

# 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*

---

## Abstract

A Project Scheduling Problem consists in deciding who does what during the software project lifetime. This is a capital application 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 a project, 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 about the influence the most important attributes of the problem 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.

*Key words:* 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 what resources are used

---

\* Corresponding author. Tel.: +34 95213 3303; fax: +34 95213 1397.

*Email addresses:* [eat@lcc.uma.es](mailto:eat@lcc.uma.es) (Enrique Alba), [chicano@lcc.uma.es](mailto:chicano@lcc.uma.es) (J. Francisco Chicano).

to perform each task and when it should be carried out. The 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 actual project, the manager wants an automatic plan reconciling 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 software management process. We can find expert systems to diagnose problems in the software development [21], neural networks for deciding when to deliver the software to the users [7], genetic algorithms for project scheduling [4], CASE tools for the knowledge management of software development [11], all of them forming 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 factors addressed are related to the time, human skills, budget, and project complexity involved. All of these factors make our study more difficult and mean a step toward their application 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 it [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 a 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 included 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 and which has been solved with both exact techniques [6,19,24] and metaheuristic ones [12,18,20]. However, there are some differences between PSP and RCPS. First, in PSP there is a cost associated with the employees and a project cost which must be minimized (in addition to the project duration). Besides, in RCPS there are several kinds of resources while PSP has one only kind (the

employee) with several possible skills. We must 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 normal 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  $i$  goes from 1 to  $E$  (the number of employees). Let  $SK$  be the set of skills, and  $s_i$  the  $i$ -th skill with  $i$  varying from 1 to  $S = |SK|$ . The skills of the employee  $e_i$  will be denoted with  $e_i^{skills} \subseteq SK$ , the monthly salary with  $e_i^{salary}$ , and the maximum dedication to the project with  $e_i^{maxded}$ . The salary and the maximum dedication are both 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 working day length of the employee. 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.

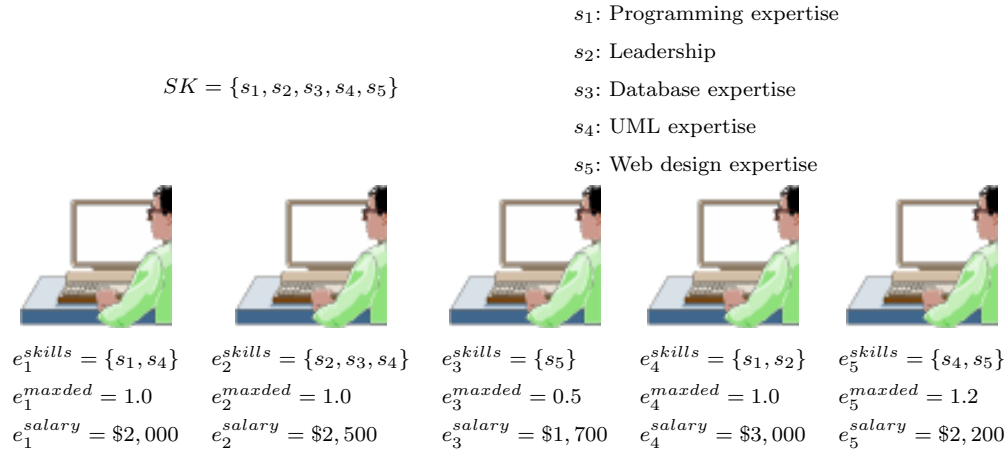


Fig. 1. Possible staff of an example software company.

