Applied Soft Computing 48 (2016) 419–430

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Adaptive scheduling on unrelated machines with genetic programming

Marko Đurasević*, Domagoj Jakobović, Karlo Knežević

University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia

ARTICLE INFO

Article history:
Received 1 May 2015
Received in revised form 1 April 2016
Accepted 13 July 2016
Available online 20 July 2016

Keywords:
Scheduling on unrelated machines
Genetic programming
Priority scheduling

ABSTRACT

This paper investigates the use of genetic programming in automatized synthesis of heuristics for the parallel unrelated machines environment with arbitrary performance criteria. The proposed scheduling heuristic consists of a manually defined meta-algorithm which uses a priority function evolved separately with genetic programming. In this paper, several different genetic programming methods for evolving priority functions, like dimensionally aware genetic programming, genetic programming with iterative dispatching rules and gene expression programming, have been tried out and described. The performance of the suggested approach is compared to existing scheduling heuristics and it is shown that it mostly outperforms them. The described approach could prove useful when used for optimizing scheduling criteria for which no adequate scheduling heuristic exists.

1. Introduction

Scheduling can be defined as a decision-making process concerned with the allocation of scarce resources to tasks over a given time period in order to optimize one or more objectives [1]. Unfortunately, most of the important scheduling objectives represent NP-hard problems. As a consequence, no efficient algorithms are available for obtaining the optimal solution for a given objective. Therefore, solutions are usually obtained by using heuristic algorithms. In this context, we divide the algorithms in two groups: the first group consists of metaheuristics which search the space of solutions (schedules) to find the best one; the second group consists of problem-specific heuristics that construct the solution using some features of the problem [2].

Since most scheduling problems are combinatorial by nature, *search-based* metaheuristic methods (such as genetic algorithms, ant colony optimization, particle swarm optimization, etc.) can be used to search the space of solutions. The solutions obtained by such methods are often of a good quality, mostly better than the solutions obtained by constructive heuristics. Unfortunately, these methods require a substantial computational time in order to obtain solutions of acceptable quality (since they search the entire solution space). Another disadvantage of these methods is that they are generally not applicable in dynamic conditions, in which a constant adaptation to the changing conditions may be needed (e.g. unplanned arrival of new jobs, machine outages, etc).

Constructive scheduling heuristics, on the other hand, do not search the space of all possible solutions, but instead directly build the solution (schedule). Because of that, these heuristics can quickly react to changes in the environment, making them applicable in dynamic conditions. The advantage of these heuristics over the *search-based* methods is that their computational complexity is almost negligible. However, constructive scheduling heuristics also cope with a certain number of problems. For instance, it is often hard to select the optimal heuristic for the given criteria and problem instance. This was shown in [3] where evolutionary algorithms were used in order to create problem instances for the job shop scheduling problem, on which certain constructive scheduling heuristics made poor choices, which in the end resulted in schedules of low quality.

It should also be noted that such heuristics are not designed to optimize arbitrary criteria which could be defined by the user (it would be necessary to design a new heuristic to handle such a case). When all this is considered, it can be concluded that selecting an appropriate scheduling policy is not easy and that a heuristic for optimizing a given criteria might not even exist.

Genetic programming (GP), although rarely used for solving scheduling problems, is very suitable for searching the space of algorithms which, in turn, can provide solutions to scheduling problems. Recently, GP has been used to evolve scheduling policies for a wide variety of environments (single machine scheduling

* Corresponding author.
E-mail addresses: marko.durasevic@fer.hr (M. Đurasević), domagoj.jakobovic@fer.hr (D. Jakobović), karlo.knezevic@fer.hr (K. Knežević).

http://dx.doi.org/10.1016/j.asoc.2016.07.025
1568-4946/

[4–6], job shop scheduling [7–13], parallel proportional machine scheduling [14], airplane scheduling in air traffic control [15,16], scheduling in semiconductor manufacturing [17]). In addition, a recent survey about evolving dispatching rules with GP has been conducted by Branke et al. in [18]. This method has proven to be very successful, because not only does it allow to create scheduling policies for arbitrary criteria, but it also provides solutions which are on par with solutions obtained by heuristic methods.

