Dynamic Scheduling with Genetic Programming

Abstract. This paper investigates the use of genetic programming in automatized synthesis of scheduling heuristics. The applied scheduling technique is priority scheduling, where the next state of the system is determined based on priority values of certain system elements. The priority calculation is evolved with genetic programming, which allows creating scheduling algorithms with less effort. The evolved solutions are compared with existing scheduling heuristics for single machine dynamic problem and job shop scheduling with dynamic bottleneck estimation.

Introduction

Scheduling is concerned with the allocation of scarce resources to activities with the objective of optimizing one or more performance measures. Depending on the situation, resources and activities can take on many different forms. There are also many different performance measures to optimize: one objective may be the minimization of the makespan, while another may be the minimization of the number of late jobs, etc. Due to inherent problem complexity and variability, a large number of scheduling systems employ heuristic scheduling methods. Among many available heuristic algorithms, the question arises of which heuristic to use in a particular environment, given different performance criteria and user requirements. The problem of selecting the appropriate scheduling policy is an active area of research [8][14], and a considerable effort is needed to choose or develop the algorithm best suited to the problem at hand. A solution to this problem may be provided using machine learning, genetic programming in particular, to create problem specific scheduling algorithms.

The combinatorial nature of most scheduling problems encourages the use of search based techniques, such as genetic algorithms, simulated annealing, tabu search etc. These methods usually offer good quality solutions, but at the cost of a large amount of computational time needed to produce such a solution. Furthermore, search based techniques are not applicable in dynamic or uncertain conditions where there is need for frequent schedule modification or reaction to changing system requirements. Scheduling with heuristic algorithms that define only the next state of the system is therefore highly effective in most instances.

In this paper we propose a methodology to devise scheduling algorithms for different environments and optimization criteria. Scheduling process relies on priority scheduling, where the system is transformed into the next state based on the current priorities of activities and resources. Scheduling algorithm is composed of two elements: a meta-algorithm which uses priority values to perform scheduling and a priority function which defines values for different elements of the system. The priority function is evolved for each scheduling environment using genetic programming, while meta-algorithm is defined separately. This approach allows the creation of various scheduling methods tailored to specific conditions and optimization criteria.

Genetic programming has rarely been employed in scheduling, mainly because it is unpractical to use it to search the space of potential solutions (i.e. schedules) to the problem. This paradigm is, however, very suitable for the search of the space of algorithms that provide solution to the problem. Previous work in this area of research includes evolving scheduling policies for a single machine maximum unweighted tardiness problem [5][6][1], single machine scheduling subject to breakdowns [15], classic job shop tardiness scheduling [2][9] and airplane scheduling in air traffic control [4][7]. In all of the above articles the authors show the potential of the approach and in most cases observe performance comparable to the human-made algorithms. In this paper we address the problem of scheduling with dynamic job arrivals, for which there is a possibility of inserted idleness in resource usage. We also tackle the problem of bottleneck recognizing in multiple machine environments and define an appropriate algorithm structure for job shop scheduling.

Priority Scheduling with Genetic Programming

A natural representation for the solution of a scheduling problem is a sequence of activities to be performed on each of the machines. While this representation is most suitable for use in combinatorial optimization, it presents only the solution to a specific scheduling instance, which means that a new solution must be found for different initial conditions. With genetic programming, we have the ability

to represent a solution to the whole class of scheduling problems by devising an algorithm that can be used to generate a schedule. The schedule itself can be produced in a very short time, which allows use of this method in on-line scheduling [12] and dynamic conditions.

The scheduling method applied in this work is priority scheduling, in which certain elements of the scheduling system are assigned with priority values. The choice of the next activity being run on a certain machine is based on their respective priority values. This kind of scheduling algorithm is also called, variously, 'dispatching rule', 'scheduling rule' or just 'heuristic'. The term scheduling rule, in a narrow sense, often represents only the *priority function* which assigns values to elements of the system (jobs in most cases). For instance, a scheduling process may be described with the statement 'scheduling is performed using SPT rule'. While in most cases the method of assignment of jobs on machines based on priority values is trivial, in some environments it is not. This is particularly true in dynamic conditions where jobs arrive over time or may not be run before some other job finishes. That is why a *meta-algorithm* must be defined for each scheduling environment, dictating the way activities are scheduled based on their priorities and possible system constraints. This meta-algorithm encapsulates the priority function, but the same meta-algorithm may be used with different priority functions and vice versa.

