Evolutionary Algorithms for the Resource Constrained Scheduling Problem

Toni Frankola, Marin Golub, Domagoj Jakobovic
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, Zagreb, Croatia
domagoj.jakobovic@fer.hr

Abstract. *This paper investigates the use of evolutionary algorithms for solving resource constrained scheduling problem which belongs to the class of NP complete problems. The problem involves finding optimal sequence of activities with given resource constraints. Evolutionary algorithms used in this paper are genetic algorithms and genetic programming, for which adequate scheduling mechanisms are defined. Presented solutions are compared with existing heuristics or optimal results.*

Keywords. resource constrained scheduling, priority scheduling, genetic programming

1. Introduction

The nature of the problem of resource constrained scheduling allows users to employ diverse techniques to find solutions. Various authors [9][10] worked with different techniques to find solutions for single instances of these problems. The downside of these methods is that they need to be adapted for each variation of the problem and their performance deteriorates as number of activities grows [10].

Genetic algorithms (GA) [14] were applied to this problem by a number of authors [9]. They used different approaches in representing a solution to this problem and achieved respectable results. As a technique, genetic algorithms have their advantages but uncertainty and amount of time needed makes them inappropriate for dynamic environments, where scheduling conditions may change over time, and for projects with large number of activities.

Genetic programming (GP) [1][12] was rarely used for solving these problems as this technique is not appropriate for finding optimal solutions (schedules) [5]. On the other hand, genetic programming is an ideal technique when searching for simple and quick algorithms that can provide solutions to the problem. Consequently, the algorithms obtained in this way may serve to produce schedules of a high quality in a short amount of time. Moreover, evolved algorithms can be easily applied to all problems from the problem domain, i.e. they are able to solve a problem that was not 'seen' before.

This paper describes the use of GA and GP for finding the schedules for resource constrained projects. We present two approaches for solving this problem: with genetic algorithms, the appropriate solution representation allows finding single schedules of a very good quality. With genetic programming we describe a methodology to evolve scheduling heuristics in the form of priority rules that can be used to find a solution of an acceptable quality in a small amount of time. The latter approach can be used in a dynamic environment where the system parameters may change over time – i.e. the activities' parameters may vary, the resources may become unavailable, new activities may appear etc. The evolved heuristics can react to these changes during the run, thus eliminating the need of repeated time costly search for new schedules. This approach has, to the best of our knowledge, not been previously used for solving resource constrained project scheduling. In the next section we present the resource constrained project scheduling model. The following sections describe the methodology of finding the schedules with genetic algorithms and genetic programming.

2. Resource Constrained Project Scheduling

The resource constrained project scheduling problem (RCPSP) can be formulated as follows [10]. A project is a set of $n+2$ activities that need to be processed $J=\{1,2,...,n,n+1,n+2\}$. Activities marked as 1 and $n+2$ are considered dummy activities marking source and sink (end) activities. These activities are not considered in resource allocation and have a duration of 0 time units.

In order for a project to complete, all activities from set J must be completed.

There are a total of k resources available during whole duration of project. Resources are in the set $K = \{1, 2, ..., k\}$. Any given resource is limited with its capacity R_k in each time period. The duration of an activity is denoted as d_j and the amount of resource k used by an activity j in each time unit is denoted as $r_{j,k}$.

The activities of a project are described with two constraints:

- precedence constraint: an activity j cannot start before all preceding activities are completed;
- resource constraint: in order for activity j to run it needs to reserve a certain capacity of resources. Resources are limited so that only a finite capacity of each resource is available in each time unit.

The goal of RCPSP is usually to find a minimal project duration for which all resource constraints are met. Additionally, activities are generally forced to run without interruption, i.e. once activity is activated it cannot be stopped for its total duration time.

