which is the number of parts manufactured. The objective of maximisation of work is however already implicit in the objective of minimisation of system unbalance (SU).

Many articles (Tiwari and Vidyarthi, 2000; Srinivas et al., 2004; Swarnkar, and Tiwari, 2004; Prakash et al., 2008; Kumar et al., 2006; Yogeswaran et al., 2009) combine the twin objectives of maximising throughput (TH) and minimising system unbalance (SU) into a single objective function. These articles optimise this single merged objective. Other authors (Tiwari et al., 1997; Vidyarthi and Tiwari, 2001; Kumar et al., 2004; Nagarjuna et al., 2006) have designed their heuristics explicitly to minimise system unbalance (SU), thus maximising throughput implicitly; throughput is calculated after the fact. In this paper, we focus on the objective function of system unbalance (SU). This objective was also considered by Shanker and Srinivasulu (1989), Mukhopadhyay et al. (1992), and Nagarjuna et al., (2006). We also present the results for throughput (TH).

4. A hybrid genetic algorithm for the FMS loading problem

4.1 Introduction

In this paper we present a hybrid genetic algorithm to address the problem discussed above. This consists of an application of the genetic algorithm (Goldberg, 1989) whose solution space consists of selections of jobs for assignment i.e., \( \{x_i\} \). For each selection, an operational assignment heuristic determines the assignment of operations to machines, i.e., \( \{x_{ijk}\} \). We first discuss the genetic algorithm; then the operational assignment heuristic is presented.

Genetic algorithm (GA) is a metaheuristic that pursues a general search strategy emulating the evolutionary process of natural selection to find the best (fittest) solution. The procedure starts with a number of solutions (which is called a population of chromosomes).
Solutions are manipulated (*mated*) to create new solutions, which create newer solutions, etc. The best solutions that are created are retained. Essentially the mating process consists of taking two chromosomes (*parents*) from the population of chromosomes, and generating two new chromosomes (*children*) from these. The processes of *crossover* and *mutation* are followed in the mating process. In crossover, some characteristics (*alleles*) of the parents are passed onto the children. In mutation, some of the characteristics of the children are made different from those of the parents. As the chromosomes are created, their *fitness* (objective function of the solution) is evaluated. In the selection of the parents higher probability of selection is given to fitter parents. Genetic algorithm (GA) can be generally stated as follows:

Create a population of chromosomes.

Repeat for a user-specified number of times:

1. Generate a new population of chromosomes from the current population by repeatedly selecting two parents from the population to create two children, following the processes of crossover and mutation.
2. Evaluate the fitness of the children as they are created and retain a number of children in the population.

The fittest chromosome ever is returned as the best solution.

Genetic algorithms require a representation of the solution (or chromosome) that can be manipulated to produce new chromosomes. The most common chromosomal representation is to use strings of binary digits (0-1). Thus it is necessary to map the solution space into binary strings. The processes of crossover and mutation in the mating process can be illustrated using strings of binary digits.

Parent 1: 01010101
Parent 2: 11100110

Crossing over the entire string from 4th bit (crossover *site*) onwards from parent 1 to child 2, and parent 2 to child 1, while retaining the first 3 bits from parent 1 to child 1 and parent 2 to child 2 results in the following:
Child 1: 01000110  
Child 2: 11110101  

Mutating (randomly) bit 1 and 5 of child 1 and bit 3 and 7 of child 2 (from 0 to 1 and vice versa) results in the children: 

Child 1: 11001110  
Child 2: 11010111  

4.2 Proposed hybrid genetic algorithm (HGA)  

One of the complexities of the FMS loading problem is that the job selection ($x_i$ variables) has to be done concurrently with the operational assignment ($x_{ijk}$ variables). In the proposed hybrid algorithm, the problem is decomposed, carrying out the job selection separately from the operational assignment. The complexity of the problem is thus reduced considerably. Genetic algorithm provides a particularly convenient way of carrying out this separation. The FMS loading problem offers a very convenient mapping of strings of binary digits to the FMS solution space. In the proposed heuristic, the binary digits of the chromosomes represent selection or non-selection of jobs (1 = selected, 0 = not selected). The length of the string is the number of jobs in the problem. Given a chromosome, which is thus a list of selected jobs, a heuristic algorithm assigns operations of the jobs to the machines. A very simple heuristic can carry out this assignment and is described later. Thus the FMS loading problem is neatly decomposed into two parts: the GA part offers solutions in the form of binary strings which represent sets of selected jobs, these solutions are evaluated by the operational assignment heuristic which assigns the operations of the selected jobs to machines and calculates the fitness of the solution from the system unbalance.