The described structure of the scheduling algorithm allows modular development and the possibility of iterative refinement, which is particularly suitable for machine learning methods. In this work the meta-algorithm part is defined manually for a specific scheduling environment, such as dynamic one machine or job shop. The priority function is evolved with genetic programming using appropriate functional and data structures. This way, using the same meta-algorithm, different scheduling algorithms best suited for the current criteria can be devised. The task of genetic programming is to find such a priority function which would yield the best results considering given meta-algorithm and user requirements.

Single Machine Dynamic Scheduling

Problem Statement

In a single machine environment, a number n of jobs J_j are processed on a single resource. In a static problem each job is available at time zero, whereas in a dynamic problem each job has a release date r_j. The processing time of the job is p_j and its due date is d_j. The relative importance of a job is denoted with its weight w_j. In this environment the non-trivial optimization criteria include weighted tardiness and weighted number of late jobs, which are defined as follows: if C_j denotes the finishing time of job j, the job tardiness T_j is defined as

$$T_j = \max\{C_j - d_j, 0\}. \quad (1)$$

Lateness of a job U_j is taken to be 1 if a job is late, i.e. if its tardiness is greater than zero, and 0 otherwise. Weighted tardiness for a set of jobs is defined as

$$T_w = \sum_j w_j T_j \,, \quad (2)$$

and weighted number of late jobs is defined as

$$U_w = \sum_j w_j U_j \,. \quad (3)$$

Note that the weights for tardiness and lateness of a job may differ, but we do not consider that case in the analysis. In the process of evaluating scheduling heuristics we use a large number of test cases with different number of jobs, job durations and weights. In order for all the test cases to have a similar influence to the overall quality estimate of an algorithm, we define normalized criteria for each test case. Normalized weighted tardiness is defined as

$$\overline{T_w} = \frac{\sum_{j=1}^{n} w_j T_j}{n \cdot \overline{w} \cdot \overline{p}} \,, \quad (4)$$

and normalized number of late jobs as

$$\overline{U_w} = \frac{\sum_{j=1}^{n} w_j U_j}{n \cdot \overline{w}}, \tag{5}$$

where n represents the number of jobs in a test case, $\overline{w}$ the average weight and $\overline{p}$ the average duration of all jobs. The average duration is not included in weighted number of late jobs because that criteria does not include any quantity of time dependant on job's processing time. The total quality estimate of an algorithm is expressed as the sum of normalized criteria over all the test cases.

Scheduling Heuristics

In a dynamic environment the scheduler can use algorithms designed for a static environment, but two things need to be defined for those heuristics: the first is the subset of the jobs to be taken into consideration for scheduling, since some jobs may arrive in some future moment in time. The second issue is the method of evaluation of jobs which have not yet arrived, i.e. the question should the priority function for those jobs be different and in what way. It can be shown that, for any regular scheduling criteria [11], a job should not be scheduled if the waiting time for that job is longer than the processing time of the shortest of all currently available jobs. In other words, we may only consider jobs j for which

$$\left| r_j - time \right| \le \min_i \{ p_i \}, \forall i : r_i \le time \,. \tag{6}$$

This approach may be illustrated with the following meta-algorithm using an arbitrary priority function:

```
while(there are unscheduled jobs)
{  wait until machine is ready;
   p_MIN = the duration of the shortest available job;
   calculate priorities of all the jobs for which |r_j - time| < p_MIN ;
   choose job with best priority;
   schedule job with best priority;
}
```

In the above algorithm, 'best' priority is defined as the one with the greatest value, but this is purely a matter of definition. Scheduling heuristics that presume all the jobs are available must somehow take into account the time needed for the job to arrive. The simplest way of using those heuristics is by taking the processing time of a job to include the job's time till arrival (waiting time), denoted with

$$ar_j = \max\left\{ r_j - time, 0 \right\}. \tag{7}$$

Thus, if an algorithm includes the processing time of a job, that time is increased by ar_j of the job. This modification is not necessary for algorithms that are specifically designed for dynamic conditions, i.e. which already include release date information in priority calculation. Having defined the conditions of use, we can present the existing heuristics used for efficiency comparison in dynamic single machine scheduling. Each heuristic is defined with its priority function π_j which is used as described in the above meta-algorithm.

