A Genetic Programming Approach to Solve Scheduling Problems with Parallel Simulation

Andreas Beham[1], Stephan Winkler[1,2], Stefan Wagner[2], Michael Affenzeller[2]

[1] Research Center Hagenberg, [2] Department of Software Engineering
Upper Austrian University of Applied Sciences, Campus Hagenberg
Softwarepark 11, A-4232 Hagenberg, Austria
{andreas.beham,stephan.winkler,stefan.wagner,michael.affenzeller}@heuristiclab.com

***Abstract*—Scheduling and dispatching are two ways of solving production planning problems. In this work, based on preceding works, it is explained how these two approaches can be combined by the means of an automated rule generation procedure and simulation. Genetic programming is applied as the creator and optimizer of the rules. A simulator is used for the fitness evaluation and distributed over a number of machines. Some example results suggest that the approach could be successfully applied in the real world as the results are more than human competitive.**

I. Introduction

Schedule Optimization Problems are of great importance in the field of production planning and a good optimization strategy can lead to significantly lower production times, lower set-up costs, increased delivery reliability or higher plant utilization depending on the definition of the goals. Often enough scheduling problems are also very complex and cannot always be evaluated deterministically. Among the many approaches to the problem one can separate two different directions that we call scheduling and dispatching respectively. They are both very similar, but differ in their respective view on the situation in a production planning environment.

The scheduling approach maintains and operates on a global view of the problem. It knows about pending jobs for the next x days, their deadlines, job-machine-tool combinations and possibly even about material availability. Given this information and the predefined goals it will attempt to create an optimal solution, i.e. an assignment of jobs, workers and tools to machines or generally work stations. The advantage of this approach is the direct optimization possibility: Moving or exchanging jobs lengthens or shortens the make-span and plant utilization is relatively easy to see when the schedule is displayed graphically using a Gantt chart. However, the disadvantage is the overall complexity and the constraints that can create a largely different schedule out of a small change. Another disadvantage is the rigidness of the solution: A schedule as such is a fixed production plan and any change or unforseen disturbance during the execution of that plan results in the creation and optimization of a new plan, a time intensive task.

Dispatching on the other hand is a local approach, its knowledge is limited to the direct needs of any of the machines in a given situation. Whenever a machine becomes idle it will query the job pool to receive a list of pending jobs. The job pool is similar in knowledge to the scheduling approach, but does not need to look ahead and it just needs to assemble a list of jobs that are possible to be produced right now on a given machine. The machine will then rank these jobs according to a number of criteria and the job with best rank will be chosen as the next to be processed. Clearly, the disadvantage of this approach is the lack of planning, but the advantage is a much greater degree of flexibility as it will react to changes more quickly. An urgent new job for example could be issued with a high priority and could find its way faster through the process while still maintaining some sort of local optimality which is achieved by the quality of the ranking function.

This paper is a summary and extension of the work done in [1][2] and emphasizes the point where the two approaches could be combined into one. The further layout of the paper is as follows: Section II will give an overview of the problem while section III will explain the details of the algorithm; in Section IV the optimization environment and the set-up of the test is presented. A few results are given in Section V and finally conclusions and an outlook is given in Section VI.

II. Production Planning Problem

The production planning problem treated here can be generalized to the flexible job shop problem (FJSP) and is described in [2]; a short rephrase is given here.

The problem is basically an assignment problem of r jobs $J = J_1, ..., J_r$ on a set of q machines $M = M_1, ..., M_q$, more specifically it's an assignment of s operations $O = O_1, ..., O_s$ on the m machines, since each job J_k consists of a sequence of operations $B_k = (O_{g(1)}, ..., O_{g(m)})$ where $g(l)$ denotes the index from the first up to the last operation necessary to complete job J_k. Each operation $o_l \in B_k$ can only occur once in the same sequence. Each machine M_i can process as many operations as its capacity c_i allows. However, since each machines is in one of t different configurations $K = K_1, ..., K_t$ at a time, only those operations can be executed in parallel on a given machine that require the same configuration. An operation that is to be processed on a machine with a different configuration requires a change of that configuration and results in a waiting time that depends in length on the source and target configuration as well as on the machine itself. Additionally not all machines are able to carry out all operations and not all machines are available all the time. Finally, there are a set of p orders $A = A_1, ..., A_p$ where each order $a_i \in A$ is bounded by an earliest start date and

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

a due date and requires the processing of different jobs in various quantities.