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, the employee  $e_2$ , who earns \$2,500 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 colleague, the employee  $e_4$ , is also able to lead a group ( $s_2$ ) and, in addition, she is a great programmer ( $s_1$ ). These two employees and the employee  $e_1$  can spend all of their working day developing the application (maximum dedication equal to one) but this doesn't mean that they do it. On the contrary,

the employee  $e_3$  can only dedicate half of her working day to the project. We can find several reasons for this fact: perhaps the employee has a part-time contract, or she has administrative tasks to carry out in the company during part of the day. The employee  $e_5$  can work overtime, her maximum dedication is greater than one ( $e_5^{maxded} = 1.2$ ), and this means that she can work on the bank application up to twenty percent more than in an ordinary working day. In this way, we can model the extra time of the employees, a quite “real world” feature included in the problem definition. However, the project manager must take into account that an overloaded employee can increase her mistake rate and with it the number of errors of the software developed. This leads to a lower quality of the final product and, possibly, to the need to correct or to develop again the wrong parts. In any case, the consequence can be an increased 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 the moment and let us study how the tasks of a software project are modelled. The tasks are denoted with  $t_i$ , where  $i$  goes from 1 to  $T$  (the number of tasks). Each task  $t_i$  has a set of required skills associated with it that we denote with  $t_i^{skills}$  and an effort  $t_i^{effort}$  expressed in person-month (PM). The tasks must be performed according to a Task Precedence Graph (TPG). It 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  $V = \{t_1, t_2, \dots, t_T\}$  and an arc set  $A$ , where  $(t_i, t_j) \in A$  if the 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.

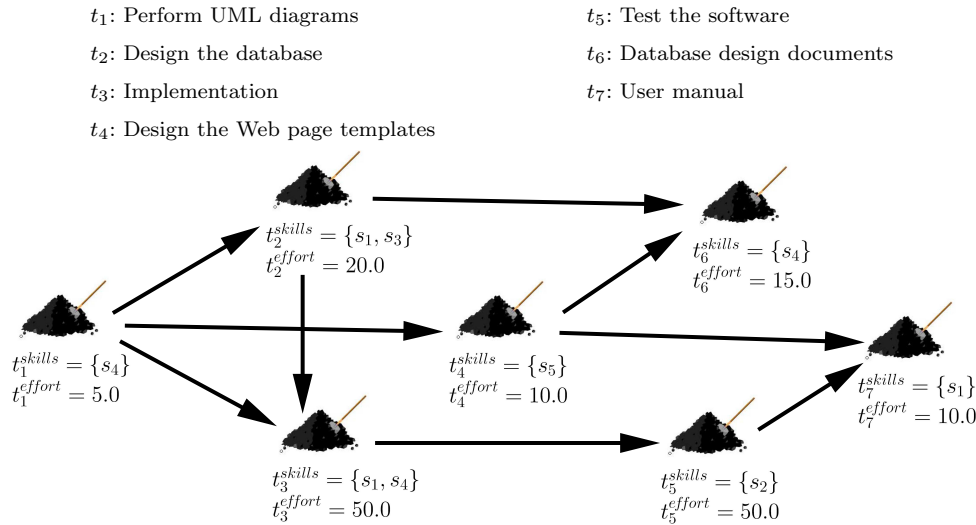


Fig. 2. Task Precedence Graph of the bank application.

For each task we inform of the effort in person-month and the set of required

skills. For example, the task  $t_1$ , which consists in performing the UML diagrams of the project in order to be used 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 ( $t_1$ ) the design of the web page templates for the documentation of the application ( $t_4$ ) and the database 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 maximum dedication to the project. The first constraint is necessary in order to complete the project: if there is even one undone task the project is not complete. The third constraint is obvious after the definition of maximum dedication. However, we could discuss more on the second constraint and we will do so 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, with a larger number of skills the abilities needed to perform the whole software project are divided into a greater number of portions than if it needed a reduced number of skills. 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 perform a project. One example could be developing software for controlling an airplane (large variety of skills needed) versus our bank application. Thus, the number of skills of a project in our model has a dual interpretation in the real-world: the degree of specialization of the abilities involved versus the amount of abilities needed to perform 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 concrete 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 have a few skills each. In the opposite direction, if our employees have some notions on several topics we have a few skills associated with vast domains. 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  $x_{ij} \geq 0$ . The element  $x_{ij}$  is the degree of dedication of the employee  $e_i$  to the task  $t_j$ . If the employee  $e_i$  performs the task  $t_j$  with a 0.5 dedication degree she spends half of her working day on the task. If an employee does not perform a task she will have a dedication degree of 0 for that task. This information helps to compute the duration of each task and, indeed, the start and end 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 taking into account the dedication and the salary of the employees. Finally, the overwork of each employee can be calculated using the time schedule of the tasks and the dedication matrix  $\mathbf{X}$ .