- Weighted shortest processing time heuristic (WSPT), defined with

$$\pi_j = w_j / \left(p_j + ar_j \right) \tag{8}$$

- Earliest due date heuristic (EDD), defined as

$$\pi_j = 1 / d_j \tag{9}$$

- Montagne heuristic [11] (MON), defined with

$$\pi_j = \left(w_j / \left(p_j + ar_j\right)\right) \cdot \left(1 - \frac{d_j}{\sum_{i=1}^{n} p_i}\right) \quad (10)$$

- Rachamadugu & Morton heuristic [10] (RM), defined with

$$\pi_j = \left(w_j / \left(p_j + ar_j\right)\right) \left[\exp\left(\frac{-\left(d_j - \left(p_j + ar_j\right) - time\right)^+}{k \cdot p_{AV}}\right)\right], \quad (11)$$

where p_{AV} denotes the average processing time of all the unscheduled jobs and the parameter k has the value 2 for single machine problems. The operator '+' is defined as

$$x^+ = \max\{x, 0\}. \quad (12)$$

- X-dispatch bottleneck dynamics heuristic [11] (XD), defined with

$$\pi_j = \left(w_j / p_j\right) \left[\exp\left(\frac{-\left(d_j - p_j - time\right)^+}{k \cdot p_{AV}}\right)\right] \cdot \left(1 - B\frac{\left(r_j - time\right)^+}{p_{AV}}\right), \quad (13)$$

where the value of parameter *B* is empirically set at value of 2.3, for which the heuristic achieved the best results on given set of test cases.

Test Cases

Each scheduling instance is defined with the following parameters: the number of jobs, their processing times, due dates, release dates and weights. Job durations may take integer values between 1 and 100 and their weights values between 0.01 and 1 in steps of 0.01. The values of processing times are generated using uniform, normal and quasi-bimodal probability distributions among the different test cases. Release times are chosen randomly in the interval

$$r_j \in \left[0, \frac{1}{2}\sum_{i=1}^{n} p_i\right]. \quad (14)$$

Job due dates are generated using two parameters: T as due date tightness and R as due date range, which both assume values in interval [0,1]. For each test case due dates are generated with uniform distribution in the interval

$$d_j \in \left[r_j + \left(\sum_{j=1}^{n} p_j - r_j\right) \cdot (1 - T - R/2),\ r_j + \left(\sum_{j=1}^{n} p_j - r_j\right) \cdot (1 - T + R/2)\right]. \quad (15)$$

Due date tightness parameter represents the expected percentage of late jobs and due date range defines the dispersion of due date values. The numbers of jobs in test cases are 12, 25, 50 and 100 whereas parameters T and R assume values of 0.2, 0.4, 0.6, 0.8 and 1 in various combinations. We define 100 scheduling instances that are used as fitness cases in learning process and additional 600 instances that are used for evaluation purposes only.

Scheduling with Genetic Programming

The task of genetic program is to find a priority function which is best suited for use with given criteria and meta-algorithm. The solution of genetic programming is represented with a single tree that embodies the priority function. The function and terminal set of genetic programming must allow the program to use all the relevant information and form an efficient solution. The choice of functions and terminals remains the most difficult part of the process of learning and has the greatest impact on resulting efficiency. The complete set of primitives used as tree elements is presented in Table 1.

Table 1. The function and terminal set for dynamic one machine problem

Function name	**Definition**
ADD	binary addition operator
SUB	binary subtraction operator
MUL	binary multiplication operator
DIV	protected division: $DIV(a,b) = \begin{cases} 1, \text{ if } \lvert b \rvert < 0.000001 \\ a/b, \text{ otherwise} \end{cases}$
POS	unary operator '+': $POS(a) = \max\{a, 0\}$
Terminal name	**Definition**
pt	processing time (p_j)
dd	due date (d_j)
w	weight (w_j)
N	total number of jobs
Nr	number of remaining (unscheduled) jobs
SP	sum of processing times of all jobs
SPr	sum of processing times of remaining jobs
SD	sum of due dates of all jobs
SL	positive slack, $\max\{d_j - p_j - time, 0\}$
AR	job arrival time, $\max\{r_j - time, 0\}$

Weighted Tardiness Problem

The described genetic programming process can be used for optimization of an arbitrary scheduling criteria, but the most common one for single machine environment is weighted tardiness. Fitness value of a genetic program solution is defined as sum of normalized criteria values, defined as (4), over all 100 learning test cases (smaller values are better). The genetic programming parameters are given in Table 2.

