Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 2380–2401
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Software project management with GAs

Enrique Alba *, J. Francisco Chicano

University of Málaga, Grupo GISUM, Departamento de Lenguajes y Ciencias de la Computación, E.T.S. Ingeniería Informática, Campus de Teatinos, 29071 Málaga, Spain

Received 4 February 2005; received in revised form 27 September 2006; accepted 24 December 2006

Abstract

A Project Scheduling Problem consists in deciding who does what during the software project lifetime. This is a capital issue in the practice of software engineering, since the total budget and human resources involved must be managed optimally in order to end in a successful project. In short, companies are principally concerned with reducing the duration and cost of projects, and these two goals are in conflict with each other. In this work we tackle the problem by using genetic algorithms (GAs) to solve many different software project scenarios. Thanks to our newly developed instance generator we can perform structured studies on the influence the most important problem attributes have on the solutions. Our conclusions show that GAs are quite flexible and accurate for this application, and an important tool for automatic project management.

Keywords: Automatic software management; Genetic algorithm; Project scheduling

1. Introduction

The high complexity of currently existing software projects justifies the research into computer aided tools to properly plan the project development. Current software projects usually demand complex management involving scheduling, planning, and monitoring tasks. There is a need to control people and processes, and to efficiently allocate resources in order to achieve specific objectives while satisfying a variety of constraints. In a general way, the project scheduling problem consists in defining which resources are used to perform each task and when each one should be carried out. Tasks may be anything from maintaining documents to writing programs, and the resources include people, machines, time, etc. The objectives are usually to minimize the project duration, to minimize the project cost, and to maximize the product quality [4]. In an real project, the manager wants an automatic plan which will reconcile as far as possible these three conflicting goals.

Some work exists which proposes and discusses advanced management techniques [2,22] and tools [15,17] which can help software managers in their work. Computers are usually applied at several steps of the

* Corresponding author. Tel.: +34 95213 3303; fax: +34 95213 1397.
E-mail addresses: eat@lcc.uma.es (E. Alba), chicano@lcc.uma.es (J. Francisco Chicano).

0020-0255/$ - see front matter
doi:10.1016/j.ins.2006.12.020

software management process. We can find expert systems to diagnose problems in software development [21], neural networks for deciding when to deliver software to the users [7], genetic algorithms for project scheduling [4], CASE tools for the knowledge management of software development [11], all of which together form a new field of knowledge related to computer assisted project management. In this paper we focus on the Project Scheduling Problem solved with genetic algorithms [10]. The issues addressed are related to the time, human skills, budget, and project complexity involved. All of these issues make our study more difficult and nearer to actual software project planning scenarios. We first define an optimization problem to deal with the search for highly efficient management and propose the use of genetic algorithms to solve this problem [1]. With the proposed tool, a project manager can evaluate different scenarios in order to later be able to take decisions on the actual project itself. We perform some *in silico* experiments [25] based on several automatically generated project scenarios.

The article is organized as follows. In Section 2 the Project Scheduling Problem is defined. Section 3 describes the genetic algorithms proposed and Section 4 discusses the representation of the individuals and the fitness function, two very important issues when applying GAs to any problem. We use an instance generator to automatically create the different project scenarios, which is described in Section 5. Finally, the experimental study and results are presented in Section 6, and some conclusions and future work are outlined in Section 7.

2. The project scheduling problem (PSP)

The PSP is related to the Resource-Constrained Project Scheduling (RCPS), an existing problem which has been extensively tackled in the literature using both exact techniques [6,19,24] and metaheuristic ones [12,18,20]. However, there are some differences between PSP and RCPS. Firstly, in PSP there is a cost associated with the employees and a project cost which must be minimized (in addition to the project duration). Additionally, in RCPS there are several kinds of resources while PSP has only one (the employee) with several possible skills. We should notice that PSP skills are different from RCPS resource types. In addition, each activity in the RCPS requires different quantities of each resource while PSP skills are not quantifiable entities. The problem as defined here is more realistic than the RCPS because it includes the concept of an employee with a salary and personal skills, also capable of performing several tasks during a regular working day. In [4] a genetic algorithm is used to solve this kind of problem with an approach which is similar to our statement. Let us specify the details of the problem tackled in this work.

The resources considered are people with a set of skills, and a salary. These employees have a maximum degree of dedication to the project. Formally, each person (employee) is denoted with e_i, where i ranges from 1 to E (the number of employees). Let SK be the set of skills, and s_i the ith skill with i varying from 1 to $S = |\mathrm{SK}|$. The skills of employee e_i will be denoted with $e_i^{\mathrm{skills}} \subseteq \mathrm{SK}$, the monthly salary with e_i^{salary}, and the maximum dedication to the project with e_i^{maxded}. Both the salary and the maximum dedication are real numbers. The former is expressed in fictitious currency units, while the latter is the ratio between the amount of hours dedicated to the project and the full length of the employee's working day. Let us consider an example to clarify the concepts. Let us suppose that we have a software company with five employees. We need to perform a software application for a bank presenting the scenario shown in Fig. 1.

In this figure we supply information about the different skills of the employees, their maximum dedication to the project at hand, and their monthly salary. For example, employee e_2, who earns \$2500 each month, is a database expert (s_3), a UML expert (s_4), and is able to lead a group of people (s_2). Her/his colleague, employee e_4, is also able to lead a group (s_2) and, in addition, s/he is a great programmer (s_1). These two employees and employee e_1 can spend all of their working day developing the application (maximum dedication equal to one) but this does not necessarily mean that they do so. On the contrary, employee e_3 can only dedicate half of her/his working day to the project. There may be several reasons for this fact: perhaps the employee has a part-time contract, or s/he has administrative tasks to carry out in the company during part of the day. Employee e_5 can work overtime, her/his maximum dedication is greater than one ($e_5^{\mathrm{maxded}} = 1.2$), and this means that s/he can work on the bank application up to 20% more than in an ordinary working day. In this way, we can model the extra time of the employees, a fairly "real world" feature included in the problem definition. However, the project manager must take into account that an overloaded employee can increase her/his mistake rate and,

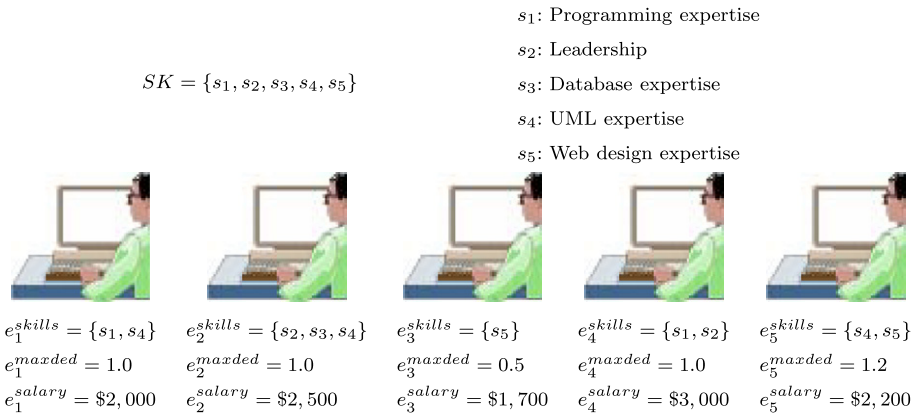

Fig. 1. Possible staff of an example software company.

consequently, the number of errors in the software developed. This leads to a lower quality of the final product and, possibly, to the need to correct or to re-develop the erroneous parts. In any case, the outcome may be an increase in the overall project duration. This does not affect the problem definition, it is a matter of psychology, but it is an important issue that project managers must take into account.

Let us leave the example for a moment and study how the tasks of a software project are modelled. The tasks are denoted with t_i, where i ranges from 1 to T (the number of tasks). Each task t_i has a set of required skills associated with it denoted by t_i^{skills} and an effort t_i^{effort} expressed in person-month (PM). The tasks must be performed according to a Task Precedence Graph (TPG). This indicates which tasks must be completed before a new task is begun. The TPG is an acyclic directed graph $G(V, A)$ with a vertex set $V = \{t_1, t_2, \ldots, t_T\}$ and an arc set A, where $(t_i, t_j) \in A$ if task t_i must be completed, with no other intervening tasks, before task t_j can start. In order to continue with our example we show in Fig. 2 all the tasks of the software project in hand.

For each task we provide information on the effort in person-month and the set of required skills. For example, task t_1, which consists in creating the UML diagrams of the project in order to be used later by the employees in the following tasks, requires UML expertise (skill s_4) and five person-month. In the same figure we show the TPG of the project, drawing an arrow from task t_i to task t_j if the former must be completed before the latter starts. For example, after the UML diagrams of the application are completed (t_1), both the design of the web page templates for the documentation of the application (t_4) and the database

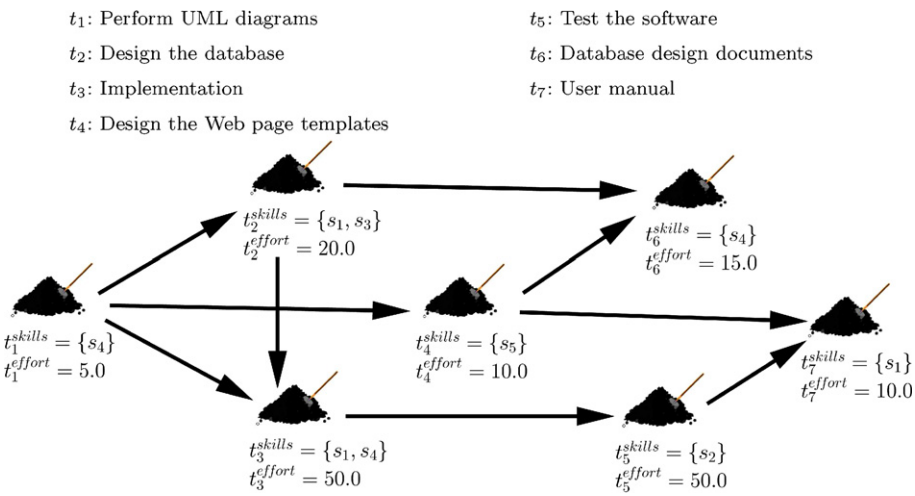

Fig. 2. Task precedence graph of the bank application.

design (t_2) can be started. However, these two tasks must be completed before the database design documentation is produced (t_6).