4.3 Details of the GA implementation  

The parameters for the GA implementation in this paper were selected on the basis of a number of initial trials, with the objective of high solution quality and low computation time.
The details of the GA as implemented in this research are given below.

**Coding scheme:** GA needs a coding of the solution space in the form of strings (chromosomes). As mentioned above, chromosomes in our GA implementation are binary strings representing solutions to the problem ($i^{th}$ job is in the solution only if the $i^{th}$ bit in the chromosome is 1).

**Fitness function:** A fitness function is needed to evaluate the fitness or solution quality of each chromosome. Obviously, in this paper our goal is the minimisation of system unbalance (SU). The system unbalance of a chromosome is found by assigning the operation of the jobs represented in the chromosome. This is done by an operational assignment heuristic (see below). However, it is customary to implement GA as a maximization problem (maximising the fitness). To convert the minimisation of SU objective to a maximisation of fitness objective, the fitness is calculated as $MH - SU$. In evaluating fitness, the issue of infeasibility is usually addressed by some penalty function. However we simply assign an arbitrary low fitness of 200 to all infeasible chromosomes.

**Initial population:** A diverse initial population is preferred in the GA literature. The initial population in our algorithm is generated randomly, assigning 0 or 1 to the bit strings with equal probability.

**Population size:** A small population does not have enough diversity; a large population increases the run time of the GA. Thus a compromise needs to be made. The population size in our GA is set at half the number of jobs, subject to a minimum of 25.

**Selection:** Perhaps the most common method of selecting parents for reproduction is the so-called *roulette wheel selection*, which was implemented in our GA. In this method, the probability of selecting a particular parent is proportional to its fitness. This is achieved by the following process:

1. The total fitness of the entire population is calculated.
2. A random number [0, 1] is picked.
3. The population of chromosomes is put in a list in non-increasing order of fitness.
4. Search for a parent starts at the top of the list, and the fitness of chromosomes is accumulated as one moves down the list, until the ratio of accumulated fitness to the
total fitness is equal to the random number.

**Cross-over and mutation:** These are the fundamental operators of GA to produce new chromosomes. We use single-point cross-over, as illustrated earlier. The single cross-over point (*allele*) is selected randomly. When a chromosome is mutated, every allele in the chromosome is independently mutated with a probability of 0.1.

**Elitism:** It is usual to keep some high fitness chromosomes in the population to enhance the chances of creating offspring of higher fitness. We do this by always transferring the fittest 15% of the chromosomes from the old population to the new population. We found that this improved the performance of the GA significantly.

**Steady state reproduction:** It is commonly held that a big part of the current genetics should be retained in the next population. We implemented this by replacing only 80% of the old population by new chromosomes. In generating the new population, following strategy was followed:

1. 15% of the old population that is the fittest (*elitism*) is simply transferred to the new population.
2. 5% of the new population comes from the remaining old population, picked randomly.
3. 10% of the new population is created randomly, assigning 0 or 1 to the bit strings with equal probability. This is to add a bit of diversity to the gene pool.
4. 10% of the new population is created by simply mutating a parent (without crossover).
5. 20% of the new population is created by crossing over of the parents (without mutation).
6. A further 40% of the new population involves both crossover and mutation of the parents.

**Stopping criterion:** In our GA, the number of iterations of population change is three times the number of machines in a problem.
4.4 Operational assignment heuristic

As mentioned before, our HGA heuristic breaks the loading problem into two stages:

- selection of jobs, done through GA as described above, and
- assignment of operations of the selected jobs to machines, done by a heuristic.

The assignment of the operations proceeds as follows:

- All the operations of the jobs selected by the GA are drawn into a list, where firstly essential operations have precedence over optional operations, and secondly, larger process times have precedence over smaller process times. The operations are then sequentially assigned to machines in the order of this list, choosing the assignment that gives the minimum system unbalance wherever there is a choice. If a feasible (i.e., tool slot constraints are satisfied) assignment cannot be found for any operation, the heuristic stops at that point and returns the chromosome (solution) as infeasible.

