Evolving Priority Scheduling Heuristics with Genetic Programming

Domagoj Jakobović, Kristina Marasović

University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia
University of Split, Faculty of Science, Teslina 12, 21000 Split, Croatia
domagoj.jakobovic@fer.hr, kristina.marasovic@pmfst.hr

Abstract

This paper investigates the use of genetic programming in automatized synthesis of scheduling heuristics for an arbitrary performance criteria. The applied scheduling technique is priority scheduling, where the next state of the system is determined based on priority values of certain system elements. Genetic programming is used to create the priority function which, coupled with an appropriate meta-algorithm for a given environment, forms the priority scheduling heuristic. The evolved solutions are compared with existing scheduling heuristics and found to perform similarly or better than existing algorithms. We intend to show that this approach may be particularly useful for those combinations of scheduling environment and criteria for which there are no adequate scheduling algorithms.

Key words: genetic programming, priority scheduling
PACS:

1 Introduction

Scheduling is concerned with the allocation of scarce resources to activities with the objective of optimizing one or more performance measures, which can assume minimization of makespan, job tardiness, number of late jobs etc. The combinatorial nature of most scheduling problems allows the use of *search based* and *enumerative* techniques [10], such as genetic algorithms, branch and bound, simulated annealing 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 (i.e. resource failures or job parameter changes). Scheduling with simple but

fast heuristic algorithms that build the schedule directly (not searching the solution space) is therefore highly effective, and the only feasible solution, in many instances.

Due to inherent problem complexity and variability, a large number of scheduling systems employ such 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 [10][19], 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.

Genetic programming (GP) has rarely been employed in scheduling, mainly because it is unpractical to use it to search the space of potential solutions (i.e. schedules). It 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 single machine unweighted tardiness problem [6][7][1], single machine scheduling subject to breakdowns [20], classic job shop tardiness scheduling [2][13] and airplane scheduling in air traffic control [4][9]. In most cases the authors observe performance comparable to the human-made algorithms. The scheduling procedure is however defined only implicitly for a given single scheduling environment. Moreover, the scheduling paradigm is reduced to list scheduling where job ordering is determined only at the beginning of the process, which reduces the choice of usable heuristics.

In this paper we structure the scheduling algorithm in two components: a *meta-algorithm*, which uses priority values to perform scheduling, and a priority function which defines priorities for different elements of the system. The priority function is evolved for a given scheduling environment using genetic programming, while the meta-algorithm is defined manually. This allows easier creation of various heuristics in an arbitrary scheduling environment. To illustrate this methodology, we address the problems of one machine and job shop scheduling (multiple identical and unrelated machine environments were also investigated), combined with several real-world properties which complicate the application of existing heuristics. The properties include job weights, dynamic job arrivals, precedence constraints, sequence dependent setup times and combinations of those. The remainder of this paper is organized as follows: Section 2 describes the approach in more detail, and the next two sections show its application on single machine and job shop problem, respectively. Section 5 covers the choice of relevant parameters and Section 6 gives a discussion of the results, followed by a brief conclusion.

2 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. This sequence presents only a solution to the specific problem instance, which means that a new solution must be found for different initial conditions. With genetic programming, we have the ability to represent an *algorithm* which will be used to generate schedules for all the problem instances in a scheduling environment. The algorithm that genetic programming evolves is in the form of a *tree*, where tree nodes represent problem specific functions, variables (terminals) and commands.

The scheduling paradigm applied in this work is priority scheduling, in which certain elements of the scheduling system are assigned priority values. The choice of the next activity being run on a certain machine is based on their priority value. 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. In virtually all the literature on the subject the meta-algorithm part is never explicitly expressed but only presumed implicitly, which can lead to misunderstandings between different projects.

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

A simple example may be given for a one machine environment with static job availability (all jobs ready to begin at time zero), where the meta-algorithm is trivial:

while there are unscheduled jobs **do**
  wait until machine is ready;
  calculate priorities of all unscheduled jobs;
  schedule job with best priority;
end while

The priority function will depend on the given scheduling criteria; a simple function that minimizes total flowtime (the amount of time all the jobs spend in the system) is $\pi = w_j/p_j$, where w_j is relative weight and p_j the processing time of job j (also known as WSPT rule). In this work, the actual priority functions are evolved using GP, given some scheduling criteria as a measure of fitness.