Our objectives are to minimize the cost and the duration of the project. The constraints are that each task must be performed by at least one person, the set of required skills of a task must be included in the union of the skills of the employees performing the task, and no employee must exceed her/his maximum dedication to the project. The first constraint is necessary in order to complete the project: the project will not be complete if even one task is left undone. The third constraint is obvious after the definition of maximum dedication. However, more could be said regarding the second constraint and therefore we will deal with it below.

At this point we can talk about the number of skills involved in a project. This number can be viewed as a measure of the degree of specialization of the abilities involved in the project. That is, the more skills, the more portions the abilities required to perform the whole software project must be divided into. In our example we could further break down some of the skills. For instance, we can divide the programming expertise into three skills: Java expertise, C/C++ expertise, and Visual Basic expertise. On the other hand, the number of skills can be viewed as a measure of the amount of abilities needed to carry out a project. One example could be developing software for controlling an airplane (large variety of skills needed) versus our bank application. Thus, in our model, the number of skills involved in a project has a dual interpretation in the real world: the degree of specialization of the abilities involved versus the amount of abilities needed to carry out the project. The correct interpretation depends on the specific project. From the project manager point of view, the skills assigned to each task and employee depends on the division of the abilities required for the project at hand. For example, we can do a very fine division of the abilities if our employees are very specialized (they are experts in very specific domains). In such a situation we have a lot of very specific skills involved in the project. Each task can require many of these skills and the employees may have a few skills each. In a different scenario, if our employees have some knowledge of several topics then we will have a few skills associated with vast domains. In this case, the number of skills required by the tasks is smaller than in the previous scenario.

Once we know the elements of a problem instance, we can proceed to describe the elements of a solution to the problem. A solution can be represented with a matrix $\mathbf{X} = (x_{ij})$ of size $E \times T$ where $x_{ij} \geqslant 0$. The element x_{ij} is the degree of dedication of employee e_i to task t_j. If employee e_i performs task t_j with a 0.5 dedication degree s/he spends half of her/his working day on the task. If an employee does not perform a task s/he will have a dedication degree of 0 for that task. This information is used to compute the duration of each task and, indeed, the starting and finishing time of each one, i.e., the time schedule of the tasks (Gantt diagram). From this schedule we can compute the duration of the project (see Fig. 3). The cost can be calculated after the duration of the tasks has been established, taking into account the dedication and the salary of the employees.

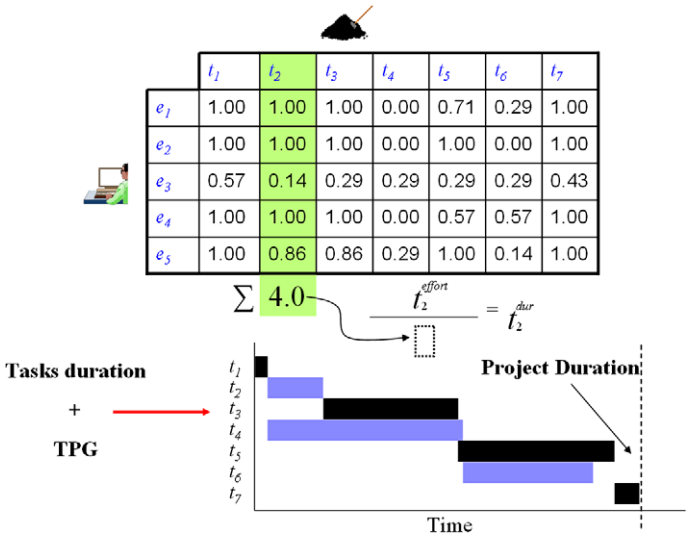

Fig. 3. A tentative solution for the previous example. Using the task durations and the TPG, the Gantt diagram of the project can be computed.

Finally, the overwork of each employee can be calculated using the time schedule of the tasks and the dedication matrix **X**.

In order to evaluate the quality of a given project management solution, we take three issues into account: project duration, project cost, and solution feasibility. To compute the project duration, denoted with p_{dur}, we need to calculate the duration of each individual task (t_j^{dur}). This is calculated in the following way:

$$t_j^{\text{dur}} = \frac{t_j^{\text{effort}}}{\sum_{i=1}^{E} x_{ij}} \tag{1}$$

The next step is to compute the starting and finishing times for each task (t_j^{start} and t_j^{end}). At the same time (thus allowing our algorithm to have a reduced computational cost), the algorithm also calculates the project duration, which will be the maximum finishing time ever found.

The project cost p_{cost} is the sum of the fees paid to the employees for their dedication to the project. These costs are computed by multiplying the salary of the employee by the time spent on the project. The time spent on the project is the sum of the dedication multiplied by the duration of each task. In summary:

$$p_{\text{cost}} = \sum_{i=1}^{E} \sum_{j=1}^{T} e_i^{\text{salary}} \cdot x_{ij} \cdot t_j^{\text{dur}} \tag{2}$$

Now, we detail how the constraints are checked. In order to find whether a solution is feasible, we must first check that all tasks will be performed by somebody, i.e., no task is left undone. That is:

$$\sum_{i=1}^{E} x_{ij} > 0 \quad \forall j \in \{1, 2, \ldots, T\} \tag{3}$$

The second constraint of a feasible solution is that the employees performing one task must have the skills required by that task:

$$t_j^{\text{skills}} \subseteq \bigcup_{\{i | x_{ij} > 0\}} e_i^{\text{skills}} \quad \forall j \in \{1, 2, \ldots, T\} \tag{4}$$

Now, we can discuss the meaning of this constraint. Observe that, if a task requires a skill, the constraint demands that at least one person, not necessarily all of them, have that skill. This makes sense in some situations, for example when the skill is the capacity to lead a group of people and the task requires a single leader to be appointed. Hence, it is possible that one employee working on a task may have none of the skills specifically required, or indeed no skills. In this way, we can model scenarios where some employees do not have the skills required of the task at hand, but they are in contact with and can therefore learn from other employees who have these skills. However, in some scenarios we need all the people working on a task to have a required skill. For example, coming back to our bank application we can require that all the employees implementing the application (t_3) be expert programmers. To tackle this scenario we can allocate a dedication degree of zero on the task to all the employees without the required skill. In our particular case we can set $x_{i3} = 0.0$ for all employees e_i without the skill s_1, that is, e_2, e_3, e_5. This means that the elements of the solution matrix with a zero value imposed are not considered when the optimization algorithm is applied, reducing thereby the number of problem variables. However, when the solution is evaluated a zero value is inserted in the corresponding positions of the matrix.

According to the second constraint, the tasks requiring a skill which no employee has cannot be performed and the project cannot be finished. When this happens all the solutions proposed for the scheduling problem are unfeasible because they violate the second constraint. The project manager can solve this problem in several ways. Firstly, s/he can hire one or several new employees with the required skills. We can model this situation in our formulation of the PSP by enlarging the set of employees with the new ones. Furthermore, if the new employees are hired only to perform the task with the skill demanded we can set the degree of dedication of the new employees to zero for all the other tasks. A second solution to the problem consists in training some of the employees in order to obtain the required skills. In our model this solution is performed by adding new skills to the employees trained.

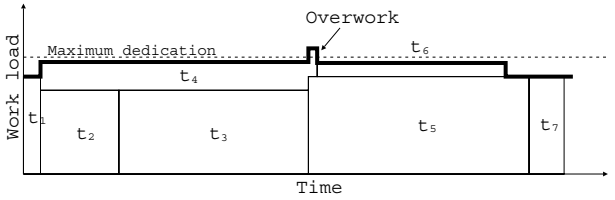

Fig. 4. Working function of employee e_5 in our example.