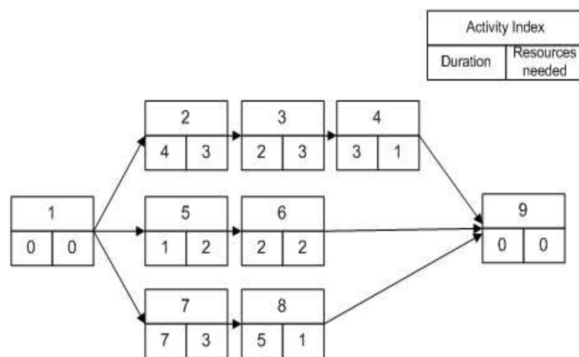

Figure 1. A sample project

A simple project is shown in Fig. 1. The project contains nine activities that have to be executed, where activities 1 and 9 are dummy activities that have duration of 0 units and do not require any resources for execution. The figures below each activity denote duration and capacity of resource needed. Only one resource (R1) is required for this project and there are four units of this resource available in each time unit. With ideal scheduling this project has optimal makespan of 16 time units. The optimal sequence of activities is shown in Fig. 2.

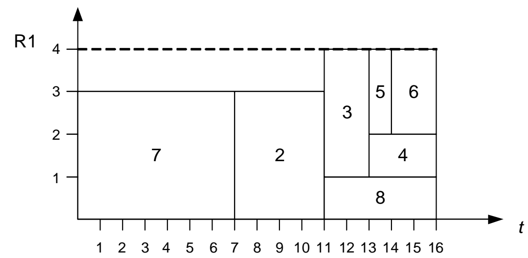

Figure 2. Optimal sequence of activities

3. Priority Scheduling With Genetic Algorithms

In RCPSP the greatest problem with genetic algorithms, and evolutionary algorithms in general, is finding a suitable chromosome representation. Representing a chromosome in this problem as a sequence of activities is possible but would require a complex functional set. If a chromosome is coded as sequence of activities, standard crossover and mutation operators over an individual would often result in non-feasible solutions that do not satisfy the precedence constraints. A number of authors [1][3][7][8][13][17][17] have looked into this problem and found different solutions. Traditional research was concentrated on operational research [10] and in last years numerous new heuristics were proposed including genetic algorithms, tabu search, simulated annealing and ant systems [9]. Different approaches have been taken for solving this problem with genetic algorithms, including priority based encoding [13], sequence scheduling and rule based scheduling [3].

In this work the priority based chromosome representation was used for GA, i.e. a chromosome is a list of priority values for all activities in a project. The chromosomes are encoded as permutations of distinctive values so that no two activities have the same priority. The choice of the next activity that can book resources is based on their respective priority and resource availability. For example, the chromosome $(0, 6, 5, 2, 3, 1, 7, 4, 0)$ would, for the project in Fig. 1, yield the schedule in Fig 2. (dummy activities 1 and 9 always have priority of 0).

Initial population is generated randomly (i.e. we do not presume we already have the best known or any other solution).

3.1. Calculating Project Makespan from Priority Sequence

The method for calculation of project makespan uses the list of priorities of project activities as an input parameter. With given priorities for each activity, the method virtually 'runs' current project. All constraints defined in project network plan and resource constraints are taken into account. When more than one activity becomes available, the activity with higher priority will be granted all the requested resources. The activity with lower priority will then be granted resources only if resource constraints are not violated. The procedure for calculating project makespan can be described with the algorithm in Fig. 3.

```
procedure(calculate makespan)
for (each activity i )  {
    priority P_i = extract from chromosome;
}
while (there are unprocessed activities) {
    activeActivities = activities whose
predecessors are completed;
    sort activeActivities by P_i ;
    while (there are activities in
activeActivities and resources are
available) {
        assign resources to activities;
    }
    update projectDuration;
}
```

Figure 3. Project simulation algorithm

3.2. Test Cases

The algorithms described in the paper were tested on resource constrained projects from PSPLIB developed by Kolisch et al [11]. The library for RCPSP contains 2040 projects with 30, 60, 90 or 120 activities. The projects from the library also include solutions obtained by different methods which present either optimal, best known or lower bound solutions.

3.3. Fitness Function

Fitness function of a GA individual is defined simply as makespan of the project that was scheduled using activity priorities encoded in the individual. Genetic algorithm is set to find an individual with minimal fitness, meaning that chromosomes that produce lower makespan are considered to be better ones.

3.4. Results

We concluded two sets of experiments to evaluate genetic algorithm efficiency. In the first set of experiments, we ran GA on four randomly chosen projects from each project class (30, 60, 90 and 120 activities). These experiments served to find an adequate set of parameters and to show expected variations of GA results over multiple runs. The GA parameters were varied in terms of population size and total number of generations and 10 runs were executed for each parameter set and for each selected project. Table 1 shows the set of parameters for which the best results were obtained.