The system unbalance obtained by this heuristic provides the fitness of the solution for the GA heuristic (fitness $= MH - SU$). Any infeasible solution is deemed to have a fitness of 200.

5. Computational experience

The HGA heuristic as described in the above section and a few existing heuristics were programmed and tested with a number of test problems. The existing heuristics, which were proposed specifically for the objective considered in this article, SU, are described briefly below. For full details of these approaches, readers are referred to the original articles.

5.1 Complete enumeration (CE)

For smaller problem sizes the optimum solution may be found by a complete enumeration of all possible solutions. Let $a_{ij}$ denote the number of alternate machines on which operation $j$ of job $i$ can be performed (this is the cardinality of $B(i, j)$). If a job $i$ is selected for processing, the number of total possible alternate routes for this particular job is $\prod_j a_{ij}$.
However, an additional alternative is not to select the job at all, thus the total number of solutions, considering all the jobs is:

A complete enumeration of solutions will require the evaluation of all these solutions. This

\[ \prod_i \left(1 + \prod_j a_{ij}\right) \]

can well prove computationally expensive.

### 5.2 A branch and backtrack algorithm (BB)

A branch and backtrack algorithm (BB) was programmed, based on Shanker and Srinivasulu (1989). The BB algorithm breaks the problem into two parts: job selection, and operational assignment. Their objective is to minimise system unbalance and maximise the assigned workload. A complete enumeration of job selection is carried out in a depth-first search: each job can be either selected or not. The jobs are considered in the lexical order, and operational assignments are made before considering the next job. Thus the number of solutions is exponential \(2^N\) and these solutions may well be too numerous to evaluate in a reasonable computer time. If selected, the operations for the selected job are assigned by solving a sub-problem that maximises assigned workload. When the search reaches an infeasible node or a leaf, it backtracks.

### 5.3 Essential machine heuristic (EMH)

The “essential machine heuristic” (EMH) is based on Mukhopadhyay et al. (1992), who have suggested exploiting the fact that not all operations of jobs have options \(a_{ij} = 1\), for some \(i\) and \(j\). They have called these operations essential operations \(j \mid a_{ij} = 1\), other operations having multiple options have been called optional \(j \mid a_{ij} > 1\). In this heuristic, jobs are placed on a list in shortest processing time (SPT) order. Attempts are made to sequentially assign the operation of these jobs to machines. In assigning the operations of a job, the essential operations are assigned first. The machine alternatives for the optional
operations are considered next. In selecting an assignment from among the alternatives, the
updated remaining processing time on an alternative machine is compared against the updated
sum of all the essential processing time requirements on that machine. The assignment is
made on the machine that has the maximum of this ratio (essentiality ratio, $ERM$). As each
job is selected and its operations are allocated, the system unbalance is calculated. If the
allocation of a job causes an increase in system unbalance, the heuristic stops. EMH then
attempts to improve the solution obtained above by trying out a few exchanges in selections
and assignments. The EMH heuristic returns the job selection with the minimum system
unbalance in all the above trials as the best solution.

5.4 Multistage heuristic (MSH)

The “multistage heuristic” (MSH) is based on heuristic 1 of Nagarjuna et al. (2006).
MSH also uses the essential operations idea, as above, but instead of using a ratio, MSH uses
the difference between the sum of optional time and remaining time and the essential time
required on a machine to choose between machines. Jobs are assigned one at a time, optional
operations being assigned on the basis of the above difference. The system unbalance is
calculated after each job assignment. At first, assignments of only one job are examined. In
the next stage, for each of these allocations another single job is added one at a time, thus all
$\binom{N}{2}$ combination of 2 jobs are assigned. This then leads to all $\binom{N}{3}$ combination of 3 jobs in
the next stage, etc. For each combination of jobs, only the job-machine allocation with the
smallest system unbalance is retained. The heuristic returns the allocation with the least
system unbalance found in all the stages of the search. The MSH provides a partial
enumeration of the solution space, by examining multiple ways by which each combination of
jobs is allocated, thus this heuristic is non-polynomial, requiring high computational time for
large problem sizes. Since this is a breadth-first search, requiring retention of previous stages
in memory, this heuristic also has high memory requirements.
5.5 Tests with existing data-set

The extensive literature on this problem is based on 10 problems first published in Mukhopadhyay et al. (1992). We carried out computational tests by applying HGA and the other heuristics discussed above to these 10 problems. The algorithms were programmed in C++, and run in an IBM PC (Intel Core Duo CPU @ 3.00 GHz and 2.99 GHz, with 1.93 GB of RAM, Windows XP Professional). Table 1 presents the results.