Finally, in order to compute the overwork p_{over} we need the starting and finishing times for each task, previously computed. For each employee e_i we define her/his working function as

$$e_i^{\text{work}}(t) = \sum_{\{j | t_j^{\text{start}} \leqslant t \leqslant t_j^{\text{end}}\}} x_{ij} \tag{5}$$

If $e_i^{\text{work}}(t) > e_i^{\text{maxded}}$ employee e_i exceeds her/his maximum dedication at instant t. The overwork of employee e_i^{over} is

$$e_i^{\text{over}} = \int_{t=0}^{t=p_{\text{dur}}} \text{ramp}(e_i^{\text{work}}(t) - e_i^{\text{maxded}})\,\mathrm{d}t \tag{6}$$

where ramp is the function defined by

$$\text{ramp}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{7}$$

In Fig. 4 we illustrate the working function of employee e_5 in our example. We have included the tasks that s/he performs at any time. The bold line is the function $e_i^{\text{work}}(t)$ and the dashed line indicates the maximum dedication of the employee (1.2). When the working function passes above the maximum dedication there is overwork. The total overwork of the project is the sum of the overwork for all the employees, i.e.

$$p_{\text{over}} = \sum_{i=1}^{E} e_i^{\text{over}} \tag{8}$$

3. Genetic algorithms

In this article we use a GA to solve the PSP, and thus a discussion of this kind of metaheuristic is appropriate in order to make this work self contained. Genetic Algorithms (GAs) are stochastic search methods that have been successfully applied in many search, optimization, and machine learning problems in the past [1]. Unlike other optimization techniques, GAs maintain a population of encoded tentative solutions that are competitively manipulated by applying some variation operators to find a global optimum. To achieve this goal the problem variables are encoded (binary or floating point, for example) into what are called the *chromosomes*, which are merged and manipulated by the genetic operators to improve their associated quality (called the *fitness*). Thus, one individual is composed of one chromosome and its associated fitness, and the set of individuals forms the population used by the algorithm. Population-based algorithms contrast with trajectory-based ones (like simulated annealing) in that they search from multiple points at the same time, thus reducing the probability of getting stuck in local optima; in addition, they can offer multiple optima to the same problem, an interesting feature that the researchers can use to have an assorted set of solutions to the problems at hand.

After creating an initial set of solutions (randomly or by using a seeding algorithm) GAs normally apply a crossover operation to combine the contents of two parents forming a new solution. This will be modified later by the mutation operation which alters some of the contents of the individual. Not all the individuals participate in the reproduction, only the fittest ones (elitism is very common) are selected from the population by a

```
t:=0;
P(0):={a_1(0), ..., a_μ(0)} ∈ I^μ;                    // initialize
P(0): {Φ(a_1(0)), ..., Φ(a_μ(0))};                    // evaluate
while not ι(P(t)) do                                  // Reproductive loop
        P'(t):=s_Θs(P(t));                            // select
        P''(t):=⊗_Θc(P'(t));                          // recombine
        P'''(t):=m_Θm(P''(t));                        // mutate
        P'''(t): {Φ(a'''_1(t)), ..., Φ(a'''_λ(t))}; // evaluate
        P(t+1):=r_Θr(P'''(t) ∪ Q);                    // replace
        t:=t+1;
endwhile;
```

Fig. 5. Pseudocode of a genetic algorithm.

selection operator such as binary tournament (each parent is selected as the best of two randomly taken individuals). The operators are applied in a stochastic way, thus each one has an associated probability of application in the iterative loop (each step is called a *generation*). Usually, the best individuals in the present and the newly created generation are combined in order that the best ones can be retained for use in the next step of the algorithm (elitist replacement).

The outline of a general GA is presented in Fig. 5. It begins by randomly creating a population $P(t=0)$ of μ solutions (individuals), each one encoding the p problem variables, usually as a vector over $\mathscr{B}=\{0,1\}$ ($I=\mathscr{B}^{p\cdot l_x}$) or $\mathscr{R}$ ($I=\mathscr{R}^p$). An evaluation function Φ is used to associate a quality real value to every solution. The stopping criterion ι of the reproductive loop is to fulfill some condition such as reaching a number of generations or finding a solution. The final solution is identified as the best solution found.

Metaheuristics and, in particular, GAs are not as intensively applied in the software engineering domain as they are in fields like engineering, mathematics, economics, telecommunications or bioinformatics [1,13]. However, the work of Clarke et al. [5] is a good reference for solving software engineering problems with metaheuristics. They identify three areas where the metaheuristics have been successfully applied: software testing, module clustering, and cost estimation. In software testing the approach adopted in the literature is the generation of test data with metaheuristics in order to detect faults in the software execution [14,16] or to find out the worst case execution time of a code fragment [27]. For module clustering, the metaheuristic algorithms are used to get a partition of the system components into clusters with high cohesion among components in the same cluster and a loose coupling among different clusters [8]. Finally, in the cost estimation problem the goal is to estimate the effort needed to carry out a software project [3]. Clarke et al. point out other software engineering domains where metaheuristics could be applied: definition of requirements, system integration, maintenance, and re-engineering using program transformation. In fact, some applications of GAs exist concerning the software engineering experimentation [9], software integration [23], and software release planning [28].

4. Representation and fitness function

In this section we discuss the solution representation and the fitness function used in the genetic algorithm. As we stated in Section 2, a solution to the problem is a matrix $\mathbf{X}$ whose elements x_{ij} are non-negative. Here we have to decide how these elements are encoded. In this article we consider that no employee works overtime, so the maximum dedication of all the employees is 1. For this reason, the maximum value for x_{ij} is 1 and therefore $x_{ij}\in[0,1]$. On the other hand, we use a GA with binary string chromosomes to represent problem solutions. Hence we need to discretize the interval $[0,1]$ in order to encode the dedication degree x_{ij}. We distinguish eight values in this interval which are equally distributed. Therefore, three bits are required for representing them. The matrix $\mathbf{X}$ is stored into the chromosome $\vec{x}$ in row major order.[1] The chromosome length is $E\cdot T\cdot 3$. Fig. 6 shows the representation used.

[1] We use $\vec{x}$ to refer to the chromosome (binary string) which represents the matrix solution $\mathbf{X}$.

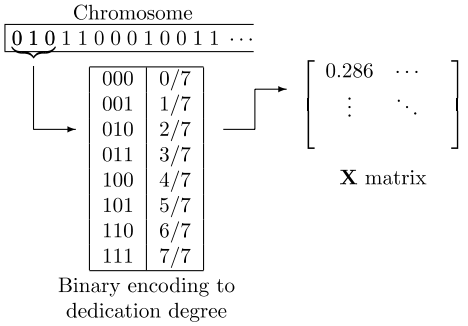

Fig. 6. Representation of a solution in the genetic algorithm.

To compute the fitness of a chromosome $\vec{x}$ we use the next expression

$$f(\vec{x}) = \begin{cases} 1/q & \text{if the solution is feasible} \\ 1/(q+p) & \text{otherwise} \end{cases} \tag{9}$$

where

$$q = w_{\text{cost}} \cdot p_{\text{cost}} + w_{\text{dur}} \cdot p_{\text{dur}} \tag{10}$$

and

$$p = w_{\text{penal}} + w_{\text{undt}} \cdot \text{undt} + w_{\text{reqsk}} \cdot \text{reqsk} + w_{\text{over}} \cdot p_{\text{over}} \tag{11}$$

The fitness function has two terms: the cost of the solution (q) and the penalty for unfeasible solutions (p). The two terms appear in the denominator because the goal is to minimize them, i.e., maximize $f(\vec{x})$. The first term is the weighted sum of the project cost and duration. In this term, w_{cost} and w_{dur} are values weighting the relative importance of the two objectives. These weights allow the fitness to be adapted according to our needs as project managers. For example, if the cost of the project is a primary concern, the corresponding weight must be high. However, we must take into account the order of magnitude of both the project cost and duration. This can be done by setting all the weights to one initially and then executing the GA several times. Next, the cost weight is divided by the average project cost and the duration weight is divided by the average project duration. In this way, the weighted terms related to project cost and duration are in the same order of magnitude. At this point, the project manager can try different weight values in order to adapt the solutions proposed by the GA to her/his requirements.

The penalty term p is the weighted sum of the parameters of the solution that make it unfeasible, that is: the overwork of the project (p_{over}), the number of tasks with no employee associated (undt), and the number of skills still required in order to perform all project tasks (reqsk). Each of these parameters is weighted and added to the penalty constant w_{penal}. This constant is included in order to separate the fitness range of the feasible solutions from that of the unfeasible ones. The weights related to the penalties must be increased until a great number of feasible solutions is obtained. The values for the weights used in this work are shown in Table 1. They have been obtained by exploring several solutions and with the aim of maintaining all the terms of the sum within the same order of magnitude.

Table 1
Weights of the fitness function

Weight	Value
w_{cost}	10^{-6}
w_{dur}	0.1
w_{penal}	100
w_{undt}	10
w_{reqsk}	10
w_{over}	0.1

5. Instance generator

In order to perform a meaningful study we must analyze several instances of the scheduling problem instead of focusing on only one, which could bias the conclusions. To do this we have developed an instance generator which creates fictitious software projects after setting a set of parameters such as the number of tasks, the number of employees, etc. An instance generator is an easily parameterizable tool which derives instances with growing difficulty at will. Also, using a problem generator removes the possibility of hand-tuning algorithms to a particular problem, therefore allowing greater fairness when comparing algorithms. With a problem generator the algorithms can be evaluated on a high number of random problem instances, because a different instance can be solved each time the algorithm is run. Consequently, the predictive power of the results for the problem class as a whole is increased. In this section we describe the instance generator in detail.

The components of an instance are: employees, tasks, skills, and the task precedence graph (TPG). Each of these components has several parameters which must be determined by the instance generator. There are two kinds of values to be generated: single numeric values and sets. For the numeric values a probability distribution is given by the user and the values are generated by sampling this distribution. In the case of sets, the user provides a probability distribution for the cardinality (a numeric value) and then, the elements of the set are randomly chosen from its superset.