Table 1. The genetic algorithm parameters

Parameter / operator	Value / description
population size	500
selection	Steady state, tournament size 3
stopping criteria	Maximum number of generations (300) or maximum number of consecutive generations without best solution improvement (50)
crossover	50% probability, uniform vector crossover
mutation	5% probability, swap mutation

Instead of fitness value, which equals the project makespan, the results are expressed as the relative difference from the optimal (or best known) project makespan:

$$rel_diff = \frac{m_{GA} - m_{OPT}}{m_{OPT}}, \qquad (1)$$

where m_{GA} is makespan obtained with GA and m_{OPT} is known optimal makespan. The results from this set of experiments in the form of mean relative difference and standard deviation (in brackets) are shown in Table 2.

Based on those results, we chose the combination of 500 individuals in a population and maximum number of 300 generations (with at most 50 generations without improvement) as the adequate parameter setup. The second set of experiments served to show overall GA efficiency on different projects.

Rather than solving every test case in the test bed, we included random 10 projects from each project class and ran it with the chosen parameters. Each project is optimized only once and the results are expressed as the average relative difference from known optimum for each project group regarding the number of activities. The results for this set of experiments are shown in Table 3.

Table 2. Results with different GA parameters

Problem size (no. of activities)	Population size / Number of generations		
	100/200	200/300	500/300
30	3.57 % (σ = 0.0)	3.57 % (σ = 0.0)	3.57 % (σ = 0.0)
60	8.19 % (σ = 1.2)	7.34 % (σ = 1.1)	6.28 % (σ = 0.8)
90	9.72 % (σ = 1.2)	8.17 % (σ = 1.1)	7.75 % (σ = 0.7)
120	13.05 % (σ = 1.3)	12.47 % (σ = 1.4)	10.58 % (σ = 0.9)

Results in the table are comparable to those presented in [9]. Although genetic algorithm is able to find solutions with very small deviations from known optimal solutions, the described approach has disadvantages that prevent its commercial exploitation. Namely, GA cannot guarantee that a solution of an acceptable quality will be found in each run and the time for finding a solution can be longer than we are willing to lose, which could be impractical for implementation into commercial software. In our GA experiments, a single GA run lasted anywhere from several minutes for smallest 30-activities projects, to 9 hours for 120-activities projects. In the following section we present a possible use of genetic programming to amend these problems.

Table 3. Average GA results

Problem size	Average rel. difference to optimum
30 activities	1.36 %
60 activities	3.73 %
90 activities	1.92 %
120 activities	8.20 %

4. Priority Scheduling With Genetic Programming

The genetic algorithm described in the previous section deals with individuals that are lists of priority values for each activity in a scheduling instance. In other words, each GA individual can be decoded in a solution for a single RCPSP. For each new project the GA has to be restarted and the whole evolution process repeated to obtain a new solution, because the GA searches the space of solutions of the problem. With genetic programming, on the other hand, we have the ability to search the space of *programs* that provide a solution to the problem.

In this work we use the priority scheduling paradigm with GP approach. In priority scheduling, real world schedulers may use a scheduling rule that, given current conditions and certain activity properties, schedules activities based on their priorities. The term 'scheduling rule' in a narrow sense often represents only the priority function used to define relative importance of an activity, i.e. its priority. Examples of those functions include for instance LPT (longest processing time), LNS (largest number of successors) etc. The actual scheduling algorithm simply calls upon that function to determine which activity to start next.

In our implementation, the task of GP is to find an appropriate *priority function* that can be used to calculate priorities for all activities in any project. The priority function evolved with GP will therefore be used to obtain schedules for every new RCPSP instance. The individual of a genetic program is thus represented with a single tree that embodies the priority function (an example in Fig. 4 is given below). The variables that appear in the priority function (also called GP terminals) are various activity properties (i.e. duration, resource usage etc.) that may be used to assess activity importance. A single GP individual (single priority function) is evaluated on several projects (the learning set), since our aim is to evolve a priority rule that can be used in new unseen problem instances. After a GP run is finished, the best evolved priority rule is then tested on the evaluation set of RCPS problems.