The time complexity of priority scheduling algorithms depends on the meta-algorithm, but it is in most cases negligible compared to search-based techniques, which allows the use of this method in on-line scheduling [17] and dynamic conditions. For instance, a common meta-algorithm simply finds the best priority value among all the available jobs, which takes $\mathrm{O}(n)$ time. It is obvious that the priority function has to be previously evolved with genetic programming, which is a lengthy process, but this can be performed offline before the actual scheduling takes place. All the heuristics presented in this paper, both the existing and the evolved ones, produce schedules for several hundred instances in less than a second.

3 Single Machine Scheduling

3.1 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

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

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

$$T_w = \sum_j w_j T_j \tag{2}$$

and weighted number of late jobs as

$$U_w = \sum_j w_j U_j \ . \tag{3}$$

In case where a machine may need to process more than one *type* of job, there is sometimes a requirement to adjust the machine for the processing of the next job. If the time needed for adjusting depends on the previous and/or the following job, then this is referred to as *sequence dependent setup time* and must be defined for every possible combination of two jobs [12] [11]. This condition further increases the problem complexity for some scheduling criteria.

Additionally, if certain jobs cannot start until some other jobs have finished, then the problem includes *precedence constraints*. The precedence relation $\prec$ is defined for two jobs $J_i \prec J_k$ iff job J_i must finish before job J_k could be started. Precedence constraints are usually represented as a directed graph where nodes denote jobs and connections represent constraints.

In summary, we address the following variants (and their combinations) of a single machine scheduling problem:

- static or dynamic problem,
- with or without setup times,
- with or without precedence constraints.

3.2 Benchmark Scheduling Heuristics

The effectiveness of the presented approach is estimated by comparing the GP evolved priority rules with existing heuristics for a given variant of the problem and scheduling criteria. The basic heuristics used in this work are EDD (earliest due date [15][7]), WSPT (weighted shortest processing time [15][7][6]), MON (Montagne heuristic, [15][7][6]), RM/ATC (Rachamadugu & Morton heuristic, apparent tardiness cost [15][16][14]), XD (X-dispatch bottleneck heuristic [15][16]) and ATCS (apparent tardiness cost with setups [12][16]). We call them 'basic' because most of these can be further modified for use in dynamic problem variant or with existence of setup times.

3.3 Handling Dynamic Job Arrivals

In a dynamic environment, where jobs arrive over time, scheduling heuristics that presume all the jobs are available are modified so that the processing time of a job is increased by job's "time till arrival" (waiting time), defined as

$$wt_j = \max\{r_j - current\_time, 0\} \quad . \tag{4}$$

The question remains as to which jobs to include when calculating the priority function? It can be shown that, for any regular scheduling criteria [15], 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 unscheduled jobs (some scheduling software implementations also include this condition [8]). In other words, we may only consider jobs j for which

$$wt_j < \min_i \{p_i\}, \forall i : r_i \leq current\_time \quad . \tag{5}$$

Therefore, when testing in dynamic conditions, we use the following meta-algorithm with an arbitrary priority function:

while there are unscheduled jobs **do**
  wait until the machine and at least one job are ready;
  p_{MIN} = processing time of the shortest available job;
  calculate priorities of all jobs j with $wt_j < p_{MIN}$;
  schedule job with best priority;
end while

3.4 Handling Precedence Constraints

The existing heuristics can also be used with precedence constraints, as long as the scheduler considers only those jobs whose predecessors have finished. However, this would yield poor results since precedence information is not included in priority calculation. In this scheduling environment we include additional benchmark heuristics:

- *Highest level heuristic* (denoted HL) chooses the job with the highest level in the precedence graph, among the available jobs. If we define the *path length* as the sum of processing times of all jobs (nodes) in a path in the graph, then the level of a job is defined as the length of the longest path from that job to any job with no successors (i.e. any node with no child nodes).