<< Table 1 about here>>

It can be seen from Table 1 that, in regard to system unbalance (SU), HGA finds solutions close to the optimum as found by complete enumeration, and compares favourably with the other heuristics. In our implementation MSH performs better than BB, which is slightly better than EMH. However, in regard to throughput, the algorithms are placed in this order: EMH, CE, MSH, EGA, and BB.

Some of the results for BB, EMH, and MSH are different from results published earlier (Mukhopadhyay et al., 1992; Nagarjuna et al., 2004). First of all, the BB results in Table 1 are from stage 2 of the heuristic; the results in the literature appear to be from stage 1. Second, the EMH heuristic implemented here is based only on the essentiality ratio (ERM) of processing time (Mukhopadhyay et al., 1992), choosing a machine with maximum ERM ratio, but always selecting a machine with zero essential times where available. Thirdly, the heuristics can take a different path in different implementations depending on how ties are broken in that implementation. Fourthly, the results in Table 1 were obtained by programming published pseudo-codes of the heuristics, which have some ambiguities.
5.6 Tests with larger data-set

It is an anomaly that although the published literature on this problem justifies the need for heuristics on the basis of the intractability of large problems, the proposed heuristics have not been tried on any large problems. In fact, as seen in Table 1, the optimum solution to the 10 problems in the existing data set can be found quickly by complete enumeration, yet most of the published literature has not compared their results to the optimum, and none of the literature has tested their heuristics on larger problems. To rectify this, we created a problem generator which generated random problems of various sizes (problem size = number of jobs, \( N \)). The particulars of this generator are:

1. A planning horizon of 8 hours is used.
2. Each job has a number of identical parts (a batch) to process. The batch size of jobs is uniformly distributed between 5 and 20.
3. The number of operations for a job is uniformly distributed between 5 and 10, the number of alternates for an operation is also uniformly distributed between 1 and 3. Alternate machines are picked randomly.
4. The processing time for each operation is uniformly distributed between 5 and 15 minutes.
5. The number of available tool slots in a machine is 60. The number of tool slots needed for an operation is uniformly distributed between 6 and 20. Any overlap of tools between operations is ignored.
6. The number of machines is one-and-half times the number of jobs. On average, for \( N \) jobs, the total workload is:
   \[
   \text{Number of jobs} \times \text{average batch size} \times \text{average number of operations} \times \text{average processing time} \\
   = N \times 12.5 \times 7.5 \times 10 \text{ minutes} = 937.5 \ N \text{ minutes}
   \]

   With 1.5 * \( N \) machines available, the total available processing time is
   \[
   (1.5 \times N) \times 480 \text{ minutes} = 720 \ N \text{ minutes}
   \]

   Thus, the total workload in the job pool exceeds the available processing time by a
ratio of 937.5 : 720 (approximately 1.3:1).

Using this generator, 20 problems each were generated for problem sizes (number of jobs) ranging from 2 to 60. These problems were solved by each of the algorithms discussed above. Table 2 provides the result.

<< Table 2 about here>>

As can be expected, the CPU times for the CE, BB, and MSH search rise rapidly with respect to the number of operations, although the rate of rise for the BB and MSH is slower than that of CE. These methodologies could not find solutions for larger problems because of time /space limits. Complete enumeration, as implemented here, comes with a little bit of pruning of the search space in that the search backtracks as soon as it is determined that a branch is infeasible. Still, this procedure is prohibitive for problems with number of jobs higher than 4. The BB algorithm prunes the search further by myopically assigning the operation of jobs to machines as soon as a job is selected. Thus the solution depends on the order in which the jobs are considered. We considered the jobs in the order in which they were generated (order of arrival). Even though the search space is considerably curtailed, this approach is also prohibitively time-expensive for problems with more jobs than 20. The search in MSH algorithm proceeds from single jobs to all the combinations of 2 jobs, which lead to all the combinations of 3 jobs, etc. This is a breadth-first search where nodes are kept alive until the entire tree is mapped. This has a high computer time and memory requirement.