The time complexity of scheduling with a priority rule is negligible compared to search based techniques such as GA, since the priority

function has a complexity of $O(1)$ and it is called only when a new activity may be started. For instance, we were able to provide solutions for all the test cases (described below) within a second. That is, of course, possible only after a priority function has been obtained by GP in one or more runs. It should be noted that a single GP run in our experiments usually takes several hours, but this can be done beforehand and not need to be repeated once an adequate priority rule is evolved. This approach is therefore especially suitable for use in dynamic conditions where system variables may change over time.

4.1. The Function and Terminal Set

The most important task for creating a good GP program is finding a sufficient set of operators and terminals (variables). The complete set of operators and terminals used in our genetic program is presented in Table 4.

Table 4. The operator and terminal set

Operator name	Definition
ADD, SUB, MUL	binary addition, subtraction and multiplication operators
DIV	protected division: $DIV(a,b)=\begin{cases}1, \text{ if } \lvert b\rvert<0.000001\\ a/b, \text{ otherwise}\end{cases}$
Terminal name	**Definition**
D	Activity duration (in time units)
RR	Total number of resources required for activity completion
RRT	Total number of resources times quantity required from each resource
ARU	Average resource usage
SC	Total number of activities that succeed current activity
PC	Total number of activities that precede current activity

As shown in the above table, all terminals are attributes of an activity. The terminal D represents the duration of the activity and is expressed as an integer number of time units. The terminal RR is a total number of resources required to run the activity (there are at most four different resources available in each test case in the test bed). Each activity (except for source and sink activity) must require at least one of four available resources and activities may require different quantities of different resources. The quantity of the requested resource cannot be changed during activity runtime. The terminal RRT (requested resources total) is defined as

$$RRT = D\sum_{i=0}^{k} Q_i \qquad (2)$$

and ARU (average resource usage) as

$$ARU = \frac{RRT}{RR} \qquad (3)$$

where Q_i is the quantity of resource i required by current activity in each time unit.

4.2. Fitness Function

Using the operators and variables defined above, the GP population may include any syntactically valid combination of those as individuals. Additionally, standard GP tree crossover and mutation operators, as well as tree creation algorithms, will always maintain the syntactic correctness. An example of a GP individual in this problem is shown in Fig. 4.

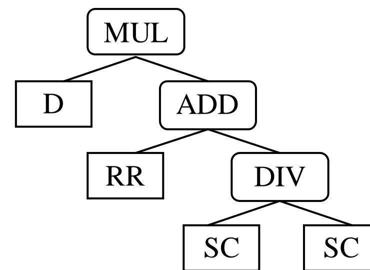

Figure 4. An example GP individual

The above individual represents the priority function $D\cdot(RR+SC/SC)$ which, when used as a priority value (for activity comparison), reduces to $D\cdot RR$. This rule will therefore favor activities that last longer and require a larger number of different resources. The fitness of this individual is determined as its performance over a number of learning test cases (as opposed to GA individual fitness which can only be evaluated against a single test case).

In the process of evaluation, an individual in GP population is used to calculate the

priorities for those activities that at a given moment compete for resource usage. When the priority is obtained, it is used in the same way as in the scheduling process with priorities decoded from a GA individual: the project makespan is determined using the simulation algorithm described in the previous section. The same individual is applied to generate schedules of all the projects in the learning set of test cases. The fitness value is then defined as the average relative difference from optimal (or best known) makespan value for all the test cases:

$$fitness_{GP} = \frac{1}{t} \sum_i \frac{m_i - m_{i,OPT}}{m_{i,OPT}} \qquad (4)$$

where i is the test case index, $m_{i,OPT}$ is the optimal value, m_i is makespan obtained from an individual and t is the total number of test cases. In our experiments the optimal or lower bound solution was always available, but in the case where they are not known, we may define the fitness function as the sum (or average) of total makespan values for all the test cases.

When GP is evolving on the training set, the fitness is evaluated as described in the algorithm in Fig. 5 which makes use of the makespan calculating procedure defined in the previous section.

```
procedure(evaluate GP individual)
for (each project P_i in training set TS ) {
    for (each activity j in project P_i ) {
      calculate priority using current GP
individual;
    }
    m_i = procedure(calculate makespan);
    update individual fitness;
}
```

Figure 5. Priority function evaluation

4.3. Test Cases

Genetic programming was tested on the same test data as genetic algorithms [11]. In our experiments we used two sets of projects. The first set (learning set) consisted of 10% of projects (204) randomly selected from database containing an equal combination of projects with 30, 60, 90 and 120 activities. The second set (evaluation set) contained all the other projects (1836 instances) from the database. The GP was run on the learning set and results shown here were then obtained by testing the best evolved priority function on the evaluation set.