- *Sidney algoristic* (denoted SIDNEY) is a more complex algorithm given in [5]. The word 'algoristic' is used because it can be shown that this procedure is optimal for assembly tree and near optimal for general tree precedence graphs [15], with weighted tardiness and weighted flowtime as objectives.

3.5 Experimental Setup

The genetic programming heuristics are evolved using a set of learning test cases - scheduling instances. Each scheduling instance is defined with the following parameters: the number of jobs n, their processing times, due dates and weights. Job release dates, setup times and precedence constraints may also be included for variants of the problem. If precedence constraints are taken into account, test cases are defined with the most general form of constraints, i.e. where precedence graphs do not exclusively take the form of chains, branching trees or assembly trees. The values of the above parameters are generated in accordance with methods and examples given in [7], [12], [11] and [15].

We define 100 scheduling instances that are used as test cases in learning process and additional 300 instances that are used for evaluation purposes only. In the experiments we follow the standard machine learning paradigm: GP is trained on learning set of test cases for a limited number of generations, which is repeated in 20 runs for each experiment. Only the best solutions (evolved heuristics) from each run are then compared with existing heuristics on the evaluation set and an average performance with standard deviation is given. Out of these, a single best heuristic is chosen for head-to-head comparison with existing algorithms.

3.5.1 Fitness Function

In the evolution process, a single scheduling criteria can be selected as fitness function, which is in the case of one machine scheduling either weighted tardiness or weighted number of late jobs (smaller values indicate greater fitness). Evaluation of scheduling heuristics involves 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

$$\overline{T_w} = \frac{\sum_{j=1}^{n} w_j T_j}{n \cdot \bar{w} \cdot \bar{p}} \ , \tag{6}$$

and normalized number of late jobs as

$$\overline{U_w} = \frac{\sum_{j=1}^{n} w_j U_j}{n \cdot \bar{w}} \quad , \tag{7}$$

where n represents the number of jobs in a test case, $\bar{w}$ the average weight and $\bar{p}$ the average duration of all jobs. The total quality estimate of a scheduling algorithm is expressed as the sum of normalized criteria over all the test cases.

3.5.2 Genetic Programming Functions and Terminals

The choice of functions (inner tree nodes) and terminal tree nodes is a crucial step in the overall optimization process since they must allow the program to use all the relevant information and form an efficient solution. We define the same function set for every scheduling environment and a different terminal set depending on the variant of the problem. The complete set of primitives used as tree elements for single machine problems is presented in Table 1.

3.6 Scheduling in Static Environment

In a static environment all jobs are available at time zero, hence the scheduling procedure is simple: every time the machine is available, the priorities of all available jobs are calculated and the best priority job is scheduled. This procedure is the same regardless of the priority function used (either existing or GP evolved). The solution of genetic programming is represented with a single tree that embodies the priority function.

Three series of experiments were conducted in this environment: the first for the simple static problem, second with additional sequence dependent setups and third with precedence constraints. In each series the genetic program was trained in 20 runs on learning set of test cases with weighted tardiness criteria (denoted 'Twt' in results) as a fitness function. Additionally, we also include the other non-trivial criteria (weighted number of late jobs, denoted 'Uwt') for better insight. Resulting best priority functions from each run are then compared with existing heuristics on the evaluation set of test cases; average achieved quality and standard deviation (on evaluation set) is included in the results. Apart from total criteria values, a good performance measure for a scheduling heuristic may be defined as the percentage of test cases in which the heuristic provided the best achieved result (or the result that is not worse than any other heuristic). We denote this value as the *dominance percentage* and also include it in the results.