The EMH heuristic does not carry out a search to any degree: the idea is to order the jobs into a list and to select and assign the jobs from the list. For this reason its speed of execution is very good. Obviously the ordering of jobs plays a major role here, and the SPT ordering as used here does not seem particularly successful in terms of system unbalance.

The HGA heuristic performs better for system unbalance than the other heuristics, except complete enumeration, but there is a small time penalty. The HGA heuristic outperforms the BB and MSH heuristic even though the two searches are quite exhaustive and
very time-consuming (for larger problems).

Table 3 shows results for throughput. CE has the highest performance in this regard. EMH and MSH perform about the same. HGA and BB perform the worst.

<< Table 3 about here>>

6. Conclusion

In this paper we addressed a well-researched problem in FMS loading and presented a heuristic for the problem that is based on genetic algorithm. The GA heuristic is particularly well-suited for this purpose since the binary string representation of GA lends itself to a convenient interpretation of job selection/non-selection, thus neatly separating the job selection problem from the operation assignment problem. Once the selection of the jobs is made, a simple and effective heuristic has been constructed that seeks to minimise system unbalance for that selection. We have taken advantage of this characteristic of the FMS loading problem that it can be conveniently split into these two subproblems: one amenable to GA and the other amenable to a simple heuristic.

The contribution of this article is the development of the hybrid heuristic and testing it with larger sized problems, which has not been done before. The comparison of this HGA heuristic with a number of alternative approaches from the literature showed that the HGA heuristic appears to provide better solutions for system unbalance, but not for throughput. The EMH heuristic has performed particularly well in regard to computational time and throughput.

Future research directions in this area include a more comprehensive comparison of this heuristic with existing heuristics as well as other “intelligent” search heuristics such as the tabu search heuristic, and the simulated annealing heuristic. The optimal solutions of the smaller problems in this paper show that the heuristic solutions are not very close to optimal: this suggests the need for better heuristics. Empirical research also needs to be carried out to find out the actual decisions faced by a FMS planner/scheduler and to develop appropriate decision support systems to aid them in their tasks.
References


SHANKER, K. and SRINIVASULU, A., 1989, Some methodologies for loading problems in


Table 1. Results of computational tests with existing data-set

<table>
<thead>
<tr>
<th>Problem No.</th>
<th>System Unbalance</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HGA</td>
<td>CE</td>
</tr>
<tr>
<td>1</td>
<td>87</td>
<td>81</td>
</tr>
<tr>
<td>2</td>
<td>290</td>
<td>202</td>
</tr>
<tr>
<td>3</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>819</td>
<td>819</td>
</tr>
<tr>
<td>5</td>
<td>133</td>
<td>133</td>
</tr>
<tr>
<td>6</td>
<td>178</td>
<td>178</td>
</tr>
<tr>
<td>7</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>8</td>
<td>117</td>
<td>111</td>
</tr>
<tr>
<td>9</td>
<td>462</td>
<td>309</td>
</tr>
<tr>
<td>10</td>
<td>184</td>
<td>184</td>
</tr>
<tr>
<td>Average</td>
<td>248.9</td>
<td>223.6</td>
</tr>
</tbody>
</table>

The problems (numbered 1-10) appeared first in Mukhopadhyay et. al. (1992). Subsequent authors have published their results on this data-set. CPU Time taken for these heuristics were all less than 1 millisecond.
Table 2. System unbalance and CPU time for larger problems