All the probability distributions are specified in a configuration file. This file is a plain text file containing attribute-value pairs. We can see a sample file in Fig. 7. Each parameter of the instance has a key name in the configuration file. These key names are included in Table 2. The value of a key name is the name of the probability distribution sampled to generate the value of the parameter. The probability distributions have param-

```
# Configuration File for the Instance Generator

task.number = UniformInt
task.number.parameter.minvalue = 30
task.number.parameter.maxvalue = 30

task.cost = Round
task.cost.parameter.distribution = Normal
task.cost.parameter.distribution.parameter.mu = 10
task.cost.parameter.distribution.parameter.sigma = 5

task.skill = UniformInt
task.skill.parameter.minvalue = 2
task.skill.parameter.maxvalue = 3

graph.e-v-rate = Normal
graph.e-v-rate.parameter.mu = 1.5
graph.e-v-rate.parameter.sigma = 0.5

employee.number = UniformInt
employee.number.parameter.minvalue = 15
employee.number.parameter.maxvalue = 15

employee.salary = Normal
employee.salary.parameter.mu = 10000
employee.salary.parameter.sigma = 1000

employee.skill = UniformInt
employee.skill.parameter.minvalue = 6
employee.skill.parameter.maxvalue = 7

skill.number = UniformInt
skill.number.parameter.minvalue = 10
skill.number.parameter.maxvalue = 10
```

Fig. 7. A sample configuration file for the instance generator.

Table 2
Key names of the configuration file and their associated parameter

Key name	Parameter
task.number	Number of tasks
task.cost	Effort of the tasks
task.skill	Number of the required skills of the tasks
employee.number	Number of employees
employee.salary	Salary of the employees
employee.skill	Number of skills of the employee
graph.e-v-rate	Ratio *edges/vertices* of the TPG
skill.number	Cardinality of the skills set

eters that are specified with additional key-value pairs of the form: ⟨key-name⟩.parameter.⟨param⟩ = ⟨value⟩. For example, the property *employee.skill* in the sample file of Fig. 7 indicates that employees have either 6 or 7 of the 10 possible skills (property *skill.number*).

The instance generator reads the configuration file and then it generates the skills, the tasks, the TPG, and the employees, in that order. For each task, it generates the effort value and the required skill set. For each employee it generates the salary and the set of skills. The pseudocode of the instance generator is shown in Fig. 8.

S = sample (skill.number);
$SK = \{1,\ldots,S\}$;
T = sample (task.number);
for $i = 1$ **to** T **do**
  t_i^{effort} = sample (task.cost);
  $t_i^{skills} = \emptyset$;
  $card$ = sample (task.skill);
  for $j = 1$ **to** $card$ **do**
    s = random (SK/t_i^{skills});
    $t_i^{skills} = t_i^{skills} \cup \{s\}$;
  end;
end;
$evrate$ = sample (graph.e-v-rate);
$A = \emptyset$;
for $i = 1$ **to** $evrate \cdot T$ **do**
  $edge$ = random edge (t_a, t_b) with $a < b$ and not in A;
  $A = A \cup \{edge\}$;
end;
E = sample (employee.number);
for $i = 1$ **to** E **do**
  e_i^{salary} = sample (employee.salary);
  $e_i^{skills} = \emptyset$;
  $card$ = sample (employee.skill);
  for $j = 1$ **to** $card$ **do**
    s = random (SK/e_i^{skills});
    $e_i^{skills} = e_i^{skills} \cup \{s\}$;
  end;
end;

Fig. 8. Pseudocode of the instance generator.

The numeric values of an instance are: the number of tasks, the effort of the tasks, the number of employees, the salary of the employees, and the number of skills. The sets of an instance are: the required skills of the tasks, the skills of the employees, and the set of edges of the TPG graph. For the set of edges we do not specify a distribution for the cardinality directly, but rather for the ratio *edges/vertices*, that is, the generated numeric value is multiplied by the number of tasks in order to get the number of edges of the TPG. The maximum degree of dedication of the employees is not part of the instance itself, but a part of the optimization problem. This parameter can be different for each employee and it is established in the solver configuration file. For this reason the values for this parameter are not generated. A deeper description of the generator, and the program itself can be found at URL http://tracer.lcc.uma.es/problems/psp.

In this work, we use the instance generator to study instances with different parameterizations, that is, different number of tasks, employees, and skills. The difficulty of the instances depends on these parameters. For example, we expect the instances with a larger number of tasks to be more difficult than those with a smaller set of tasks, as in real world projects. This is common sense since it is difficult to do more work with the same number of empdoyees (without working overtime). Following this reasoning, when we increase the number of employees while maintaining the number of tasks we expect easier instances to emerge from the generator. However, these rules of thumb are hard to find in complex projects like ours, because there are interdependencies of some other parameters which have an influence on the difficulty of an instance. One of these parameters is the TPG: with the same number of tasks, one project can be tackled by fewer employees in the same time as another project with a different TPG.

On the other hand, if we compare instances with the same number of tasks we expect that, as the number of employees decreases, the project will last longer. However, with an increment in the number of employees we identify two opposite effects related to the cost: with more people working, operational costs rise; but at the same time the project duration and the expenditure are reduced. Hence, we cannot conclude anything about the project cost directly from the number of employees.

With respect to the number of project skills, we expect instances which have a higher number of demanded skills to be more difficult to solve. With more skills, we have more specialized employees and we expect to need more employees to cover the required skills involved in a task. Hence, the employees work on more tasks and probably some of them may exceed their maximum dedication degree thus making the solution unfeasible. All these features make it very important for the project manager to have an automatic computer tool for taking decisions.

6. Experimental study and results

For the experimental study we generated a total of 48 different instances with the instance generator and solved them with a genetic algorithm. We have grouped the instances into five benchmarks. In the first three groups we change only one parameter of the problem. With these studies we want to analyze how sensitive the results obtained are to the variation of these parameters. In the last two groups we change several parameters at the same time. In this way we study whether the results change in the way suggested by the studies of the first three groups.

To solve the instances, we use a genetic algorithm with a population of 64 individuals, binary tournament selection, 2-D single point crossover, bit-flip mutation, and elitist replacement of the worst (steady-sate genetic

Table 3
Parameters of the GA

GA parameters	
Population	64
Selection	2-tournament (2 inds.)
Recombination	2-D SPX
Mutation	Bit-Flip (1/*length*)
Replacement	Elitist
Stop	5000 steps

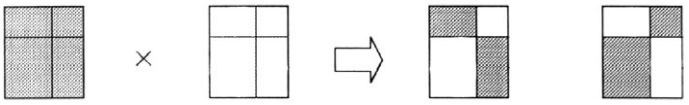

Fig. 9. 2-D single point crossover.

algorithm). The stopping criterion is to reach 5000 steps of the main loop (5064 evaluations). We performed 100 independent runs for each instance. In Table 3 we summarize the GA parameters.

The 2-D single point crossover [26] is an unusual recombination operator applied to matrices. It randomly selects a row and a column (the same in the two parents) and then it swaps the elements in the upper left quadrant and in the lower right quadrant in both individuals (Fig. 9).

In the following subsections we present the studies performed and the results for all the identified benchmarks.

6.1. First benchmark: changing the number of employees

The first step is to study the influence which the number of employees has on the solutions. We use four different instances of the problem with the same software project, i.e., they have the same tasks and the same TPG. The only difference in the instances is the number of employees. The maximum dedication and the salary of the employees is also the same. In addition, the constraint related to the skills is not taken into account. That is, all the employees have the necessary skills to perform any given task. This situation has been modelled by introducing only one skill and providing all the employees with that skill. All the instances are based on the same software project with 10 tasks, thus, the total work to be done is always the same. For this reason we expect the project duration of the solutions proposed by the genetic algorithm to decrease when the number of employees increases. More precisely, the project duration and the number of employees must have an inverse relationship and their product must be constant. In Table 4 we show the results obtained with four different numbers of employees: 5, 10, 15, and 20. For each case we present the hit rate (percentage of runs getting a feasible solution), the average duration of the feasible solutions proposed, and the product of the number of employees and the average project duration in months.

We observe in the results that the hit rate decreases when the number of employees increases, that is, the problem becomes more difficult when we increase the number of employees. One would have imagined that with more employees it would be easier to find a solution for the problem. However, in this situation the third constraint (requiring no overwork) is more difficult to satisfy. At the same time, the search space is larger and this does not help the search process. As we predicted before, the project duration decreases when the number of employees increases. In fact, the product of the number of employees and the average duration is very similar for the different instances (fourth column). However, it increases slightly with the number of employees for the same reason as the hit rate is reduced: the instances are more difficult for the GA. The cost of the software project is exactly the same in all the solutions because all the employees have the same salary, that is, the cost of a one person-month is fixed throughout all the instances.

6.2. Second benchmark: changing the number of tasks

Now we study the influence of the number of tasks on the solutions. We solve three instances where we maintain the employees and we change the software projects. In particular, the three software projects have

Table 4
Results obtained when the number of employees changes

Employees	Hit rate	Avg. duration	Avg. $E \times p_{dur}$
5	87	21.88	109.40
10	65	11.27	112.70
15	49	7.73	115.95
20	51	5.88	117.60

Table 5
Results obtained when the TPG changes

Tasks	Hit rate	Cost	Avg. duration	Avg. p_{cost}/p_{dur}
10	73	980,000	21.84	44,944.33
20	33	2,600,000	58.29	44,748.35
30	0	–	–	–

a different number of tasks: 10, 20, and 30. As in the previous benchmark, all the employees have the same salary and maximum dedication. For this reason all the solutions for the same project have the same cost. Since we use the same probability distribution in order to generate the cost of the project tasks in the three projects, we expect an increase in the project cost with an increase in the number of tasks. In addition, we do not consider the second constraint, so we expect a proportional relationship between the duration and the cost of the projects. Furthermore, if all the employees are working all the time for the project, the ratio between the cost and the duration must be exactly the sum of the salary of the employees. In the instances there are five employees with a monthly salary of \$10,000, so the cost-duration ratio must be nearly \$50,000. We present the results of the three instances in Table 5. We show the hit rate, the project cost in dollars, the average duration in months of the feasible solutions proposed, and the average cost per month of the projects in dollars per month.