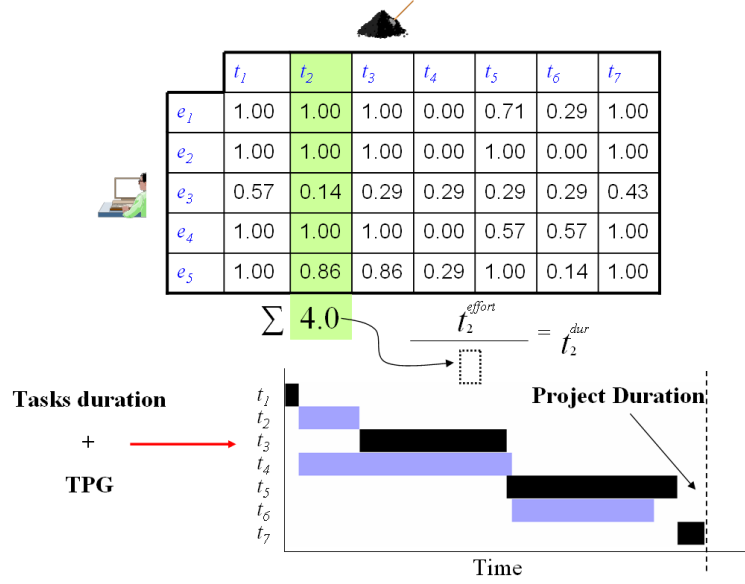


Fig. 3. A tentative solution for the previous example. Using the task durations and the TPG the Gantt diagram of the project can be computed.

In order to evaluate the quality of a given project management solution, we take into account three issues: 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:

$$t_j^{dur} = \frac{t_j^{effort}}{\sum_{i=1}^E x_{ij}} \quad (1)$$

The next step is to compute the starting and ending 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 is the maximum end 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 charges 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:

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

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

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

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

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

Now, we can discuss the meaning of this constraint. Observe that, if a task requires a skill, the constraint demands 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 one only leader to be appointed. Hence, it is possible that one employee working in a task have none of the required skills of the tasks. In particular, the employee can have no skill. In this way, we can model scenarios where some employees do not have the required skills of the task they work on, but they are in contact with other employees which have the skills and they can learn the 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 impose 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 the 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 in this way 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 em-

ployee 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 demanded skill 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 of training some of the employees in order to have the required skills. In our model this solution is performed by adding new skills to the employees trained.

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

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

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

$$e_i^{over} = \int_{t=0}^{t=p_{dur}} ramp(e_i^{work}(t) - e_i^{maxded})dt \quad (6)$$

where  $ramp$  is the function defined by:

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

In Fig. 4 we illustrate the working function of the employee  $e_5$  in our example. We have included the tasks that she performs at any time. The bold line is the function  $e_i^{work}(t)$  and the broken 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.:

$$p_{over} = \sum_{i=1}^E e_i^{over} \quad (8)$$



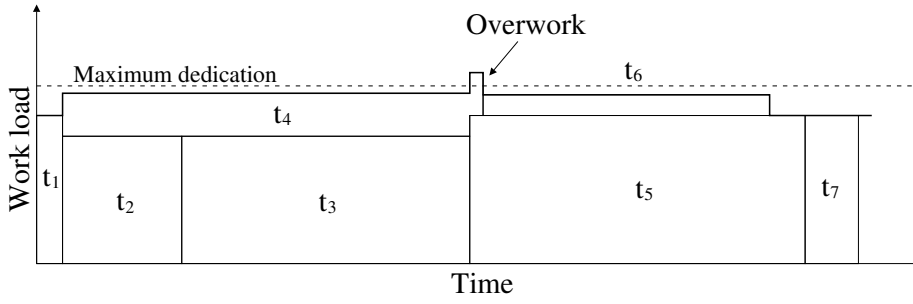


Fig. 4. Working function of the employee  $e_5$  in our example.