The objective function of the problem was set to decrease pass-through time, which means that for all jobs j in the manufacturing process the time between entry and exit should become minimal. In mathematical form the objective function looks as following:

$$f(A) = \sum_{i=0}^{p} \sum_{j=0}^{|a_i|} F_j - E_j$$

where F_j and E_j is the exit and enter time of a processed job respectively.

III. Genetic Programming Approach

Genetic programming (GP) [3] is a special type of encoding and generally circumscribes a GA [4] that operates on a tree representation, although theoretically any optimization algorithm could be used instead of the GA. The binary one point crossover is replaced by a subtree crossover and bit flip mutation is now a variation in the leaves or nodes of the tree. The advantage of this representation is that mathematical formulas can be represented in an expression tree. Thus GP is able to automatically build and optimize formulas and even computer programs [3]. In the tree representation a node denotes a function and a leaf represents a variable or a constant. The terminology that is often used refers to nodes as *non-terminal symbols* and to leaves as *terminal symbols*. Typical areas where GP has been successfully applied are classification, regression and time series problems [5][6].

In this work, GP is applied on the simulation model introduced in [2]. As was described in Section II, the objective function optimized in this paper evaluates a schedule, so to say a permutation of jobs with start and end time assigned on a number of machines. But the point of optimization is not the schedule itself.

If there is no planning during the production process, the question on what to do next needs to be determined every time a job is introduced into the system or a machine becomes idle. Scheduling could cope well with idle machines, but if a lot of jobs will be introduced the whole schedule would need to be reevaluated quite often - a task which is computationally quite expensive. This is the advantage of dispatching where the jobs get registered to the buffers of all machines that they need to be processed on. The machine, when it becomes idle, applies a ranking of all the jobs in its buffer and chooses the best next job according to the ranking and deletes it from all the other buffers. This ranking function is the point of optimization and lies within the domain of GP. The goal is to find an optimal ranking function for all the machines in the production environment. However GP still needs a measurement of the quality of its individuals which are ranking functions. Local objectives that are closer to the machines like a reduction of setup times or plant utilization can be measured directly, but the results from other objectives such as delivery reliability might be first available after a few days and even plant utilization is more interesting as an average over a few days. Additionally, such small time windows may not seem adequate enough to measure the performance of a certain ranking function.

This leads to the conclusion that the application of the ranking function needs to be simulated into the future. The simulation over a longer period can provide a more reliable fitness measure which is fed back into the GP optimizer. The simulator thus is similar to a scheduler and at this point both approaches merge: GP optimizes the ranking function of the scheduler which applies it for every machine at the decision points. The simulator finally calculates the performance characteristic which is fed back into the GP approach to determine the quality of the ranking function and thus of the individual.

The ranking function that GP will optimize is structured as an expression tree where the nodes or non-terminal symbols represent mathematical functions. The leaves can represent constant numbers as well as special strings. These strings denote a simple priority rule, which is a quantitative characteristic of a given job. The simulator will replace the string in the formula with the value of the respective characteristic for each job and evaluate its rank. Genetic programming will thus create a complex rule out of a few simple rules. The 17 non-terminal symbols consist of 11 functions: $+$, $-$, $\times$, $\div$, $\hat{}$, $\hat{e}$, Sqrt, Sin, Cos, Log, Sign and 5 relations: $<$, $>$, $<=$, $>=$, $==$. The last non-terminal symbol is a conditional IF-THEN-ELSE node. The terminal symbols consist of uniformly distributed constants $c \in [0, 20]$ as well as the 8 simple priority rules identified in [2]: LotID, FIFO, JobTime, JobFlex, MID, AGE, CR and EDD, which are explained as follows: LotID is a number identifying a job, FIFO is the job's number in the machine waiting queue, JobTime represents the amount of time a job has spent waiting in that queue, JobFlex denotes the number of machines a job is currently registered to, MID is a number representing the ID of the machine, AGE is the amount of time a job has spent in the whole manufacturing process, CR describes whether a job is on time, before or behind schedule and EDD represents the due data of a job. These symbols then create a tree which can be easily written down in infix, prefix or postfix notation depending on the way the tree is traversed. There are also size restrictions that apply on the tree to constrain the size of the search space.