Table 2. The genetic programming parameters

Parameter / operator	**Value / description**
population size	10000
selection	steady-state
selection operator	tournament of size 3
stopping criteria	maximum number of generations (300) or maximum number of consecutive generations without best solution improvement (50)
crossover	85% probability, standard crossover
mutation	standard, swap and shrink mutation, 3% probability for each
reproduction	5% probability
initialization	ramped half-and-half, max. depth of 5

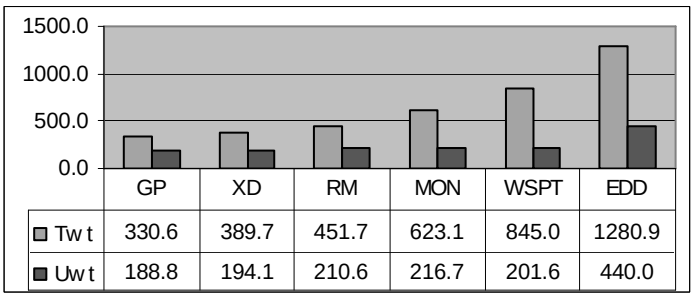
We conducted 20 GP runs with the above parameters using the defined meta-algorithm. The overall solution was chosen among best solutions of each run as the one with the best performance on the unseen set of 600 evaluation test cases. Since genetic programming is able to learn and adapt to the given meta-algorithm, we propose an alternative procedure coupled with the appropriate evolved priority function:

```
while(there are unscheduled jobs)
{  wait until machine is ready;
   p_MIN = the duration of the shortest available job;
   p_MAX = the duration of the longest job j for which r_j ≤ time + p_MIN ;
   calculate priorities of all the jobs for which |r_j − time| < p_MAX ;
   choose job J_i with best priority;
```

```
    if( |r_i − time| < p_MIN )
       schedule J_i ;
    else
    { choose job J_k with best priority for which r_k + p_k ≤ r_i and
         |r_k − time| < p_MIN ;
       schedule J_k ;
    }
}
```

The motivation behind the above algorithm is the avoidance of the situation in which a high priority job may arrive after the time limit of the shortest available job's duration and can be delayed by some other scheduled job. In other words, we extend the time horizon to p_{MAX} but still prevent waiting longer than p_{jMIN}. Of course, it is up to genetic programming process to find such a priority function that can efficiently exploit this meta-algorithm.

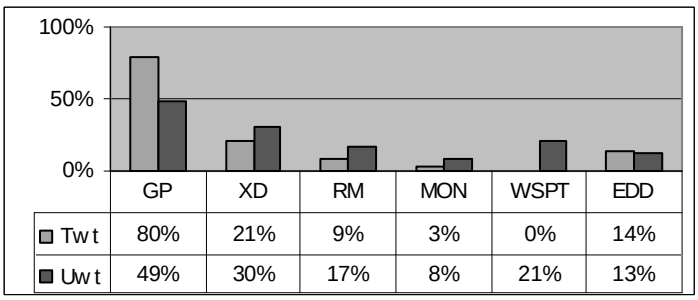
Additional 20 runs were conducted using the new meta-algorithm and the best solution of those runs is chosen in regard with its performance on evaluation set of test cases. The scheduling heuristic evolved with the new meta-algorithm showed slightly better performance than the one using the previously defined algorithm. Specifically, normalized tardiness penalty over all evaluation test cases with the modified meta-algorithm was 330.62 whereas the original meta-algorithm achieved 335.09. The difference is not great, which can be attributed to the lack of situations, among the test cases, in which the modified meta-algorithm could take advantage of the extended time horizon to make a better scheduling decision. The best solution achieved with the new meta-algorithm was compared with existing scheduling heuristics, and the results are presented in Fig. 1 ('Twt' denotes weighted tardiness and 'Uwt' weighted number of tardy jobs).



	GP	XD	RM	MON	WSPT	EDD
Twt	330.6	389.7	451.7	623.1	845.0	1280.9
Uwt	188.8	194.1	210.6	216.7	201.6	440.0

Fig. 1. Total normalized criteria values for one machine dynamic problem

Apart from total criteria values, the performance measure for each heuristic may also be described as the percentage of test cases in which a heuristic achieved the best known result (or at least the result that is not worse than any other heuristic). This value can be denoted as the dominance percentage, and the comparative results for all the heuristics are shown in Fig. 2.



	GP	XD	RM	MON	WSPT	EDD
Twt	80%	21%	9%	3%	0%	14%
Uwt	49%	30%	17%	8%	21%	13%

Fig. 2. Dominance percentages for one machine dynamic problem

It can be perceived that the evolved scheduling heuristic achieved the best overall performance for both scheduling criteria. In addition, we performed experiments with weighted number of tardy jobs as the fitness function and, as expected, the evolved algorithm's efficiency for that criteria was improved.

At the same time, the performance in regard of weighted tardiness decreases significantly, so we may conclude that the weighted tardiness criteria optimization pays off better considering overall algorithm quality.