### 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 in hand.

After creating an initial set of solutions (in a random way or by using a seeding algorithm) it is usual that GAs apply a crossover operation to recombine the contents of two parents in a new one. 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 selection operator like 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  $\mu$  solutions (individuals), each one encoding the  $p$  problem variables, usually as a vector over  $\mathcal{B} = \{0, 1\}$  ( $I = \mathcal{B}^{p \cdot l_x}$ ) or  $\mathcal{R}$  ( $I = \mathcal{R}^p$ ). An evaluation function  $\Phi$  is used to associate a quality real value to every solution. The stopping criterion  $\iota$  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.

```

t:=0;
P(0):={ $\vec{a}_1(0), \dots, \vec{a}_\mu(0)$ }  $\in I^\mu$ ;           // initialize
P(0): { $\Phi(\vec{a}_1(0)), \dots, \Phi(\vec{a}_\mu(0))$ };         // evaluate
while not  $\iota(P(t))$  do                               // Reproductive loop
     $P'(t) := s_{\Theta_s}(P(t))$ ;                         // select
     $P''(t) := \otimes_{\Theta_c}(P'(t))$ ;                 // recombine
     $P'''(t) := m_{\Theta_m}(P''(t))$ ;                   // mutate
     $P'''(t) : \{\Phi(\vec{a}'_1(t)), \dots, \Phi(\vec{a}'_\lambda(t))\}$ ; // evaluate
     $P(t+1) := r_{\Theta_r}(P'''(t) \cup Q)$ ;             // replace
    t:=t+1;
endwhile;

```

Fig. 5. Pseudocode of a Genetic Algorithm.

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 representation of the solutions in the genetic algorithm and the fitness function employed. As we said in Section 2 a solution for the problem is a matrix  $\mathbf{X}$  whose elements  $x_{ij}$  are non-negative. Here we have to decide how to represent these elements. 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 we have  $x_{ij} \in [0, 1]$ . On the other hand, we use a GA with binary string chromosomes to represent problem solutions and hence we need to discretize the interval  $[0, 1]$  in order to represent the dedication degree  $x_{ij}$ . We distinguish eight values in this interval that are equally distributed, and employ three bits to represent them. The matrix  $\mathbf{X}$  is stored in the chromosome  $\vec{x}$  in row major order<sup>1</sup>. The chromosome length is  $E \cdot T \cdot 3$ . Fig. 6 shows the representation used.

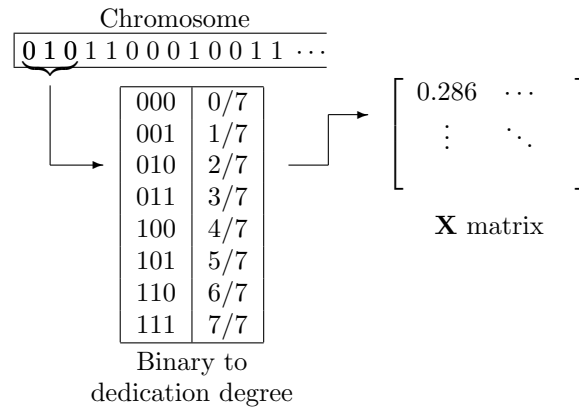


Fig. 6. Representation of a solution in the genetic algorithm.