4.4. Results

As with genetic algorithm, we experimented with a limited number of combinations of parameters for genetic programming. Specifically, we varied the size of the population and maximum depth of trees. All the other parameters of the evolution process were constant and they are presented in Table 5.

Table 5. The GP parameters

Parameter / operator	Value / description
selection	steady-state, tournament of size 3
stopping criteria	maximum number of generations (80) or maximum number of consecutive generations without best individual improvement (30)
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

The results achieved with different population sizes and tree depths are shown in Table 6, where for each combination of those parameters 15 GP runs were conducted. The values in the table represent mean fitness value (which is the difference from the optimal or best known solution) and standard deviation obtained on the evaluation set of test cases. It can be seen from the table that parameter changes did not have a substantial effect on the quality of evolved priority functions. Nevertheless, the combination with population size of 10000 and tree depth of 9 achieved slightly better results than the other combinations.

Mean fitness values obtained with GP evolved priority rule are in most cases worse than those obtained by running a GA on every project in the evaluation set. This is not surprising, as GA solves each test case

separately, but the time that GA would need to solve all the evaluation test cases could easily reach several weeks on a single computer, whereas with a (previously evolved) priority rule the schedules can be generated in less than a second.

Table 6. Average GP results

Population size	Tree depth	Mean (std. deviation)
2000	9	13.54 % (0.06)
5000	15	13.50 % (0.24)
10000	9	13.37 % (0.11)

It is interesting to see how a single evolved priority function behaves on different problems regarding its efficiency. The distribution of relative makespan differences on all the problems in evaluation set for a single GP priority rule is shown in Fig. 6.

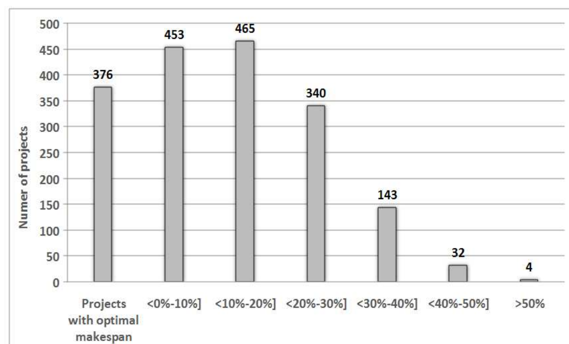

Figure 6. The distribution of makespan deviations from optimal value for a single GP priority function

4.5. Discussion

The distribution in Fig. 6 shows that GP priority rules can cope successfully with a majority of RCPS problems. There is still a number of projects in which the obtained rule was not of a great quality, but this phenomena is always present when using any other human made scheduling heuristic (on a general set of independent test cases, no single heuristic can exert better performance than any other heuristic on all the problem instances [15]).

We have to state that the proposed GP approach is *not* suitable in a situation where only the optimal or a near-optimal solution is allowed and there is ample time for schedule generation and we are certain that the system parameters will not change during system execution. If that is the case, we are probably better off with an algorithm that searches the space of schedules (such as GA, branch and bound etc.) and may use as much time as possible. However, if any of the previous assumptions do not hold, e.g. in on-line scheduling systems [16] or in computer cluster environment [3], the GP approach may offer a quick solution of an acceptable quality.

The described method of evolving priority functions with GP may be especially useful in specific user-defined scheduling environments where there are no suitable heuristics, or the existing ones are not directly applicable [6]. Furthermore, if one needs to obtain schedules with another scheduling criteria, the only adaptation we need to make is to define a different fitness function for the GP system.

It should be noted that the GP trees in our implementation usually include a number of *introns* – the parts of the priority function that do not contribute to the resulting function value (such as the division of two identical terminals, which always equals one). The greatest problem with GP priority functions in our view, however, is not their oversizing, but the possible inclusion of parts that in the majority of test cases do not contribute significally to the calculation of priority, but may be the cause of results of a lower quality obtained in some (previously unseen) problem instances. This phenomenon, as well as the ever present danger of overfitting the heuristics to the learning set of problems, is the main issue we still need to clarify.