Table 1
The function and terminal node set - one machine scheduling

Function name	**Definition**
ADD, SUB, MUL, DIV	binary addition, subtraction, multiplication and protected division
POS	POS $(a) = \max\{a, 0\}$
Terminal name	**Definition**
Terminals used in every problem variant	
pt	nominal processing time of a job (p_j)
dd	due date (d_j)
w	weight (w_j)
SL	positive slack, $\max\{d_j - p_j - time, 0\}$
Nr	number of remaining (unscheduled) jobs
SPr	sum of processing times of remaining jobs
SD	sum of due dates of all jobs
Additional terminals for dynamic environment	
AR	job arrival time (waiting time), $\max\{r_j - time, 0\}$
Additional terminals for sequence dependent setups	
STP	setup time from previous to current job j
Sav	average setup time from previous (l) to all jobs $\frac{1}{n-1}\sum_{j=1}^{n} s_{lj}$
Additional terminals with precedence constraints	
SC	number of immediate successors
LVL	job's level in precedence graph

For the first series, the GP evolved priority functions achieved mean best result of 1468.0 with $\sigma = 2.4$ in normalized weighted tardiness on evaluation set of test cases. A single priority function is compared with the appropriate existing heuristics and the results are shown in Table 2. It can be observed that the evolved priority function dominates over other heuristics in a majority of test cases, and is at the same time second best for the other non-optimized criteria.

In the second series additional sequence dependent setup times are included; here GP achieved mean best value of 2082.2 with $\sigma = 17.4$ and those results can be seen in Table 3. Finally, the evolved heuristics are compared with appropriate algorithms in a precedence constrained environment, for which

Table 2
Single machine, static weighted tardiness

Heuristic	**Normalized criteria**		**Dominance**	
	Twt	Uwt	Twt	Uwt
GP	**1468.1**	156.6	**70.8%**	25.2%
RM	1510.2	**147.0**	28.5%	**31.0%**
MON	1871.9	157.5	2.5%	11.3%
WSPT	2347.6	168.4	0.3%	20.3%
EDD	3958.8	307.3	21.5%	18.8%

Table 3
Single machine, static weighted tardiness with setups

Heuristic	**Normalized criteria**		**Dominance**	
	Twt	Uwt	Twt	Uwt
GP	**2075.1**	**182.9**	**67.5%**	**44.7%**
ATCS	2190.2	196.8	26.3%	22.5%
RM	2629.9	305.4	4.0%	0.7%
MON	2536.6	202.9	2.2%	12.3%
WSPT	3035.9	197.9	0.2%	24.5%

Table 4
Single machine, static weighted tardiness with precedence constraints

Heuristic	**Normalized criteria**		**Dominance**	
	Twt	Uwt	Twt	Uwt
GP	**4353.4**	279.7	**53.0%**	13.0%
SIDNEY	4421.0	**244.9**	40.2%	**48.0%**
RM	5015.9	260.5	3.3%	23.2%
MON	5078.8	263.7	1.8%	11.7%
HL	5955.9	308.3	0.0%	1.2%

the mean weighted tardiness value is 4351.7 with $\sigma = 40.4$; the results are compared in Table 4.

The results show that the presented approach has the ability to generalize and perform well on unseen instances of scheduling problems. This is further investigated in the next subsection.

Table 5
Single machine, dynamic weighted tardiness

Heuristic	**Normalized criteria**		**Dominance**	
	Twt	Uwt	Twt	Uwt
GP	**331.3**	**94.7**	**75.8%**	**44.0%**
XD	389.7	97.0	22.5%	31.7%
RM	451.7	105.3	9.5%	17.0%
MON	623.1	108.3	3.0%	8.8%
WSPT	845.0	100.8	0.0%	20.3%

3.7 Scheduling in Dynamic Environment

In a dynamic environment the jobs have distinct release times, which GP may use in the construction of the priority function. In this environment we included the appropriate existing heuristics for comparison. For this experiment we achieved mean best result of 331.9 with $\sigma = 2.8$ and the comparison results are shown in Table 5.

3.8 Arbitrary Scheduling Environments

For each presented variant of the problem (setup times, constraints, ...) there is usually a heuristic that best suits given environment. However, if the problem includes combinations of the variants, scheduling becomes more complicated and the choice of suitable heuristic is not trivial. To illustrate this, we conducted another two series of experiments: the first one is concerned with a dynamic environment with setup times, in which GP evolved heuristic achieved mean best result of 642.9 with $\sigma = 33.2$ in normalized weighted tardiness (Table 6). In the second experiment the static environment includes setup times and precedence constraints, and the results obtained were 5649.6 with $\sigma = 79.6$ (Table 7).