From the results we observe that the problem becomes harder when the number of tasks increases. In fact, the genetic algorithm is not able to obtain any feasible solution for the software project with 30 tasks. The reason for this behavior is the same as in the previous benchmark: when the number of tasks increases it is more difficult for the GA to get a solution satisfying the overwork constraint. We also observe that both the cost of the projects (third column) and the project duration (fourth column) increase with the number of tasks. The cost per month of the project (fifth column) is nearly \$50,000 in the two cases as we predicted. This parameter cannot be greater than \$50,000 because this implies a violation of the overwork constraint. When the value of this parameter is near the optimal one (\$50,000 in our case) this means an efficient allocation of employees to tasks. We conclude from the results that the allocation gained for the 10 tasks instance is more efficient than that obtained for the 20 tasks one. We can explain this result with the increase in the search space when shifting from 10 to 20 tasks.

6.3. Third benchmark: changing the employee expertise

In this section we study how the number of skills per employee, i.e., the expertise of the employees, influences the results. We solve five instances with the same software project and the same number of employees. The employees all have the same monthly salary and the same maximum dedication. Instances differ from each other in the employee skills. We analyze five different values for the number of skills of the employees: 2, 4, 6, 8, and 10. The employee skills are randomly selected from the set of 10 project skills. All the tasks require five different skills. We present the hit rate, the average duration of the projects, and the average cost per month in Table 6.

We observe that the problem is harder to solve with a lower number of skills per employee, that is, if the expertise of the employees is low, it is more difficult to allocate them to the tasks without violating the skills constraint (the second one). We can also notice that the average project duration obtained in the different

Table 6
Results obtained when the number of skills per employee changes

Skills	Hit rate	Avg. duration	Avg. p_{cost}/p_{dur}
2	39	21.71	45,230.15
4	53	21.77	45,068.64
6	77	21.98	44,651.28
8	66	22.00	44,617.02
10	75	22.11	44,426.90

instances remains almost constant with a slight increase for higher values of the employee expertise. This means that the GA is able to allocate the employees to the tasks in a more efficient way when the level of employee expertise is lower. The reason is that the feasible region of the search space is enlarged when the employees have more skills, and therefore the average quality of the solutions included in the feasible region decreases.

6.4. Fourth benchmark: expertise specialization fixed

We include in this group 18 different problem instances generated by the instance generator. The instances include different software projects and they differ from each other in all the previously studied parameters. In particular, we assign different values to the number of employees, the number of tasks, and the number of employee skills. The number of different skills of the project is set to 10 in all the instances. The number of employees can be 5, 10, or 15 and the number of tasks 10, 20, or 30. Two ranges of values are considered separately for the number of skills of the employees: from 4 to 5, and from 6 to 7. As in the previous benchmarks, the maximum dedication for all the employees is 1.0 (full working day). We show in Table 7 the hit rate for all the instances (from 100 independent runs).

From these results we can conclude that the instances with a larger number of tasks are more difficult to solve than those with a smaller set of tasks, as we concluded in Section 6.2. In the second row of results, we observe an inverse relationship between the number of employees and the difficulty of the problem. This contrasts with the results of the first benchmark (Section 6.1). What is happening? The main difference between the two cases resides in the skills. In the first benchmark the skills constraint was not considered whereas in this case it is. When the number of employees increases, it is more difficult to fulfill the overwork constraint but it is easier to fulfill the skills constraint because the staff is highly skilled. These two trends are in conflict with each other, but in this case the second one seems to be predominant.

In order to better illustrate the meaning of these results we plot the solutions obtained in a graph showing their cost versus their duration (Figs. 10 and 11). Cost and duration are clear tradeoff criteria in any project. This is the kind of graph that a manager would like to see before taking any decision on the project. We have put a label ⟨tasks⟩-⟨employees⟩ near the solutions of the same instance.

In the figures, the solutions of the instances can be seen as point clusters. Their elongated form depends on the scale of the axis (chosen to maintain the solutions of all the instances in the same graph), however we can appreciate a slight inclination of the clusters showing the tradeoff mentioned between cost and duration: when the cost of a solution is lower, its duration is longer.

As we expected, when the number of employees decreases for a given number of tasks, the project lasts longer. This observation is maintained despite each point cluster representing a different instance with different TPG. In the figures, we can observe that a larger number of employees does not necessarily mean a more expensive project in all the cases. However, we cannot obtain any fundamental conclusion on this fact because the instances belong to very different software projects.

6.5. Fifth benchmark: employees expertise fixed

In this final benchmark composed of 18 instances we study the influence of the number of different skills on a project. This will shed some light on existing large companies where an assorted set of persons of varied

Table 7
Hit rate for the fourth benchmark

	4–5 Skills			6–7 Skills		
	Employees			Employees		
Tasks	5	10	15	5	10	15
10	94	97	97	84	100	97
20	0	6	43	0	76	0
30	0	0	0	0	0	0

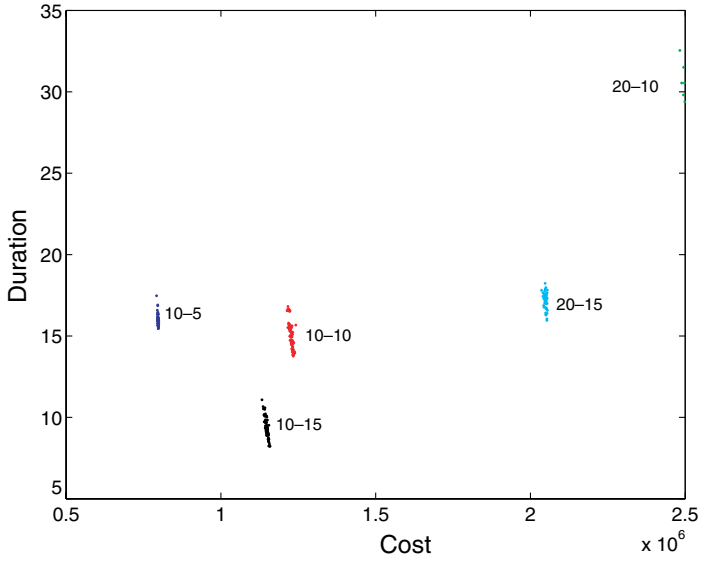

Fig. 10. Results with 4–5 skills per employee. Labels show the number of tasks and employees of the instance.

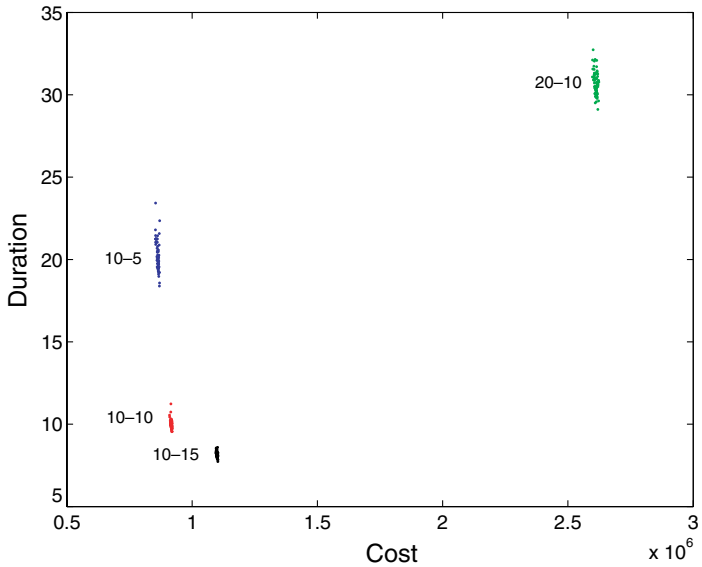

Fig. 11. Results with 6–7 skills per employee. Labels show the number of tasks and employees of the instance.

experience are to be optimally assigned to software projects. In this case we fix the range of the number of skills per task and employee from 2 to 3. The number of tasks can be 10, 20, or 30 and the number of employees takes values 5, 10, and 15 as in the previous benchmark. The number of different skills is either 5 or 10. In Table 8 we show the results.

As in the previous subsection we can see that an increment in the number of tasks means an increment in the difficulty of the problem. The participation of more employees usually implies a decrement in the difficulty of the instance (it is easier to manage the project). However, we can now conclude one additional fact: we confirm, as expected, that a larger number of demanded skills makes the instance more difficult (in general) to solve.

Table 8
Hit rate for the fifth benchmark

	5 Skills			10 Skills		
	Employees			Employees		
Tasks	5	10	15	5	10	15
10	98	99	100	61	85	85
20	6	9	12	8	1	6
30	0	0	0	0	0	0

From Figs. 12 and 13 we conclude that the cost of the project increases with the number of tasks, and the duration of the project decreases with the increment in the number of employees. This was also observed in the previous benchmarks. However, with more employees, the overall cost of the project is reduced in all cases, a fact that was not observed before (only similar to 10–15 and 20–15 in Fig. 10). Previously we argued that different instances use different projects and for this reason we cannot obtain any definitive conclusion. Here, we are in the same situation but analyzing the particular solutions of the instances we observe that with a larger number of employees all of them work on all the tasks at a low degree of dedication. In this way, the tasks are performed more quickly and the global cost of the project is lower.