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

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

where

$$q = w_{cost} \cdot p_{cost} + w_{dur} \cdot p_{dur} \quad (10)$$

and

$$p = w_{penal} + w_{undt} \cdot undt + w_{reqsk} \cdot reqsk + w_{over} \cdot p_{over} \quad (11)$$

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

The fitness function has two terms: the quality 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 quality 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 us to adapt the fitness according to our necessities as project managers. For example, if the cost of the project is a main concern, the corresponding weight must be high. However, we must have into account the order of magnitude of the project cost and duration. This can be done by setting all the weights to one in a first try and executing the GA several times. Then, 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 its necessities.

The penalization 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 the tasks of the project ( $reqsk$ ). Each of these parameters are 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 just 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 task which derives instances with growing difficulty at will. Also, using a problem generator removes the opportunity to hand-tune 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, since a different instance can be solved each time the algorithm runs, then the predictive power of the results for the problem class as a whole is increased. In this section we describe in detail the instance generator.

The components of an instance are: employees, tasks, skills, and the task precedence graph (TPG). Each of these components have 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 that we can see 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 parameters that are specified with additional key-value pairs with the form: `<key-name>.parameter.<param> = <value>`. For example, the property *employee.skill* in the sample file of Fig. 7 indicates that the number of skills of the employees are 6 or 7 from the 10 possible skills (property *skill.number*).

```

# 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 <i>edges/vertices</i> of the TPG
skill.number	Cardinality of the skills set

The instance generator reads the configuration file and then 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);
 $SK = \{1, \dots, S\}$ ;
 $T$  = 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  $evrate * T$  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 for the ratio *edges/vertices*, that is, the generated numeric value is multiplied by the number of tasks 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 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 article, 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 employees (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 few employees in the same time as another project with 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 spans a longer time. However, with an increment in the number of employees we identify two opposing trends influencing the cost: with more people working operational costs rise, but at the same time the project duration is reduced and the expenditure is reduced. Hence, we cannot conclude anything about the project cost directly from the number of employees.