In an arbitrary scheduling environments such as these, GP evolved heuristic clearly dominates over existing algorithms, which is not surprising: for a specific combination of environment characteristics, no suitable heuristic may even exist that will fit the given conditions.

Table 6
Single machine, dynamic weighted tardiness with setups

Heuristic	**Normalized criteria**		**Dominance**	
	Twt	Uwt	Twt	Uwt
GP	**642.6**	**115.5**	**60.7%**	**54.8%**
ATCS	773.6	138.7	21.5%	12.0%
XD	813.3	180.8	13.7%	2.7%
MON	943.6	148.7	0.7%	5.8%
WSPT	1191.1	123.0	0.2%	25.0%

Table 7
Single machine, static weighted tardiness with setups and precedence constraints

Heuristic	**Normalized criteria**		**Dominance**	
	Twt	Uwt	Twt	Uwt
GP	**5643.6**	297.9	**74.8%**	21.8%
ATCS	6282.6	**289.8**	7.3%	**27.2%**
RM	6361.2	306.3	7.5%	10.5%
SIDNEY	6352.8	293.5	4.8%	21.8%
MON	6416.3	299.2	3.7%	12.7%

4 Job Shop Scheduling

4.1 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). Duration of one operation of job j on machine i is denoted with p_{ij}. 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. We define *normalized* weighted flowtime of a set of jobs as

$$\overline{F_w} = \frac{\sum_{j=1}^{n} w_j F_j}{n \cdot \bar{w} \cdot \bar{p}} \ , \tag{8}$$

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

$$\overline{C_{max}} = \frac{\max\{C_j\}}{n \cdot \bar{p}} \quad . \tag{9}$$

Although the jobs are considered to be available from time zero, 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 therefore modify the processing time of an operation as in the single machine dynamic problem (inserted idleness approach).

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 **do**
  wait for a machine with pending operations;
  p_{MIN} = processing time of the shortest available operation;
  calculate priorities of all operations with waiting time less than p_{MIN};
  schedule best priority operation;
  update machine and job's next operation ready time;
end while

The choice of operations considered for scheduling is still restricted to those operations whose waiting time (4) is smaller than the duration of the shortest available operation.

4.2 Benchmark Scheduling Heuristics

In efficiency comparison we used the following job shop heuristics: WSPT, processing time to the total work remaining (WSPT/TWKR), weighted total work remaining (WTWKR), dynamic slack per remaining process time (SLACK/TWKR), COVERT (cost over time) and Rachamadugu & Morton job shop heuristic (RM). Each heuristic is described with its priority function; detailed descriptions of the listed heuristics can be found in [3] and [15].

Table 8
The function and terminal node set - job shop problem

Function name	**Definition**
ADD, SUB, MUL, DIV, POS	as in Table 1
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: $\max\{r_{ij} - time, 0\}$, 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
TWKr	processing time of remaining operations of a job
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

4.3 Scheduling with GP in Job Shop Environment

As in the single machine case, the solution of genetic programming is a single tree which represents the priority function to be used with the given 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 8. In this environment we define 80 test cases for learning and 80 evaluation test cases, in addition to 80 instances taken from [18], used for evaluation only. The evaluation process is the same as in the single machine problem.

Two series of experiments were conducted in job shop environment, both concerned with static job shop scheduling: in the first one weighted job tardiness is minimized and total makespan in the second one. In the first series the obtained normalized weighted tardiness was 151.5 and $\sigma = 2.0$ on the evaluation set of test cases and the overall performance is presented in Table 9. In the second series the achieved makespan was 18.12 with $\sigma = 0.51$, whereas the algorithm performance is shown in Table 9.