5. Conclusion

In this work we have presented two approaches for finding schedules of RCPSP. The goal was to find a solving methodology that could be easily applied to real word problems. Besides finding a method that can produce accurate results, we wanted to find a method that is quick and whose quality is acceptable in each run. The two presented methods, GA and GP, share the same idea of evolution but they operate on a different principle. Although results achieved with GA are comparable with solutions found with other heuristics, this technique tends to be slow and unpredictable for implementation into commercial software for finding schedules. GP approach shares the same advantages as GA, but a given GP priority rule, evolved in the

form of a priority function, could be easily implemented into a commercial software product (or even used by practitioners in the field without additional software, as some human made heuristics are).

Furthermore, the scheduling rule obtained with GP can be applied in a dynamic environment where the project parameters and characteristics are allowed to change during the project execution. In that kind of situation it is not practical to employ a search based procedure, such as genetic algorithm, 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 the part of the project is already underway and additional synchronization constraints must 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.

Future work includes applying this method to different resource constrained libraries, larger projects and multimode RCPSP.

6. References

[1] Banzhaf W. et al. Genetic Programming – An Introduction. San Francisco: Morgan Kaufmann Publishers; 1998.

[2] Bartschi Wall M. A Genetic Algorithm for Resource-Constrained Scheduling. PhD thesis. Massachusetts Institute of Technology; 1996.

[3] Grudenic I, Bogunovic N. Analysis of Scheduling Algorithms for Computer Clusters. In: Proc. 31th Int'l Convention MIPRO 2008; Opatija, Croatia (*to appear*)

[4] Hartmann S. A competitive genetic algorithm for resource-constrained project scheduling. Naval Research Logistics; 1998.

[5] Jakobović D, Budin L. Dynamic Scheduling with Genetic Programming. Lecture Notes in Computer Science 2006; 3905:73-85.

[6] Jakobovic D, Jelenkovic L, Budin L. Genetic Programming Heuristics for Multiple Machine Scheduling. Lecture Notes in Computer Science 2007; 4445:321-330

[7] Kamaraien O, Ek V, Nieminen K, Ruuth S. Large scale generalized resource constrained scheduling problems: A genetic algorithm approach. IC-Parc, Imperial College, London; 2003.

[8] Kim J. A framework for integration model of resource-constrained scheduling using genetic algorithms. In: Proceedings of the 37th conference on Winter simulation; 2005; Orlando, Florida. Orlando: Winter Simulation Conference; 2005. p. 2119 - 2126

[9] Kolisch R, Hartmann S. Experimental investigation of heuristics for resource-constrained project scheduling: An update. European Journal of Operational Research 2006; 174(1):23-37.

[10] Kolisch R, Hartmann S. Heuristic algorithms for solving the resource-constrained project scheduling problem: Classification and computational analysis. In J.Weglarz, editor. Project scheduling: Recent models, algorithms and applications, pp 147–178. Kluwer Academic Publishers; 1999.

[11] Kolisch R, Spracher, A. PSPLIB — A project scheduling problem library. European Journal of Operational Research 1997; 96(1):205–216

[12] Koza J. Genetic Programming. Cambridge: MIT Press; 1992.

[13] Mendes J, Gonçalves J, Resende M. A Random Key Based Genetic Algorithm for the Resource Constrained Project Scheduling Problem. AT&T Labs Research Technical Report TD-6DUK2C; 2003.

[14] Michalewicz Z. Genetic Algorithms + Data Structures = Evolution Programs. New York: Springer-Verlag; 1994.

[15] Morton TE, Pentico DW. Heuristic Scheduling Systems. John Wiley & Sons, Inc; 1993.

[16] Pinedo M. Offline deterministic scheduling, stochastic scheduling, and online deterministic scheduling: A comparative overview. In: Leung JYT. editor. Handbook of Scheduling. Chapman & Hall/CRC; 2004.

[17] Sriprasert E, Dawood N. Genetic algorithms for multi-constraint scheduling: An application for the construction industry. Centre for Construction Innovation Research, University of Teeside; 2003.