6.6. Further understanding of the dynamics of our algorithm

In order to end our presentation of results we plot the average best fitness evolution of some instances in the 100 runs. Our goal is to present a trace of the search performed by the GA. In Fig. 14 on the left we can see the evolution of the instances with 10 tasks and 5 skills: the final average best fitness increases with the number of employees. With a larger number of employees the algorithm can compute a more efficient scheduling reducing the duration and/or the cost of the project, which in turn increases the fitness value of the solutions. This trend can also be observed in Fig. 14 on the right for the 10-tasks/10-skills instances.

In Fig. 15 we plot the evolution of the average best fitness for the instances with 10 tasks, 10 skills, and 4–5 and 6–7 skills per employee. In this case the relationship between the fitness and the number of employees is

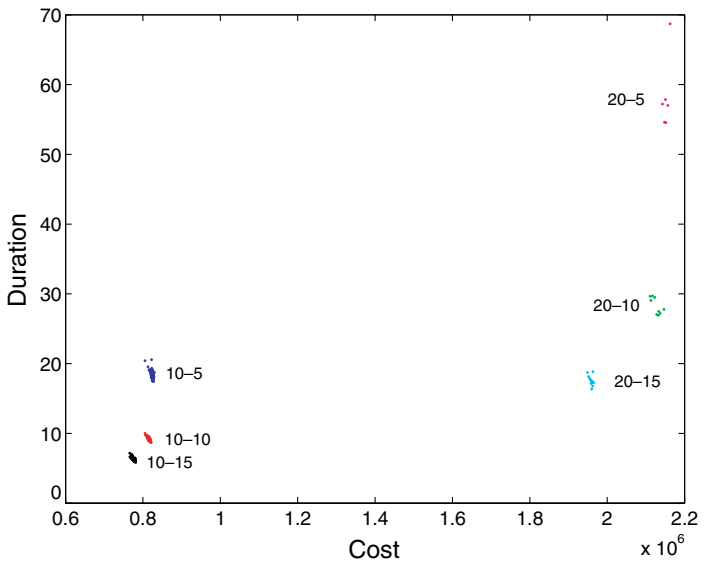

Fig. 12. Results with five skills. Labels show the number of tasks and employees of the instance.

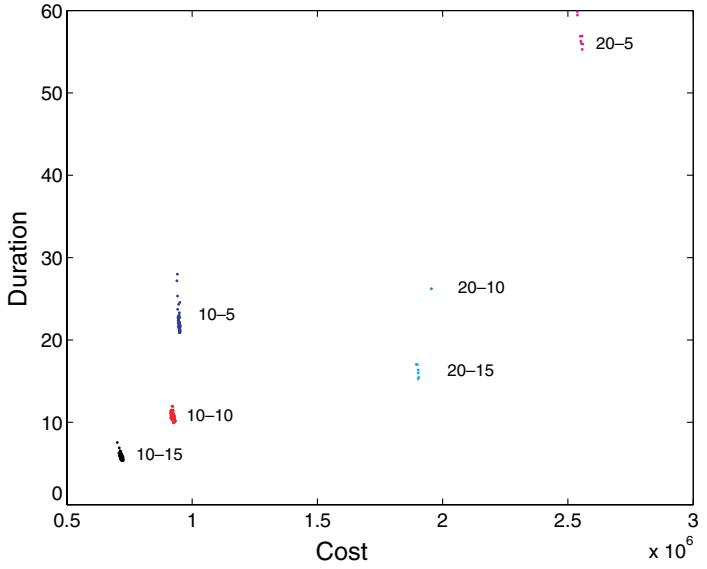

Fig. 13. Results with 10 skills. Labels show the number of tasks and employees of the instance.

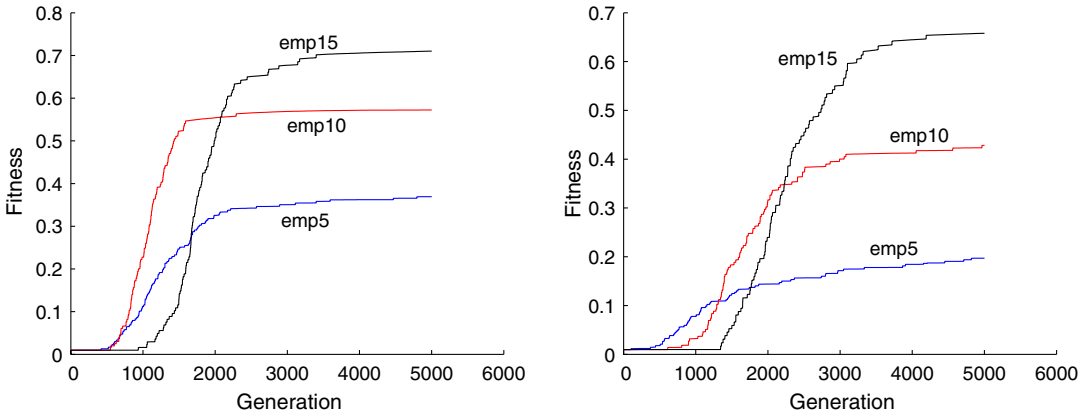

Fig. 14. Average best fitness evolution of the instances with 10-tasks/5-skills (left) and 10-tasks/10-skills (right). The label emp*i* identifies the instance with *i* employees.

not so clear. However, we can notice that for the instances with 10 and 15 employees the number of skills per employee significantly affects the best attained fitness: with 6–7 skills per employee the best fitness is higher than with 4–5; i.e., a varied and larger set of skills can be profited from if an automatic tool such as ours is used in project management. This is in accordance with the idea that more qualified people do the work better. However, this trend was not observed with five employees, meaning that even efficient people need a group of help in real world projects.

The two final plots (Fig. 16) show the evolution in the instances with five skills and 20 and 30 tasks. Note in the right plot that the instances have a quasi-logarithmic evolution with a very low fitness. The algorithm fails to find a feasible solution for these instances and all the individuals are then penalized, thus keeping their fitness values below 0.01.

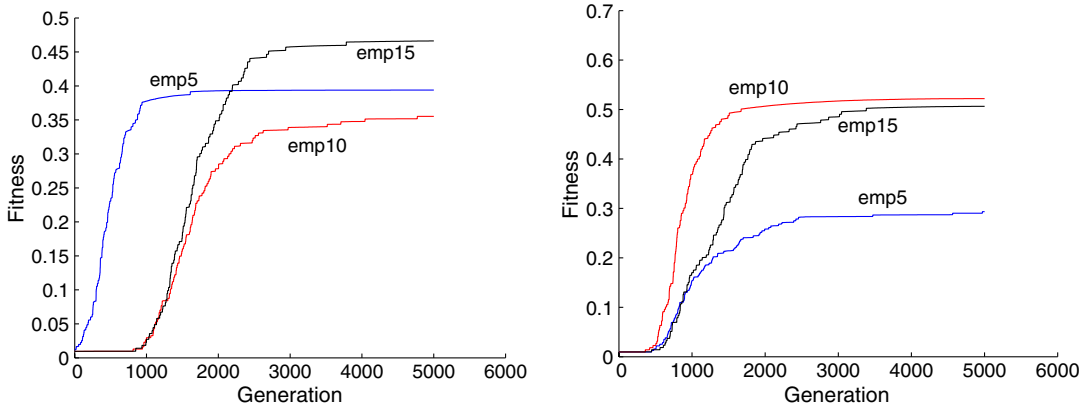

Fig. 15. Average best fitness evolution of the instances with 10-tasks/4-5-skills per employee (left) and 10-tasks/6-7-skills per employee (right). The label emp*i* identifies the instance with *i* employees.

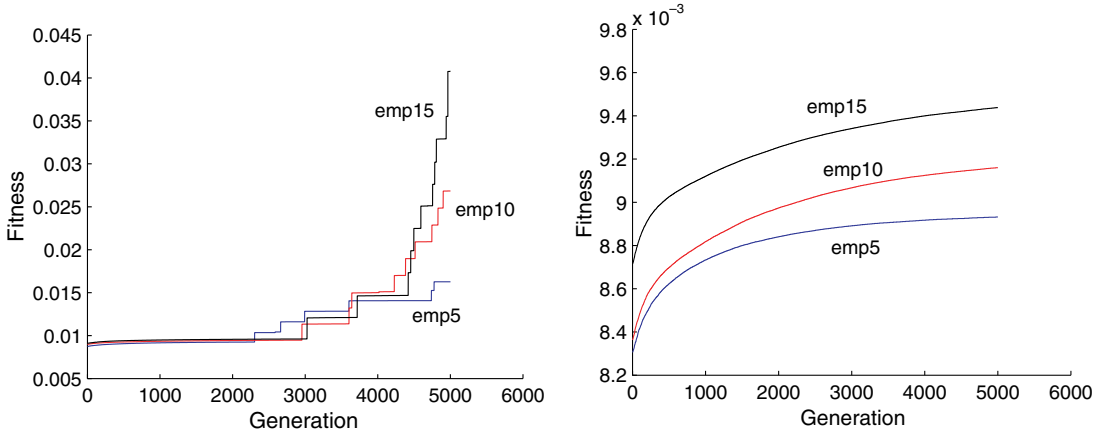

Fig. 16. Average best fitness evolution of the instances with 20-tasks/5-skills (left) and 30-tasks/5-skills (right). The label emp*i* identifies the instance with *i* employees.