Table 9
Job shop, weighted tardiness optimization

Heuristic	**Normalized criteria**				**Dominance**			
	Twt	Uwt	Fwt	Cmax	Twt	Uwt	Fwt	Cmax
GP	**151.5**	**123.7**	**393.5**	21.8	**82%**	**65%**	**73%**	0%
RM	165.2	125.4	420.7	20.2	12.5%	55%	1.9%	3.1%
COVERT	231.9	143.5	506.0	**19.5**	0%	44.4%	0.6%	**38.8%**
WSPT	175.6	128.3	400.1	20.2	2.5%	55%	6.3%	3.8%
SPT/TWKR	230.6	141.5	494.6	19.6	0.6%	42.5%	0.6%	33.8%
WTWKR	179.6	125.9	398.1	21.1	1.9%	63.1%	13.1%	3.8%
SL/TWKR	259.3	147.2	534.8	20.3	0%	44.4%	0%	16.3%

Table 10
Job shop, makespan optimization

Heuristic	**Normalized criteria**				**Dominance**			
	Twt	Uwt	Fwt	Cmax	Twt	Uwt	Fwt	Cmax
GP	339.1	155.2	624.3	**18.4**	0%	43.7%	0%	**80%**
RM	**165.2**	**125.4**	420.7	20.2	**65.6%**	66.3%	34.4%	1.3%
COVERT	231.9	143.5	506.0	19.5	0.6%	44.4%	0.6%	12.5%
WSPT	175.6	128.3	400.1	20.2	35.6%	55%	**46.9%**	1.3%
SPT/TWKR	230.6	141.5	494.6	19.6	3.8%	44.4%	3.8%	8.1%
WTWKR	179.6	125.9	**398.1**	21.1	21.9%	**72.5%**	41.9%	1.3%
SL/TWKR	259.3	147.2	534.8	20.3	0.6%	46.9%	0%	2.5%

5 Genetic Programming Parameters

In this section we present a brief exploration of the influence that GP parameters have on the efficiency of the proposed approach and justify the choice of parameter values in previous sections. The most important parameters in our view are those that influence the ability of GP to generalize over unseen problem instances. Among the possible choices, we identify the stopping criteria, learning set size, maximum tree depth and the population size, all of which are not specific to the actual problem only but are common to any machine learning application using GP. The results are presented both as average normalized criteria value and average dominance percentage for static one machine weighted tardiness problem, with 20 runs conducted for every parameter value.

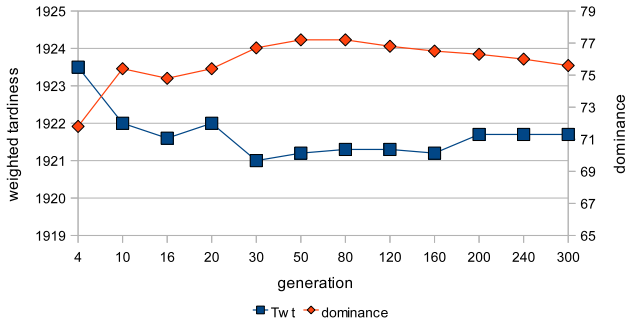

Fig. 1. Parameter influence: number of generations

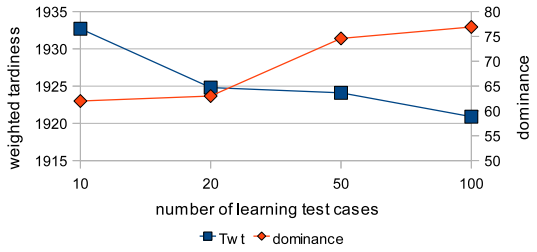

Fig. 2. Parameter influence: learning set size

The first parameter to investigate was the *number of generations* after which the evolution is stopped; the possible values and related results showing efficiency on unseen set of evaluation test cases are shown in Fig. 1. Based on the results, we chose 50 generations as stopping criteria, which coincides with Koza's original recommendations for GP. Note that the differences in efficiency are relatively small in comparison to the results of the existing heuristics.

Learning set size and the choice of learning instances is of great importance in machine learning applications in general. Among the tested values, the set size of 100 cases showed to be most adequate for this application (Fig. 2), given the evolved heuristic efficiency on evaluation problem instances.