The algorithms that are applied on the simulation model are multipopulation variants of the GA. One of these advanced algorithms is termed SASEGASA [9]. This algorithm uses an additional step in the traditional GA cycle which takes place after the mutation and evaluation have been completed and before the replacement starts. It is called *offspring selection* and it selects only those children for replacement which are better than their parents. So the algorithm would use selection, crossover, mutation and evaluation and then decide if the child is worth keeping or not. It compares its quality to that of its parents by using a *comparison factor*. This is adjustable between comparing against the worst parent, the best parent or a linearly

interpolated value between them. Naturally, after the children have passed offspring selection just a few may have survived, likely too few to replace the whole population. So the cycle is repeated until enough children were found to be worth keeping. The ratio to adjust how many children need to be better than their parents is defined in a parameter called *success ratio*. The advantage of this algorithm is that it can produce good results even with low mutation rates and low parental selection pressure. To ensure a good diversity of the individuals, the population in SASEGASA is separated into sub populations which grow together over time.

Two other algorithms are tried that simply expand the traditional GA to the multipopulation domain by using an island model in which the population is divided into sub populations called islands. They are thus termed Island GAs. Each sub population of an Island GA is optimized by a traditional GA, but they will exchange some individuals every couple of generations between them. The parameters to control the migration are called *migration rate* and *migration size*. Migration rate specifies the interval after which migration occurs while migration size denotes the amount of individuals transferred between the sub populations.

IV. HeuristicLab Optimization Framework

HeuristicLab[1] (HL) is a successful optimization framework written in C# and Microsoft .NET. The development of the first version (HL 1.0) was started in 2002 and was first released in November 2004 [7]. It is based on the concept of reusability by separating the algorithmic from the problem dependent parts and employing a plug-in mechanism to allow independent development of a number of metaheuristic concepts and optimization problems. A minor revision followed nearly a year after the initial release in August 2005, while at the same time work started on the next major version which also provided the base for this work. In HL 2.0, as [8] notes the plug-ins are split into even smaller parts that could be interchanged more or less freely. Instead of predetermining implementations of various metaheuristic techniques and their many variations the operations are of a more general nature. This way the user can create arbitrary optimization algorithms and test them on any kind of optimization problem. Another advantage of the new major version is the ability to execute the algorithm in steps and so offers the possibility to pause an optimization run, save it and continue it at a later date. Together with a powerful GUI, it is even possible to modify the algorithm while it is executing. See Figure 1 for a screenshot of the main workbench where the operations are assembled into an algorithm.

The simulation used in this work was well integrated into HL 1.1, but to take advantage of the new architecture it was ported over to HL 2.0. Since the purpose of the simulator was to evaluate a priority rule given as a formula, it was only necessary to port the evaluation operation and the data exchange protocol to the new version. This can then be combined with any of the other operations to form an optimization algorithm.

[1] http://www.heuristiclab.com

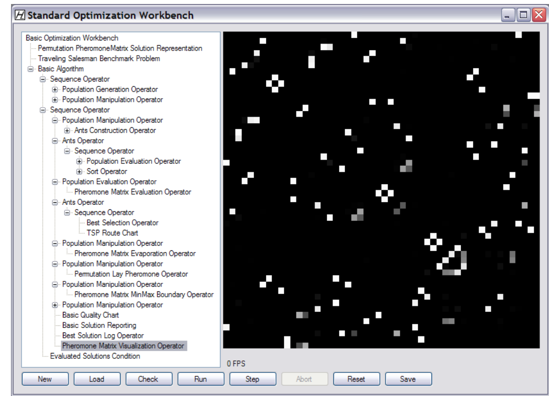

Fig. 1. Example workbench of HeuristicLab 2.0. Here: A Max-Min Ant System and the visualization of its pheromone matrix after having solved a 52 city TSP problem

The general setup of the optimization was kept the same as in [2] and is shown in Figure 2. The algorithm runs as a sequence of operations in the HeuristicLab environment and at specific points in the sequence when the generated rules need to be evaluated the newly created evaluation operator would write them to a database and assign the simulator clients that have registered themselves in the database to evaluate the formulas. The simulators run on separate clients, query the database from time to time and eventually fetch the rules, evaluate them and write the results back into the database. The evaluation operator also queries the database from time to time notes that the simulation clients have finished, fetches the results, assigns the fitness values to the respective ranking functions and continues with the optimization algorithm.