In this paper we describe the approach of using GP for creating scheduling policies for the unrelated machines environment. We also describe several variants for this approach, used in order to obtain better results. Some of these modifications have been used in other machine environments, but to our knowledge, they have not yet been applied in the unrelated machines scheduling. We compare all of these variants against each other and additionally compare them to several existing approaches for creating schedules in order to assess the quality of the achieved solutions. In that regard this paper can be viewed as a continuation in comparing different optimization approaches, which has previously been conducted by Nguyen et al. [19] and Branke et al. [20].

The remainder of the paper is organized as follows: in Section 2 the definition of scheduling in the unrelated machines environment is given. Section 3 gives a short overview of genetic programming and its usage in the creation of scheduling policies. In Section 4 various GP optimizations are described. Section 5 describes the benchmark set and the results for all the algorithms. Section 6 delivers a short discussion on the results. Finally, Section 7 gives a short conclusion.

2. The unrelated machines environment

In the unrelated machines environment, a number of n jobs compete in order to be processed on one of the m machines. All jobs have a processing time p_{ij}, which determines the time which is needed for the job with the index j to be processed on the machine with the index i, as well as a release time r_j (ready time) which determines when the job becomes available for scheduling. Jobs may also have additional properties, like a due date d_j and a weight w_j (which determines the importance of the job). In this paper we considered the more complex problem variant which includes relative importance of a job, given by its weight. Solving for this variant can also solve the simpler problem with unweighted jobs.

2.1. Scheduling criteria

The most common scheduling criteria which are used for this environment include tardiness, number of tardy jobs, flow-time and makespan. First, let us define those criteria for a single job. Let C_j denote the finishing time of the job j. We can then define tardiness (the amount of time that a job was late) of job j as:

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

Similarly, flow-time, the amount of time a job spends in the system, can be defined as

$$F_j = C_j - r_j. \tag{2}$$

We will also define an additional measure which determines if a job is tardy or not

$$U_j = \begin{cases} 1: & T_j > 0 \\ 0: & T_j = 0 \end{cases}. \tag{3}$$

Using criteria for individual jobs, criteria for the entire schedule are defined. The makespan is defined as the maximum finishing time of all the jobs in the set

$$C_{\max} = \max\{C_j\}. \tag{4}$$

The other criteria are often defined as weighted sums: weighted tardiness

$$T_w = \sum_j wT_j, \tag{5}$$

weighted flow-time

$$F_w = \sum_j wF_j, \tag{6}$$

and weighted number of tardy jobs

$$U_w = \sum_j wU_j. \tag{7}$$

2.2. Scheduling conditions

Based on the availability of the job parameters, scheduling can be performed in different conditions. If all the parameters are known before the jobs are ready, then the schedule can be produced before the system starts its execution. This type of scheduling is called off-line or static scheduling. Search-based methods are most often used to create schedules for this type of scheduling conditions.

On the other hand, if no information about the jobs is available until the job has arrived into the system and no information is available about future jobs either, then such a scheduling process is called on-line scheduling. Heuristic scheduling methods, and the GP-based scheduler described in the next section, are almost always used in this kind of scheduling conditions. In addition, both of these approaches can be used for off-line scheduling as well, but in that case they are generally less efficient than the search-based methods.

3. Scheduling in the unrelated machines environment using genetic programming

3.1. Genetic programming

Genetic programming (GP) [21] is an evolutionary algorithm which is used to discover functions or programs which provide a solution to a given problem. In the algorithm, these solutions are represented in the form of a tree. The tree consists of two types of nodes, namely functional nodes (which represent certain arithmetic, boolean or other kind of functions) and terminal nodes which represent input variables and constants.

The idea behind this approach is to simulate the process of evolution. At the beginning of the algorithm, a random number of potential solutions is generated. Each solution receives an estimation of how "good" the solution is, which is measured on some predefined test cases. This estimation is called the *fitness* of the solution. The algorithm simulates natural selection such that the "better" individuals (solutions) have a higher probability of survival, while on the other hand, "worse" individuals have a smaller probability to survive. The individuals which survived the selection participate in crossover, which is a genetic operator that combines two individuals to form a new, hopefully better, individual. After the new individual is created, another operator called mutation is applied to the newly created individual. This operator changes, with a certain probability, some elements of the individual in order to introduce new elements into the solutions. This cycle of selection,