<table>
<thead>
<tr>
<th>Number of Jobs</th>
<th>CE ASU</th>
<th>P</th>
<th>Time</th>
<th>BB ASU</th>
<th>P</th>
<th>Time</th>
<th>EMH ASU</th>
<th>P</th>
<th>Time</th>
<th>MSH ASU</th>
<th>P</th>
<th>Time</th>
<th>HGA ASU</th>
<th>P</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>294.4</td>
<td>1.00</td>
<td>&lt;1</td>
<td>423.0</td>
<td>1.44</td>
<td>&lt;1</td>
<td>421.8</td>
<td>1.43</td>
<td>&lt;1</td>
<td>543.4</td>
<td>1.85</td>
<td>3</td>
<td>510.4</td>
<td>1.73</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>249.7</td>
<td>1.00</td>
<td>585</td>
<td>648.1</td>
<td>2.60</td>
<td>&lt;1</td>
<td>806.6</td>
<td>3.23</td>
<td>&lt;1</td>
<td>765.5</td>
<td>3.07</td>
<td>4</td>
<td>721.4</td>
<td>2.89</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>2057.0</td>
<td>1.28</td>
<td>27</td>
<td>2556.7</td>
<td>1.59</td>
<td>&lt;1</td>
<td>1798.3</td>
<td>1.12</td>
<td>6</td>
<td>1607.0</td>
<td>1.00</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>*</td>
<td>3854.9</td>
<td>1.36</td>
<td>19483</td>
<td>4799.2</td>
<td>1.70</td>
<td>&lt;1</td>
<td>3050.3</td>
<td>1.08</td>
<td>3137</td>
<td>2828.9</td>
<td>1.00</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>9726.3</td>
<td>1.45</td>
<td>&lt;1</td>
<td>*</td>
<td>*</td>
<td>6712.5</td>
<td>1.00</td>
<td>628</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>15536.4</td>
<td>1.16</td>
<td>&lt;1</td>
<td>*</td>
<td>*</td>
<td>13410.8</td>
<td>1.00</td>
<td>1667</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>19865.1</td>
<td>1.12</td>
<td>2</td>
<td>*</td>
<td>*</td>
<td>17720.1</td>
<td>1.00</td>
<td>4483</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1.00</td>
<td>1.67</td>
<td>1.67</td>
<td>1.67</td>
<td>1.78</td>
<td>1.37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASU = Average system unbalance for 20 problems

P = Performance = Average system unbalance of this algorithm / Minimum of average system unbalances for all algorithms;

Time = Average CPU Time (milliseconds)

* The search could not be finished within a time limit of one hour.

♣ The search space could not fit in computer memory.
Table 3. Throughput for larger problems

<table>
<thead>
<tr>
<th>Number of Jobs</th>
<th>CE</th>
<th></th>
<th>BB</th>
<th></th>
<th>EMH</th>
<th></th>
<th>MSH</th>
<th></th>
<th>HGA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ATH</td>
<td>P</td>
<td>ATH</td>
<td>P</td>
<td>ATH</td>
<td>P</td>
<td>ATH</td>
<td>P</td>
<td>ATH</td>
<td>P</td>
</tr>
<tr>
<td>2</td>
<td>17.0</td>
<td>1.00</td>
<td>15.3</td>
<td>0.90</td>
<td>16.6</td>
<td>0.98</td>
<td>15.4</td>
<td>0.91</td>
<td>14.8</td>
<td>0.87</td>
</tr>
<tr>
<td>4</td>
<td>35.9</td>
<td>1.00</td>
<td>32.4</td>
<td>0.90</td>
<td>33.0</td>
<td>0.92</td>
<td>31.2</td>
<td>0.87</td>
<td>29.4</td>
<td>0.82</td>
</tr>
<tr>
<td>10</td>
<td>75.8</td>
<td>0.93</td>
<td>74.1</td>
<td>0.91</td>
<td>81.6</td>
<td>1.00</td>
<td>78.5</td>
<td>0.96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>150.4</td>
<td>0.87</td>
<td>149.5</td>
<td>0.87</td>
<td>172.3</td>
<td>1.00</td>
<td>159.8</td>
<td>0.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td>297.5</td>
<td>0.98</td>
<td></td>
<td></td>
<td>303.6</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td>445.0</td>
<td>1.00</td>
<td></td>
<td></td>
<td>408.1</td>
<td>0.92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td>613.9</td>
<td>1.00</td>
<td></td>
<td></td>
<td>542.8</td>
<td>0.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1.00</td>
<td>0.90</td>
<td>0.95</td>
<td>0.94</td>
<td>0.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ATH = Average throughput for 20 problems

P = Performance = Average throughput of this algorithm / Maximum of average throughput for all algorithms;