7. Conclusions

In this work we have tackled the general Project Scheduling Problem with genetic algorithms. This problem is essential for the software engineering industry nowadays and automatically finding "good" solutions to it can save software companies lots of time and money. A software manager can study different scenarios with such an automatic tool in order to take appropriate decisions on the best project for her/his company. Furthermore, in our approach, s/he can adjust the fitness weights to better represent particular real world projects. The Project Scheduling is a combinatorial optimization problem and an exhaustive search can take too much time to get a solution. Here, as in some other work [4], the utility of metaheuristic techniques for the problem is clearly stated. Our contribution to the software engineering management is an automated tool based on genetic algorithms that can be used to assign people to the project tasks in a near optimal way, trying different configurations concerning the relative importance of the cost and duration of the project. Although the project model is very simple it can serve as a first step in the application of evolutionary algorithms to the *in silico* experiments in software engineering.

We have used a genetic algorithm, and have performed an in depth analysis with an instance generator. We have solved 48 different project scenarios and performed 100 independent runs for each test in order to get

statistically meaningful solutions. The results show that the instances with more tasks are more difficult to solve and their solutions are more expensive. In the same way, the projects with a larger number of employees are easier to tackle and can be led to a successful end in a shorter time. However, the relationship between employees and cost is not so simple: in some cases it is direct and in other cases it is inverse.

In the future we plan to add new instances with additional aspects to study the influence of the instance parameters on the difficulty, such as the complexity of dealing with a large team or the overhead of assigning a large set of tasks to an employee. Also, we will solve the problem with other metaheuristic algorithms. In particular, we can directly apply multiobjective metaheuristic algorithms to optimize the two objectives tackled in the work (duration and cost of the projects). In addition, we plan to apply our algorithms to real world data in order to illustrate how to use the techniques in a real software project. Finally, we will extend the model to face real world problems from industry, once we know which are the best techniques to apply (the goal of this initial study).

Acknowledgements

This work has been partially funded by the Ministry of Science and Technology (MCYT) and Regional Development European Found (FEDER) under contract TIN2005-08818-C04-01 (the OPLINK project). Francisco Chicano is supported by a grant (BOJA 68/2003) from the Junta de Andalucía (Spain). We also thank to Guillerme Horta Travassos for easing the access to several publications and to an anonymous review for her/his constructive comments.

Appendix A. Average best fitness evolution plots

In this appendix we include the evolution of the average best fitness in the instances of the last two benchmarks. We decided to include this appendix to offer an in depth view of our results that could be interesting for

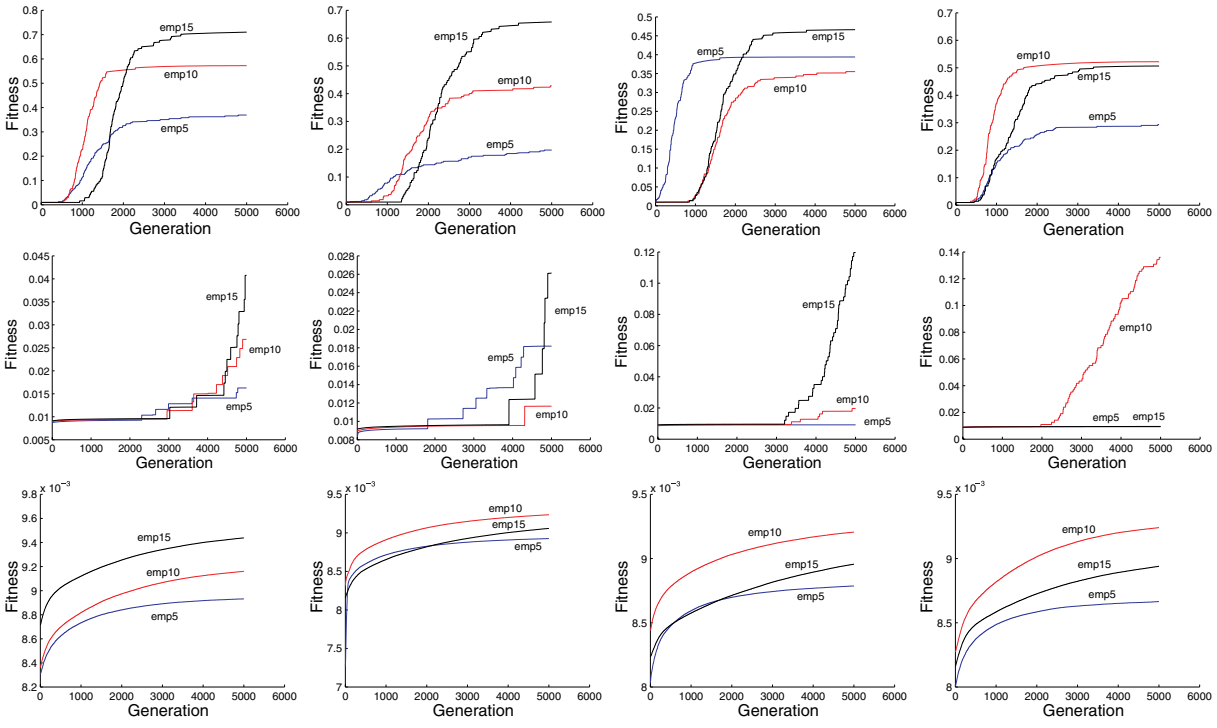

Fig. A.1. Tasks and skills fixed (horizontal: 5, 10, 10/4–5, 10/6–7 skills, vertical: 10, 20, 30 tasks).

only some readers. We group related instances in the same graph in order to compare the traces. When doing so, the question is how to group the instances. To clarify the presentation we decided to group the instances according to three different criteria.

The first criterion consists of maintaining in the same graph all the instances which have the same number of project skills, skills per employee, and the same number of tasks. As we have four possible configurations of project skills and three different tasks we have obtained 12 graphs in this way, shown in Fig. A.1. In this figure we find all the graphs shown in Section 6. Let us observe the smooth curves of the third row, all of them belonging to the 30 tasks instances where the GA does not obtain any feasible solution. This contrasts with the "noisy" curves of the central row (20 tasks instances) for which the GA does indeed enter into the feasible region of solutions (always after 2000 steps approx.). The main conclusion that we draw from these graphs is that the final best fitness value generally increases with the number of employees.

The second grouping is made by plotting together in the same graph the instances which have the same number of employees and the same configuration of skills (Fig. A.2). Again we have 12 graphs with three traces per graph (number of tasks). The first observation is that only the curves of the 10-tasks instances always get a higher fitness value than the feasible solution fitness value (0.01). The point at which the curve starts rising depends on the number of employees. With a larger number of employees the rising is delayed, perhaps due to the larger size of the chromosome. In some graphs (like the one of the 5-employees/10-skills instance) we see a modest rising of the 20 tasks curves.

Finally, the third criterion is to group the instances which have the same number of tasks and employees, thus obtaining the nine graphs of Fig. A.3. In the first column (10 tasks instances) we can see that the final best fitness of the 5-skills instances is higher than in the case of the 10-skills instances. This was already discussed in Section 6 when we observed that projects involving 10 skills were more difficult to solve than those requiring five skills. On the other hand, the point at which the curves start a deep ascent is delayed with the increment in the number of employees (this was also observed in Fig. A.2). The second column helps us to conclude that a larger number of employees makes the search easier.

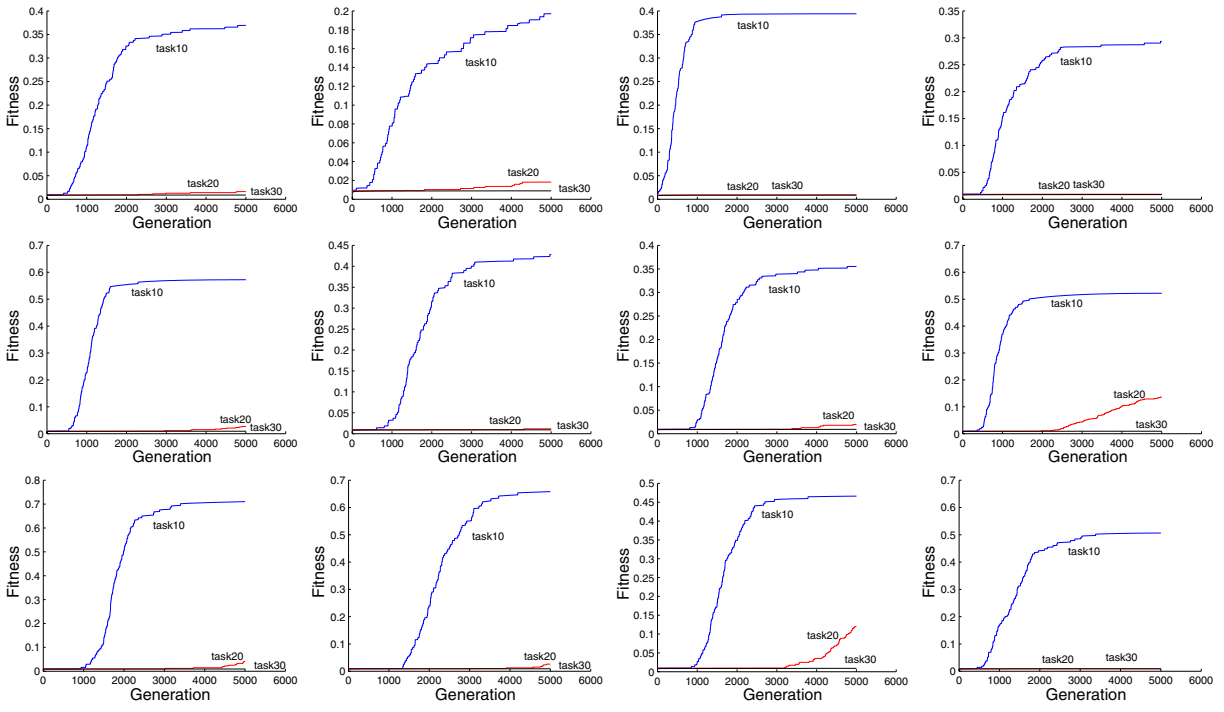

Fig. A.2. Employees and skills fixed (horizontal: 5, 10, 10/4–5, 10/6–7 skills, vertical: 5, 10, 15 employees).