Job Shop Scheduling

Problem Statement

Job shop scheduling includes running n jobs on m machines where each job has m operations and each operation is to be processed on a specific machine (more general model involves arbitrary number of operations for any job). Every machine and job is considered to be available for processing from the beginning. The operations of each job have to be completed in a specific sequence which differs from job to job. In addition to weighted tardiness and number of tardy jobs, another non-trivial and widely used criteria are weighted flowtime and makespan. Normalized weighted flowtime of a set of jobs is defined as

$$\overline{F_w} = \frac{\sum_{j=1}^{n} w_j F_j}{n \cdot \overline{w} \cdot \overline{p}}, \tag{16}$$

where F_j equals to the completion time of the last operation of a job, C_j. Normalized makespan is similarly defined as

$$\overline{C} = \frac{\max\{C_j\}}{n \cdot \overline{p}}. \tag{17}$$

Although the jobs are considered to be available from the time zero, the scheduling on a given machine is inherently dynamic because an operation may only be ready at some time in the future (after the completion of the job's previous operation). We must therefore define the priority calculation for the operations available in the future, as in the single machine dynamic problem. This can be resolved in the following ways:

1. no inserted idleness - we only consider those operations which are immediately available;
2. inserted idleness - waiting for an operation is allowed and waiting time is added to operation's processing time in priority calculation (as was the case in single machine dynamic environment);
3. inserted idleness with arbitrary priority - waiting is allowed but the priority function must be defined so it takes waiting time into account.

For test cases used in this work, the scheduling heuristics used for comparison purposes (defined below) exhibit better performance with the first method, so the first method is used when comparing efficiency with genetic programming evolved algorithm. The genetic programming, on the other hand, is coupled with the third method, as it has the ability to learn and make use of waiting time information on itself.

Scheduling Heuristics

Job shop priority scheduling involves determining the next operation to be processed on a given machine. The scheduling on a machine may only occur if the machine is available and if either of the following is true: there are operations ready to be processed on that machine or there are operations which will be ready for processing at a known time in future. The latter situation occurs if the previous operation of a job has already started and we know the time it will finish. This procedure can be described with the following meta-algorithm:

```
while(there are unprocessed operations)
{  wait till a machine with waiting operations becomes available;
   calculate priorities of all pending operations;
   schedule best priority operation;
   determine next operation of a job and update machine and
     operation ready time;
}
```

The choice of operations considered for scheduling is still restricted to those operations whose waiting time (7) is smaller than the duration of the shortest available operation. In definition of scheduling heuristics, we use the following notation:

- p_{ij} - duration of one operation of job j on machine i (the sequence of operations is not included as it is irrelevant to priority function);
- w_j - job weight;
- d_j - job due date;
- twk_j - total processing time of all operations of job j;
- $twkr_j$ - processing time of remaining operations of job j (remaining operations are all those not already being processed).

Having stated the notation, we can present some existing scheduling heuristics for the job shop problem. Each heuristic is described with its priority function; detailed descriptions of the listed heuristics can be found in [3] and [11].

- WSPT heuristic, defined as

$$\pi_j = \frac{w_j}{p_{ij}}. \tag{18}$$

- WSPT/TWKR (processing time to the total work remaining) heuristic, defined with

$$\pi_j = \frac{w_j \cdot twkr_j}{p_{ij}}. \tag{19}$$

- WTWKR (weighted total work remaining), defined as

$$\pi_j = \frac{w_j}{twkr_j}. \tag{20}$$

- SLACK/TWKR (dynamic slack per remaining process time), defined as

$$\pi_j = \frac{w_j \cdot twkr_j}{\left(d_j - twkr_j\right)}. \tag{21}$$

- COVERT (cost over time) heuristic, defined with

$$\pi_j = \frac{w_j}{p_{ij}}\left[1 - \frac{\left(d_j - ELT_{ij} - time\right)^+}{h \cdot ELT_{ij}}\right]^+, \tag{22}$$

where parameter h has the value of 0.5, as suggested by the authors, and ELT_{ij} is estimated lead time of a job, i.e. the time before the job j leaves the system after its current operation i finishes. Lead time is estimated as $ELT_{ij} = k \cdot twkr_j$, where k equals 3, which is a widely used value.

- Rachamadugu & Morton job shop heuristic (RM), defined with

$$\pi_j = \frac{w_j}{p_{ij}} \cdot \exp\left[\frac{\left(d_j - ELT_{ij} - time\right)^+}{h \cdot \overline{p}_i}\right], \tag{23}$$

where $\overline{p}_i$ is the average duration of all the operations on given machine and the value of h is empirically set at 10. Lead time estimation is calculated as in COVERT heuristic.