The *population sizes* in GP are commonly greater than in other evolutionary algorithm variants; following previous best practices we experimented with sizes as shown in Fig. 3. The chosen population size was 2000 since the improvement with greater sizes is not significant to justify the linear increase in computation time.

Finally, the size of the evolved trees is traditionally regulated with maximum tree depth parameter, which greatly influences the evolution computational time and solution parsimony. Based on the results shown in Fig. 4, we chose the maximum depth of 14.

In general, we can conclude that the GP related parameters have only a small influence on the overall efficiency, which may be considered as a good charac-

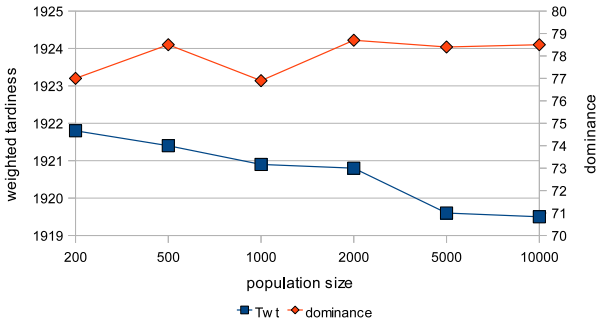

Fig. 3. Parameter influence: population size

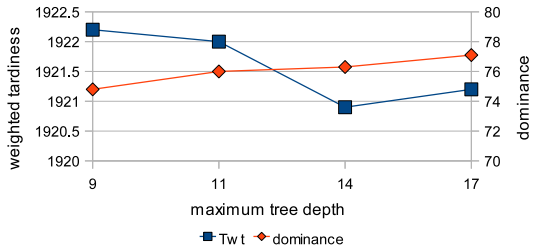

Fig. 4. Parameter influence: maximum tree depth

teristic for future applications of this type. Furthermore, the performance is quite consistent and thus only a small number of runs is needed to achieve a high certainty of producing a good result.

6 Discussion

It can be seen that GP can easily outperform other heuristics for an arbitrary scheduling criteria. On the other hand, it is not very likely that a single heuristic, either existing or evolved, will dominate over more than one criteria. This is particularly true in the presented GP system guided with a single fitness function. If we are after a scheduler with good overall performance, then it is maybe advisable to take some 'general use' existing heuristic, but if we want to maximize efficiency for a single criterium, then the evolved heuristics represent a good alternative. Furthermore, the user may also define its own criteria tailored to specific conditions (e.g. a linear combination of existing ones, dynamically varying criteria in response to current state etc.) and evolve a suitable heuristic.

The more common stopping criteria in machine learning applications also uses a validation subset of test cases to estimate good generalization performance. Preliminary results show that this technique is not readily applicable to this problem, since the performance on the validation set is very erratic from generation to generation. We are currently investigating the possibilities

of incorporating the validation set into the evolution process.

The scheduling rule obtained with GP can be applied in a dynamic environment where the system parameters are allowed to change during execution. In that kind of situation it is not practical to employ a search based procedure because it may take more time that we are willing to lose and it has to be adapted to take into account the fact that some of the jobs are already underway and additional constraints may need to be defined. On the other hand, GP evolved scheduling rule can give the solution in the form of the next state of the system practically instantaneously.

It should be stressed that the aim of the presented approach is *not* the finding of (near)optimal schedules, but the ones of acceptable quality in a small amount of time. It is clear that the solution provided with the *search-based* methods, i.e. the ones that search the space of possible schedules, will in most cases be better. However, those methods are generally not applicable in situations where we need to make a schedule repeatedly, and have to make it in a restricted time, or have to respond quickly to changes in environment (such as on-line scheduling, machine failures, job parameter changes etc). Examples may include batch scheduling in multi-user grid or cluster environment or even process scheduling in embedded systems, which describe the conditions that the priority scheduling, applied in this work, is intended to be used in.

7 Conclusion

This paper shows how genetic programming can be used to build scheduling algorithms for a specific environment with arbitrary scheduling criteria. The scheduling heuristic is composed of two parts: a meta-algorithm, which is defined manually, and a priority function, which is evolved by GP. An evolved GP priority rule, in the form of a priority function, could be implemented into other scheduling systems, or even used by practitioners in the field without additional software (as some existing heuristics are).