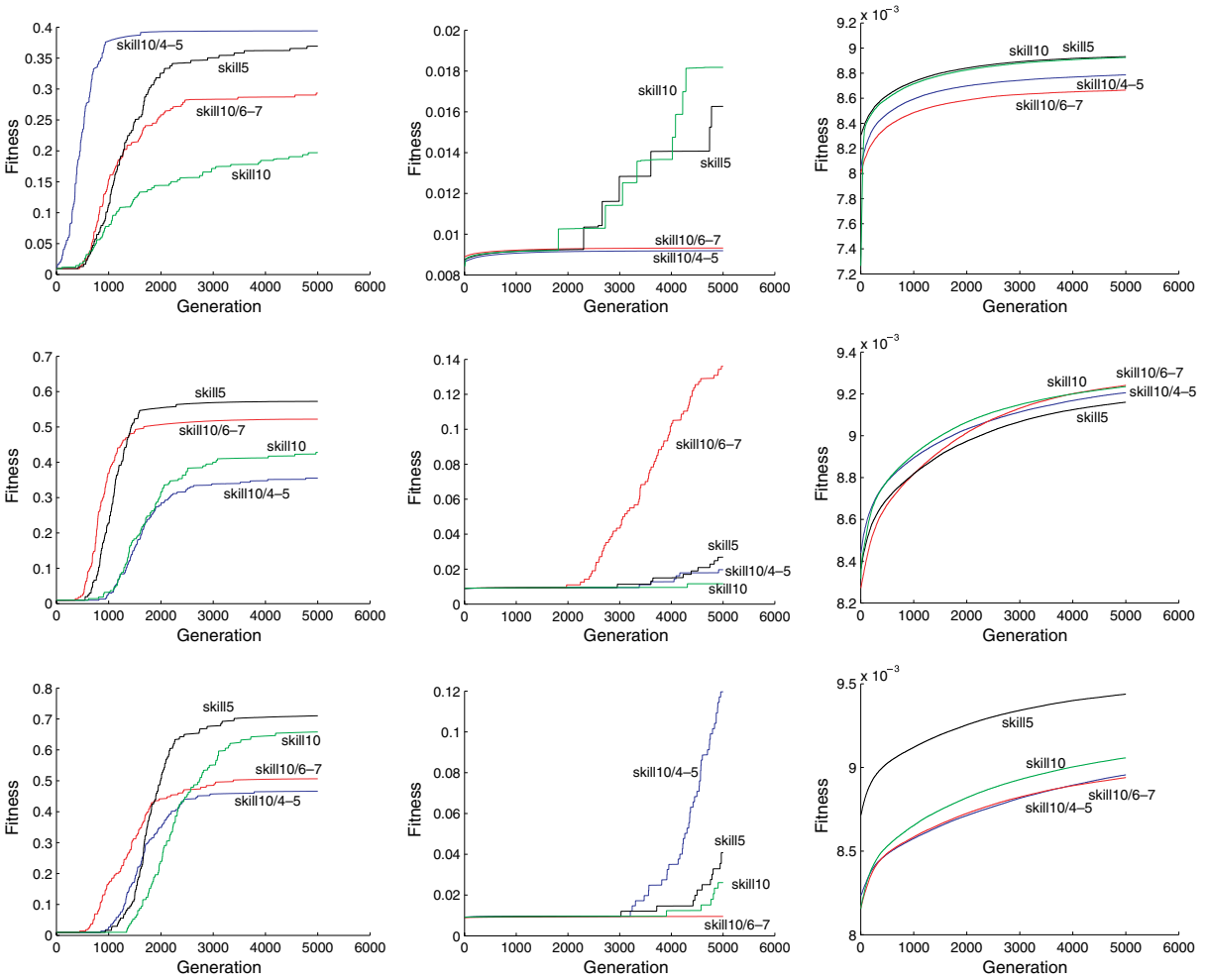

Fig. A.3. Employees and tasks fixed (horizontal: 10, 20, 30 tasks, vertical: 5, 10, 15 employees).

References

[1] T. Bäck, D.B. Fogel, Z. Michalewicz, Handbook of Evolutionary Computation, Oxford University Press, New York, USA, 1997.
[2] B. Boehm, R. Ross, Theory-w software project management: principles and examples, IEEE Transaction on Software Engineering 15 (7) (1989) 902–916.
[3] C. Burgess, M. Lefley, Can genetic programming improve software effort estimation? a comparative evaluation, Information and Software Technology 43 (14) (2001) 863–873.
[4] C.K. Chang, M.J. Christensen, T. Zhang, Genetic algorithms for project management, Annals of Software Engineering 11 (2001) 107–139.
[5] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proc. Software 150 (3) (2003) 161–175.
[6] E. Demeulemeester, W. Herroelen, A branch-and-bound procedure for the multiple resource-constrained project scheduling problem, Management Science 38 (1992) 1803–1818.
[7] T. Dohi, Y. Nishio, S. Osaki, Optimal software release scheduling based on artificial neural networks, Annals of Software Engineering 8 (1999) 167–185.
[8] D. Doval, S. Mancordis, B. Mitchell, Automatic clustering of software systems using a genetic algorithm, in: Proceedings of the International Conference on Software Technology and Engineering Practice, IEEE Computer Society, Washington, DC, USA, 1999, pp. 73–81.
[9] R. García, M. Oliveira, J. Maldonado, Genetic algorithms to support software engineering experimentation, in: Proceedings of the International Symposium on Empirical Software Engineering, IEEE Computer Society, Noosa Heads, Australia, 2005, pp. 488–497.

[10] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, Massachusetts, USA, 1989.
[11] S. Henninger, Case-based knowledge management tools for software development, Automated Software Engineering 4 (1997) 319–340.
[12] K.S. Hindi, H. Yang, K. Fleszar, An evolutionary algorithm for resource-constrained project scheduling, IEEE Transactions on Evolutionary Computation 6 (5) (2002) 512–518.
[13] F. Jiménez, J.M. Cadenas, J.L. Verdegay, G. Sánchez, Solving fuzzy optimization problems by evolutionary algorithms, Information Sciences 152 (2003) 303–311.
[14] B.F. Jones, H.-H. Sthamer, D.E. Eyres, Automatic structural testing using genetic algorithms, Software Engineering Journal 11 (5) (1996) 299–306.
[15] H.-M. Lee, S.-Y. Lee, T.-Y. Lee, J.-J. Chen, A new algorithm for applying fuzzy set theory to evaluate the rate of aggregative risk in software development, Information Sciences 153 (2003) 177–197.
[16] J.-C. Lin, P.-L. Yeh, Automatic test data generation for path testing using GAs, Information Sciences 131 (1–4) (2001) 47–64.
[17] L.-C. Liu, E. Horowitz, A formal model for software management, IEEE Transaction on Software Engineering 15 (10) (1989) 1280–1293.
[18] D. Merkle, M. Middendorf, H. Schmeck, Ant colony optimization for resource-constrained project scheduling, IEEE Transactions on Evolutionary Computation 6 (4) (2002) 333–346.
[19] A. Mingozzi, V. Maniezzo, S. Ricciardelli, L. Bianco, An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation, Management Science 44 (5) (1998) 714–729.
[20] M. Palpant, C. Artigues, P. Michelon, LSSPER: solving the resource-constrained project scheduling problem with large neighbourhood search, Annals of Operations Research 131 (2004) 237–257.
[21] C.L. Ramsey, V.R. Basili, An evaluation of expert systems for software engineering management, IEEE Transactions on Software Engineering 15 (6) (1989) 747–759.
[22] M. Ronchetti, G. Succi, W. Pedrycz, B. Russo, Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm, Information Sciences 176 (5) (2006) 475–489.
[23] G. Ruhe, D. Greer, Quantitative studies in software release planning under risk and resource constraints, in: Proceedings of the International Symposium on Empirical Software Engineering, IEEE Computer Society, Roman Castles, Rome, Italy, 2003, pp. 262–270.
[24] B. Talbot, J. Patterson, An efficient integer programming algorithm with network cuts for solving resource-constrained scheduling problems, Management Science 24 (1978) 1163–1174.
[25] G.H. Travassos, M.O. Barros, Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering, in: Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering, Fraunhofer IRB Verlag, Roman Castles, Italy, 2003, pp. 117–130.
[26] B. Wall, A genetic algorithm for resource-constrained scheduling, Ph.D. thesis, Massachusetts Institute of Technology, 1996.
[27] J. Wegener, H. Sthamer, B. Jones, D. Eyres, Testing real-time systems using genetic algorithms, Software Quality Journal 6 (2) (1997) 127–135.
[28] L. Yang, B.F. Jones, S.H. Yang, Genetic algorithm based software integration with minimum software risk, Information and Software Technology 48 (3) (2006) 133–141.