Test Cases

The operations processing times and job weights are generated randomly as for the one machine environment. Job numbers are 12, 25, 50 and 100 whereas the number of machines takes values of 5, 10, 15 or 20 for each test case. The expected total duration of all the jobs is defined as

$$\hat{p} = \frac{1}{m}\sum_{j=1}^{n}\sum_{i=1}^{m} p_{ij} \ , \tag{24}$$

and job due dates are generated randomly with parameters T and R in the following interval:

$$d_j \in \left[\hat{p}\left(1-T-R/2\right), \hat{p}\left(1-T+R/2\right)\right]. \tag{25}$$

We define 160 test cases for learning and 320 evaluation test cases, in addition to 80 instances taken from various sources [13] used for evaluation only.

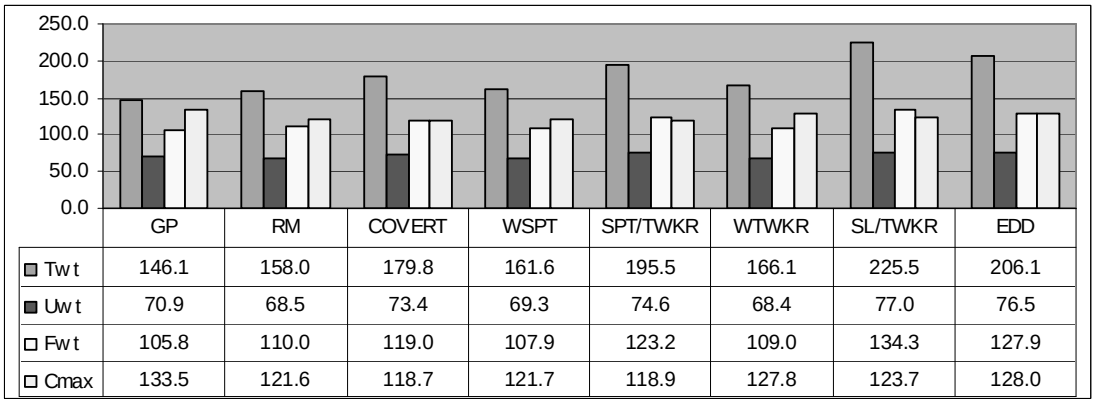
Scheduling with Genetic Programming

As in the single machine case, the solution of genetic programming is a single tree which represents the priority function to be used with defined meta-algorithm. The choice of functions is similar to the previous implementation, but the terminals are radically different, because they must include different information of the system state. The set of functions and terminals is presented in Table 3.

Table 3. The function and terminal set for job shop problem

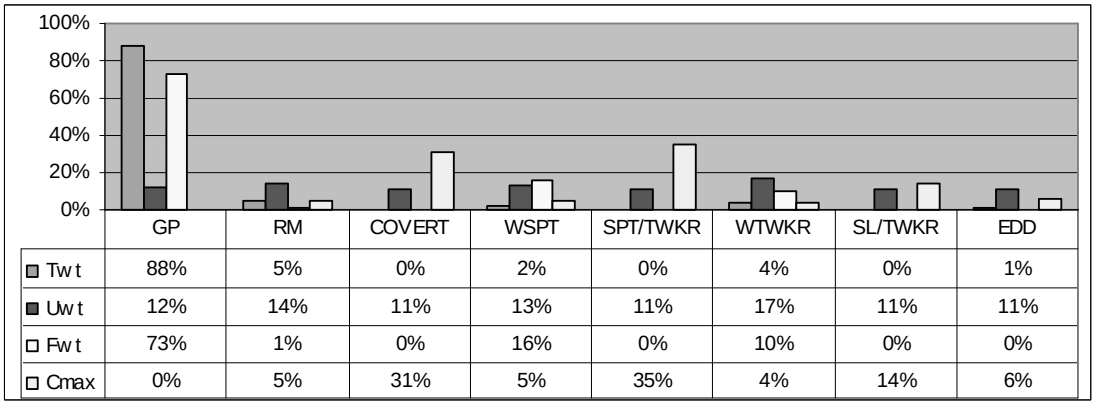
Function name	**Definition**
ADD, SUB, MUL, DIV, POS	as in Table 1
SQR	protected unary square root: $SQR(a) = \begin{cases} 1, \text{ if } a < 0 \\ \sqrt{a}, \text{ otherwise} \end{cases}$
IFGT	comparison operator: $IFGT\left(a,b,c,d\right) = \begin{cases} c, \text{ if } a > b \\ d, \text{ otherwise} \end{cases}$
Terminal name	**Definition**
pt	operation processing time (p_{ij})
dd	job due date (d_j)
w	job weight (w_j)
CLK	current time
AR	operation waiting time; $\max\left\{r_{ij} - time, 0\right\}$, where r_{ij} denotes finishing time of the previous operation (before machine i)
NOPr	number of remaining job operations
TWK	total processing time of all operations of a job (twk_j)
TWKr	processing time of remaining operations of a job ($twkr_j$)
PTav	average duration of all the operations on a given machine
HTR	head time ratio: the ratio of the total time the job has been in the system and total duration of job's completed operations