The results are promising, as for given environments the evolved heuristics exhibit performance that is equivalent or better than human-made algorithms. Heuristics obtained with GP have shown to be especially efficient in cases where no adequate algorithms exist, and we believe this approach to be particularly useful in those environments.

References

[1] T. P. Adams, Creation of simple, deadline, and priority scheduling algorithms using genetic programming, in: Genetic Algorithms and Genetic Programming at Stanford 2002, 2002.

[2] B. L. Atlan, J. Polack, Learning distributed reactive strategies by genetic programming for the general job shop problem, in: Proceedings 7th annual Florida Artificial Intelligence Research Symposium, IEEE, IEEE Press, 1994.

[3] Y.-L. Chang, T. Sueyoshi, R. Sullivan, Ranking dispatching rules by data envelopment analysis in a job shop environment, IIE Transactions 28 (8) (1996) 631.

[4] V. Cheng, L. Crawford, P. Menon, Air traffic control using genetic search techniques, in: IEEE International Conference on Control Applications, IEEE, Hawai'i, 1999.

[5] B. Dharan, T. Morton, Algoristics for single machine sequencing with precedence constraints, Management Science 24 (1978) 1011–1020.

[6] C. Dimopoulos, A. Zalzala, A genetic programming heuristic for the one-machine total tardiness problem, in: Proceedings of the Congress on Evolutionary Computation, vol. 3, 1999.

[7] C. Dimopoulos, A. M. S. Zalzala, Investigating the use of genetic programming for a classic one-machine scheduling problem, Advances in Engineering Software 32 (6) (2001) 489.

[8] A. Feldman, M. Pinedo, X. Chao, J. Leung, Lekin, flexible job shop scheduling system, http://www.stern.nyu.edu/om/software/lekin/ (2003).

[9] J. V. Hansen, Genetic search methods in air traffic control, Computers and Operations Research 31 (3) (2004) 445.

[10] A. Jones, L. C. Rabelo, Survey of job shop scheduling techniques, Tech. rep., NISTIR, National Institute of Standards and Technology, Gaithersburg (1998).

[11] S. M. Lee, A. A. Asllani, Job scheduling with dual criteria and sequence-dependent setups: mathematical versus genetic programming, Omega 32 (2) (2004) 145–153.

[12] Y. H. Lee, K. Bhaskaran, M. Pinedo, A heuristic to minimize the total weighted tardiness with sequence-dependent setups, IIE Transactions 29 (1997) 45–52.

[13] K. Miyashita, Job-shop scheduling with gp, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Morgan Kaufmann, 2000.

[14] R. Mohan, V. Rachamadugu, T. E. Morton, Myopic heuristics for the weighted tardiness problem on identical parallel machines, Tech. rep., The Robotics Institute, Carnegie-Mellon University (1983).

[15] T. E. Morton, D. W. Pentico, Heuristic Scheduling Systems, John Wiley & Sons, Inc., 1993.

[16] M. Pfund, J. W. Fowlera, A. Gadkaria, Y. Chen, Scheduling jobs on parallel machines with setup times and ready times, Computers and Industrial Engineering 54 (4) (2008) 764–782.

[17] M. Pinedo, Offline deterministic scheduling, stochastic scheduling, and online deterministic scheduling: A comparative overview, in: J. Y.-T. Leung (ed.), Handbook of Scheduling, chap. 38, Chapman & Hall/CRC, 2004.

[18] E. Taillard, Scheduling instances, "http://ina.eivd.ch/Collaborateurs/etd/problemes.dir/ordonnancement.dir/ordonnancement.html" (2003).

[19] S. S. Walker, R. W. Brennan, D. H. Norrie, Holonic job shop scheduling using a multiagent system, IEEE Intelligent Systems (2) (2005) 50.

[20] W.-J. Yin, M. Liu, C. Wu, Learning single-machine scheduling heuristics subject to machine breakdowns with genetic programming, in: Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, IEEE Press, 2003.