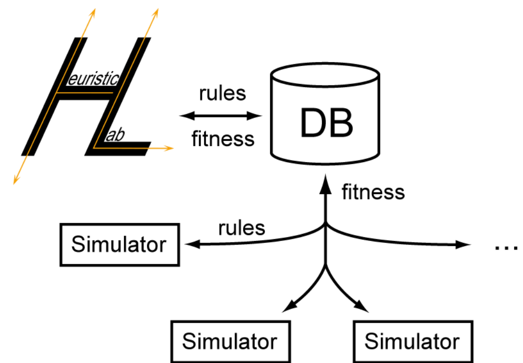

Fig. 2. Distributed Simulation setup

V. Results

The results were computed on a cluster of 15 workstations of which about 9 were continuously available. The workstations are equipped with Intel Pentium 4 processors running at 2.8 Ghz and with 1 GB of RAM. One of these

workstations hosted the database, one of them hosted the HeuristicLab application with the respective workbench together with a simulation client while a single simulation client was running on the 7 other machines. The simulation clients needed on average about 4 seconds per function evaluation as well as some initial setup time. Unfortunately the clients would not always return meaningful results, sometimes certain formulas raised an exception in the simulator and thus their fitness could not be computed. In this case it was set to the highest possible double value instead. It will be shown that the SASEGASA and offspring selection can deal with such a situation more effectively and that these formulas are kept out of the population.

In the preceding work [2] the best solution was obtained by a GA. The population size was set to 100 and it was run for 1000 generations using 20% mutation and Roulette wheel selection. The author stated that a higher mutation rate was beneficial to the quality of the results. The best average throughput time that this GA came up with was 16.3139 hours. The resulting formula was:

$$JobTime + (0.531/0.255^{JobFlex})/Sin(0.255)$$

as [2] further notes they asked a human expert to provide a solution drawing from his experience and use this solution to evaluate the strength of the algorithm. The expert suggested a sorting by JobFlex followed by JobTime which resulted in an average throughput time of 16.5597 hours. The GA showed comparable performance to that of the human expert and can be therefore considered human competitive!

In this work we tried to apply more advanced versions of the genetic algorithm. The SASEGASA was run with 10 villages and a population size of 100. It was set to use Roulette-Random Gender Specific selection. This creates a pair of parents where one parent consists of solutions selected using a proportional selection mechanism and the other is randomly selected, both from the same population. The mutation rate was set to the same 20% as in the preceding case. It also uses a Single Point CrossOver for the combination of the parents. The parameters for the Offspring Selection were set to a success ratio of 1 comparing against the better of the two parents and a maximum selection pressure of 100. After 11 generations it was able to improve the solution to an average throughput time of 16.0921 hours and could not obtain a better solution for the following 20 generations after which the run was aborted. At that time it had already evaluated 1,208,570 solutions and took 14 days to compute. See Figure 3 for a chart of the quality curve for the first 11 generations.

Additionally an Island GA with 10 islands, a population size of 100, a mutation rate of 20%, roulette wheel selection and single point crossover was applied on the problem. The migration rate was set to just 3 generations and exchanging 15% of the best individuals along an unidirectional ring. In the target population these solutions replace the double solutions or when there are not enough doubles the remaining individuals replace the worst solutions. To test whether two solutions were double their fitness values

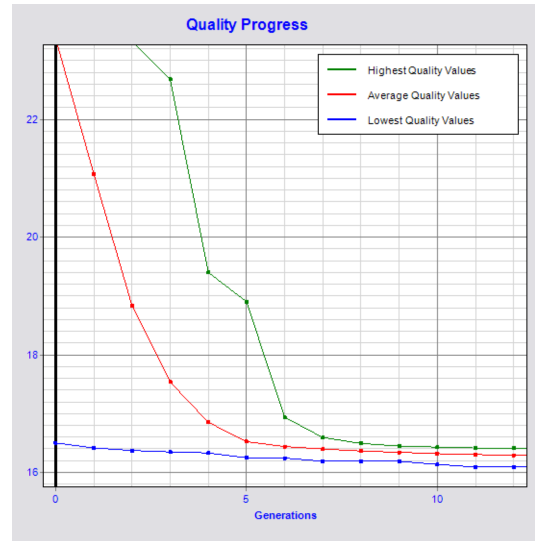

Fig. 3. Best/Average/Worst quality curve for the SASEGASA

are compared. The Island GA found its best solution at generation 1199 with a fitness value of 16.1061 and after evaluating about 1,200,000 solutions.