We conducted 20 experiments optimizing weighted tardiness criteria with the same GP parameters as in Table 2. The best solution was compared with the existing scheduling heuristics on the evaluation set of 400 (320 + 80) test cases. The results in the form of normalized criteria values and dominance percentages are shown in Fig. 3 and Fig. 4.



	GP	RM	COVERT	WSPT	SPT/TWKR	WTWKR	SL/TWKR	EDD
Twt	146.1	158.0	179.8	161.6	195.5	166.1	225.5	206.1
Uwt	70.9	68.5	73.4	69.3	74.6	68.4	77.0	76.5
Fwt	105.8	110.0	119.0	107.9	123.2	109.0	134.3	127.9
Cmax	133.5	121.6	118.7	121.7	118.9	127.8	123.7	128.0

Fig. 3. Total normalized criteria values for job shop problem



	GP	RM	COVERT	WSPT	SPT/TWKR	WTWKR	SL/TWKR	EDD
Twt	88%	5%	0%	2%	0%	4%	0%	1%
Uwt	12%	14%	11%	13%	11%	17%	11%	11%
Fwt	73%	1%	0%	16%	0%	10%	0%	0%
Cmax	0%	5%	31%	5%	35%	4%	14%	6%

Fig. 4. Dominance percentages for job shop problem

Scheduling with Adaptive Heuristic

It has already been shown [9] that the identification of the bottleneck resource, i.e. the resource with substantially higher load, may improve the scheduling process. As it is generally not known in advance which machine could become a bottleneck, we may try and develop a heuristic to determine such resource on line. The scheduling problem may then be viewed as a multi-agent system where one agent is responsible for scheduling bottleneck resource(s) and the other (or more agents) is responsible for non-bottleneck scheduling. We propose a genetic programming approach where there are two distinctive agents, or scheduling heuristics, and the GP is responsible for evolving the rule to decide which heuristic is to be applied on a given machine. The solution of genetic programming consists of three parts (represented as trees): the first part, or the decision tree, determines which heuristic should be used for every scheduling decision. The other two parts, scheduling trees, are applied depending on the result of the decision tree. Scheduling trees use the same primitives as in Table 3, but the decision tree should be able to recognize increased load on a machine and we have to define a new set of terminals for it. The terminals which can be used in the decision tree are presented in Table 4 (the functions are the same in all trees).

Table 4. The terminal set for decision tree

Terminal name	**Definition**
MTWK	total processing time of all operations on a machine
MTWKr	processing time of all remaining operations on a machine
MTWKav	average duration of all operation on all machines
MNOPr	number of remaining operations on a machine
MNOPw	number of waiting operations on a machine
MUTL	utilization: the ratio of duration of all processed operations on a machine and total elapsed time

As the result of the decision tree is a numeric value, we have to interpret it in some way and define the scheduling process. This procedure can be described with the following meta-algorithm:

```
for(each machine i)
   P_i = decision tree value;
while(there are unprocessed operations)
{  wait till a machine with waiting operations becomes available;
   P_i = current decision tree value;
   if( P_i > P_m, ∀m )
     calculate priorities using the second tree;
   else
     calculate priorities using the first tree;
   schedule best priority operation;
   determine next operation of a job and update machine and
     operation ready time;
}
```

Using the above adaptive structure, we conducted 20 runs with the same set of parameters and learning test cases. The best solution is compared with the other scheduling algorithms and the results are shown in Table 5 (we include only GP values for brevity since the other heuristics are unchanged).

Table 5. Results for adaptive (multiple tree) genetic programming heuristic

	Normalized criteria	**Dominance percentage**
Twt	143.8	94 %
Uwt	67.2	17 %
Fwt	104.5	86 %
Cmax	132.9	0 %

It is interesting to compare the performance of single and multiple tree GP algorithms; their comparison in regard to dominance percentage is shown in Fig. 5.