With respect to the number of skills of the project we expect that 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 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 separated the instances into five benchmarks. In the first three groups we change one only parameter of the problem. With these studies we want to study how sensitive 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 or not 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-state genetic algorithm). The stop 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.

Table 3

Parameters of the GA

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

The 2-D single point crossover [26] is an unusual recombination operator applied to tables. 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).



Fig. 9. 2-D Single Point Crossover.

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

### 6.1 First benchmark: variation of 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

ten 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.

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

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. It could seem 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 is increased. In fact, the product of the number of employees and the average duration is very similar for the different instances (forth column). However, it increases slightly with the number of employees for the same reason that 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: variation of 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 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 near \$50,000. In Table 5 we present the results of the three instances. For each case we present 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.

Table 5  
Results obtained when the TPG changes

Tasks	Hit rate	Cost	Avg. Duration	Avg. $p_{cost}/p_{dur}$
10	73	980000	21.84	44944.33
20	33	2600000	58.29	44748.35
30	0	-	-	-

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 is increased it is more difficult for the GA to get a solution satisfying the overwork constraint. We also observe that the cost of the projects (third column) and the project durations (fourth column) increase with the number of tasks. The cost per month of the project (fifth column) is near \$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 ten 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 ten to twenty tasks.

### 6.3 Third benchmark: variation of the employee expertise

In this section we study how the 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. The only difference among the instances concerns 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 ten skills of the project. All the tasks require five different skills. In Table 6 we present the hit rate, the average duration of the projects, and the average cost per month.

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	45230.15