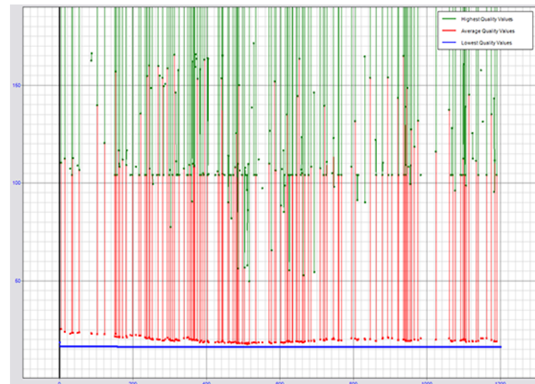

Fig. 4. Best/Average/Worst quality curve for the Island GA with high migration

The Island GA seems to benefit from a high migration rate as it could perform better than the GA and by eliminating the doubles in the replacement phase to reduce the danger of premature convergence which is present with such a setting. A comparison with a an Island GA and a lower migration rate: 15% every 15 generations resulted in a solution of slightly worse quality: 16.1557 after evaluating about 1,200,000 solutions. Both Island GAs took more than 2 weeks to compute on the cluster.

Comparing the quality charts of the SASEGASA (Figure 3) with that of the Island GA (Figure 4) shows that the SASEGASA could cope better with the states of exception in the simulator. During the run of the SASEGASA no “faulty” individual entered the population, while they would constantly appear during the run of the Island GA.

The best found formula for the given problem was found by the SASEGASA and is shown here:

$$1.02 * JobFlex + \ln(\sqrt{(e^{\cos(1.11*JobFlex)})^{1.08*CR+9.83}}$$
$$*1.01 * JobTime) * (\ln(\cos(\sin((\sin(\sqrt{0.74 * AGE})*$$
$$0.91 * AGE) + (1.11 * JobFlex)^{14.68*\ln(1.03*MID)})))+$$
$$\ln(\sqrt{(e^{\cos(0.94*JobFlex)})^{1.11*JobFlex+1.08*CR}}*$$
$$1.01 * JobTime))$$

No pruning has been applied yet. It is possible that these formulas can be shrunken in size a little, but a high complexity is likely to remain.

VI. Conclusions

We have shown that advanced genetic algorithms that make use of multiple populations could outperform a human expert by an even larger extent than in the preceding work with just a single population. The application of offspring selection in SASEGASA allowed to create a good solution without drifting into the infeasible region where the simulator would raise an exception. It could adapt better to erroneous situations in the evaluation function.

It is foreseeable by the formula found that humans will be barely able to handle this kind of complexity and that genetic algorithms are well suited to do the optimization task. However, simulation requires a substantially larger amount of CPU resources and so the success of this approach will also be decided by the amount of calculation that can be done and by deploying it on a fast parallel architecture.

Regarding future work, it would certainly be interesting to compare this approach to a scheduling approach in general and measure its competitiveness with established methods.

References

[1] W. Stöcher, B. Kabelka, and R. Preissl, "Automatically Generating Priority Rules for the Flexible Job Shop Problem with Genetic Programming," in *Proceedings of Computer Aided Systems Theory: EuroCAST 2007*, 2007.

[2] R. Preissl, "A Parallel Approach For Solving The Flexible Job Shop Problem With Priority Rules Developed By Genetic Programming," Master's thesis, Johannes Kepler University, Linz, July 2006.

[3] J. R. Koza, *Genetic Programming*. The MIT Press, 1992.

[4] J. H. Holland, *Adaption in Natural and Artifical Systems*. University of Michigan Press, 1975.

[5] S. Winkler, M. Affenzeller, and S. Wagner, "New Methods for the Identification of Nonlinear Model Structures Based Upon Genetic Programming Techniques," *Journal of Systems Science*, vol. 31, no. 1, pp. 5–13, 2005.

[6] S. Winkler, M. Affenzeller, and S. Wagner, "Advanced Genetic Programming Based Machine Learning," *Journal of Mathematical Modelling and Algorithms*, vol. 6, no. 3, pp. 455–480, 2007.

[7] S. Wagner and M. Affenzeller, "HeuristicLab: A Generic and Extensible Optimization Environment," in *Adaptive and Natural Computing Algorithms*, ser. Springer Computer Science, B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson, and N. C. Steele, Eds. Springer, 2005, pp. 538–541.

[8] S. Wagner, S. Winkler, R. Braune, G. Kronberger, A. Beham, and M. Affenzeller, "Benefits of Plugin-Based Heuristic Optimization Software Systems," 2007.

[9] M. Affenzeller and S. Wagner, "SASEGASA: A New Generic Parallel Evolutionary Algorithm for Achieving Highest Quality Results," *Journal of Heuristics - Special Issue on New Advances on Parallel Meta-Heuristics for Complex Problems*, vol. 10, pp. 239–263, 2004.