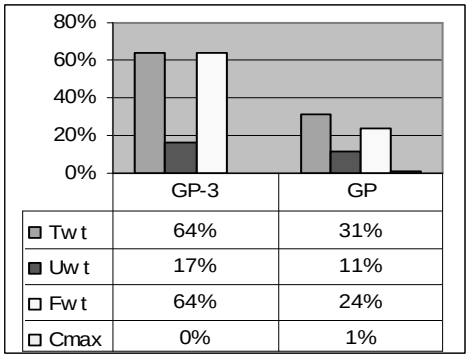


Fig. 5. Dominance percentage comparison between single tree solution ('GP') and multiple tree solution ('GP-3')

The difference between the two algorithms in terms of absolute criteria values is not great; on the other hand, the relative dominance of the multiple tree algorithm is much greater, as it is able to find non-dominated solution in a majority of problem instances.

Conclusion

This paper presents the use of genetic programming in automatized synthesis of scheduling heuristics. The genetic programming paradigm can be used efficiently to build scheduling algorithms whose performance is measurable with human-made heuristics for a specific scheduling environment. The scheduling algorithm is structured in two parts: the first part is a meta-algorithm that controls the scheduling procedure and the second one is the priority function that chooses the 'best' element to be scheduled. This structure simplifies the problem and allows machine learning techniques to be used in priority function definition.

We addressed the issue of dynamic single machine scheduling for which a suitable meta-algorithm is defined. Additionally, a multiple tree adaptive heuristic is proposed for job shop scheduling problem, where decision tree is used to distinguish between resources based on their load characteristics. The results are promising, as for given problems the evolved heuristics exhibit better performance than existing scheduling methods. The presented methodology can be particularly useful in scheduling environments where there are no adequate algorithms and could alleviate the design of an appropriate scheduling procedure.

References

1. Adams, T. P.: Creation of Simple, Deadline, and Priority Scheduling Algorithms using Genetic Programming, Genetic Algorithms and Genetic Programming at Stanford (2002)
2. Atlan, B.L., Polack J.B.: Learning distributed reactive strategies by genetic programming for the general job shop problem, Proceedings 7th annual Florida Artificial Intelligence Research Symposium, Pensacola, Florida, IEEE Press (1994)
3. Chang, Y.-L., Sueyoshi, T., Sullivan, R.: Ranking dispatching rules by data envelopment analysis in a job shop environment, IIE Transactions, 28(8):631-642 (1996)
4. Cheng, V.H.L., Crawford, L.S., Menon, P.K.: Air Traffic Control Using Genetic Search Techniques, IEEE International Conference on Control Applications, August 22-27, Hawai'i (1999)
5. Dimopoulos C., Zalzala, A.M.S.: A genetic programming heuristic for the one-machine total tardiness problem, Proceedings of the 1999 Congress on Evolutionary Computation, Volume: 3, 6-9 July (1999)
6. Dimopoulos C., Zalzala, A.M.S.: Investigating the use of genetic programming for a classic one-machine scheduling problem, Advances in Engineering Software, Volume 32, Issue 6 (2001), p 489-498
7. Hansen, J.V.: Genetic search methods in air traffic control, Computers and Operations Research, V 31, N 3, (2004), 445-459
8. Jones, A., Rabelo, L.C.: Survey of Job Shop Scheduling Techniques, NISTIR, National Institute of Standards and Technology, Gaithersburg, MD (1998)
9. Miyashita, K.: Job-Shop Scheduling with GP, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), pp. 505-512, Morgan Kaufmann, 10-12 July (2000)
10. Mohan, R., Rachamadugu, V., Morton, T.E.: Myopic Heuristics for the Weighted Tardiness Problem on Identical Parallel Machines, Working Paper, The Robotics Institute, Carnegie-Mellon University (1983)
11. Morton, T.E., Pentico, D.W.: Heuristic Scheduling Systems, John Wiley & Sons, Inc. (1993)
12. Pinedo, M.: Offline Deterministic Scheduling, Stochastic Scheduling, and Online Deterministic Scheduling: A Comparative Overview. In: Leung, J. Y-T. (ed.): Handbook of Scheduling, Chapman & Hall/CRC (2004)
13. Taillard, E.: Scheduling Instances (2003) http://ina.eivd.ch/Collaborateurs/etd/problemes.dir/ordonnancement.dir/ordonnancement.html
14. Walker, S.S., Brennan, R.W., Norrie, D.H.: Holonic Job Shop Scheduling Using a Multiagent System, IEEE Intelligent Systems, 2/2005, pp. 50-57 (2005)
15. Yin, W.-J., Liu, M., Wu, C.: Learning single-machine scheduling heuristics subject to machine breakdowns with genetic programming, Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, IEEE Press, 8-12 December (2003), 1050-1055