4	53	21.77	45068.64
6	77	21.98	44651.28
8	66	22.00	44617.02
10	75	22.11	44426.90

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 notice that the average project duration obtained in the different 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

In this benchmark we include 18 different problem instances generated with the instance generator. In these instances the software projects are different and we change simultaneously 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 skills of the instances is always 10. 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, or 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

Table 7

Hit rate for the fourth benchmark

Tasks	<i>4-5 skills</i>			<i>6-7 skills</i>		
	Employees			Employees		
	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

tasks are more difficult to solve than those with a smaller set of tasks, as we concluded in Subsection 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 (Subsection 6.1). What is happening? The main difference between the two cases resides in the skills. In the first benchmark we did not consider the constraint of the skills and this time we do. When the number of employees increases it is more difficult to satisfy the overwork constraint but it is easier to satisfy the constraint of the skills because the staff is highly skilled. These two trends 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 of the form `<tasks>-<employees>` near the solutions of the same instance.

In the figures, the solutions of each instance are observed as point swarms. 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 swarms showing the mentioned tradeoff between cost and duration: when the cost of a solution is smaller, its duration is longer.

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

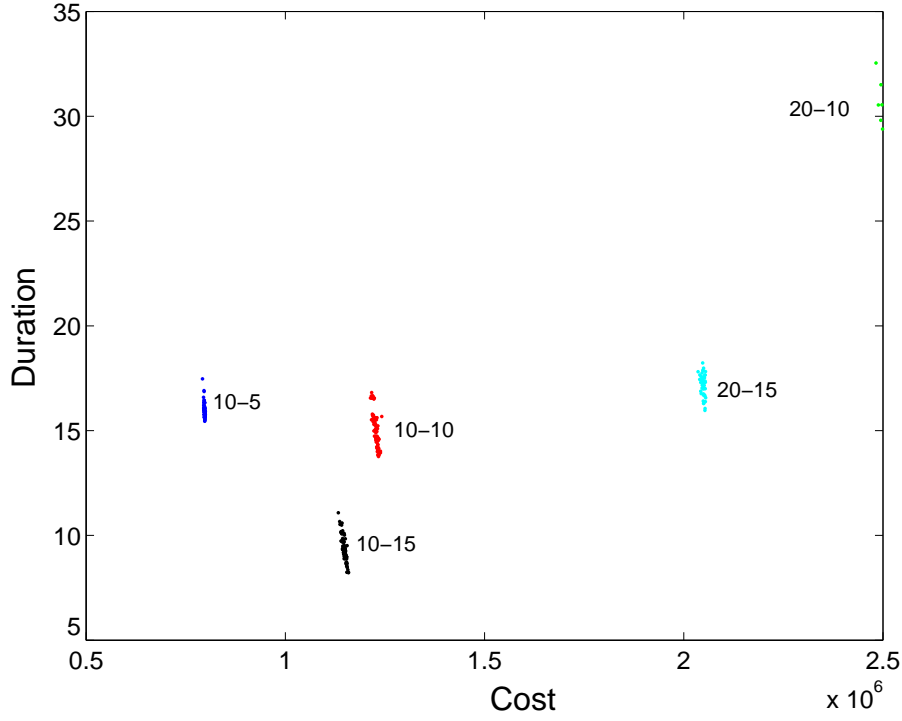


Fig. 10. Results with 4-5 skills per employee. Labels show the number of tasks and employees of the instance.

### 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 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.

Table 8

Hit rate for the fifth benchmark

Tasks	<i>5 skills</i>			<i>10 skills</i>		
	Employees			Employees		
	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

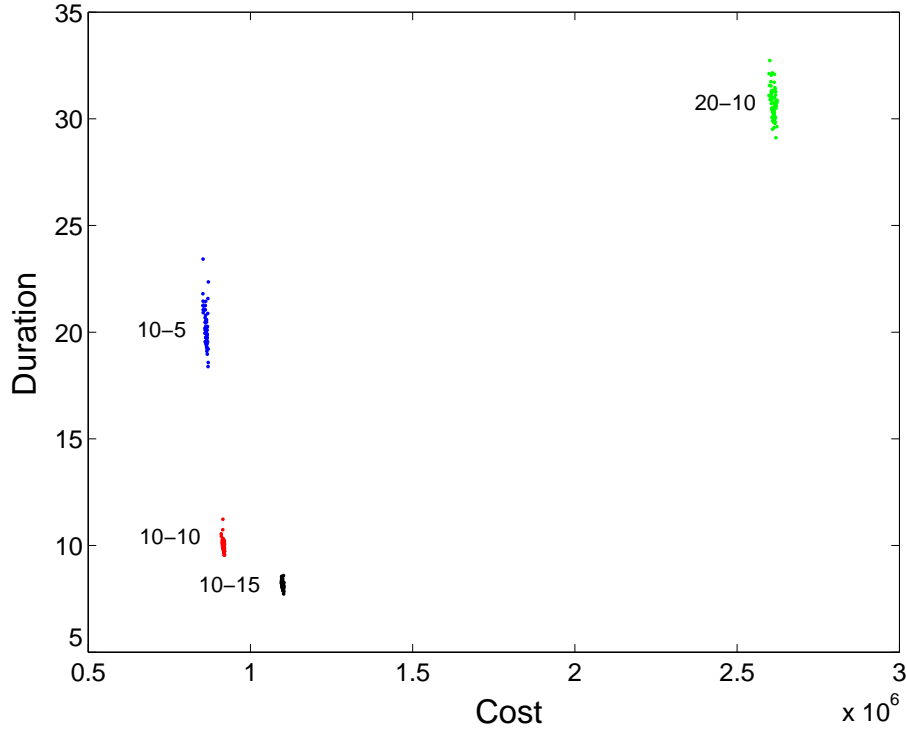


Fig. 11. Results with 6-7 skills per employee. Labels show the number of tasks and employees of the instance.

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.

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 the 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 low.

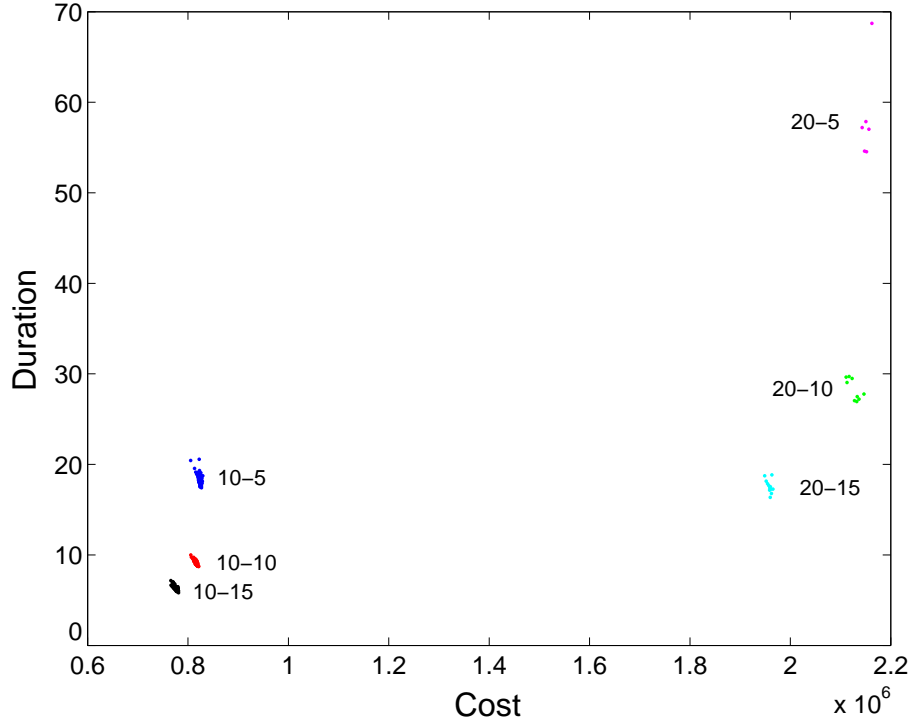


Fig. 12. Results with 5 skills. Labels show the number of tasks and employees of the instance.

#### 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 offer 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 that reduces the duration and/or the cost of the project, which 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 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 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 5 employees, meaning that even efficient people need a work group of help in real world projects.



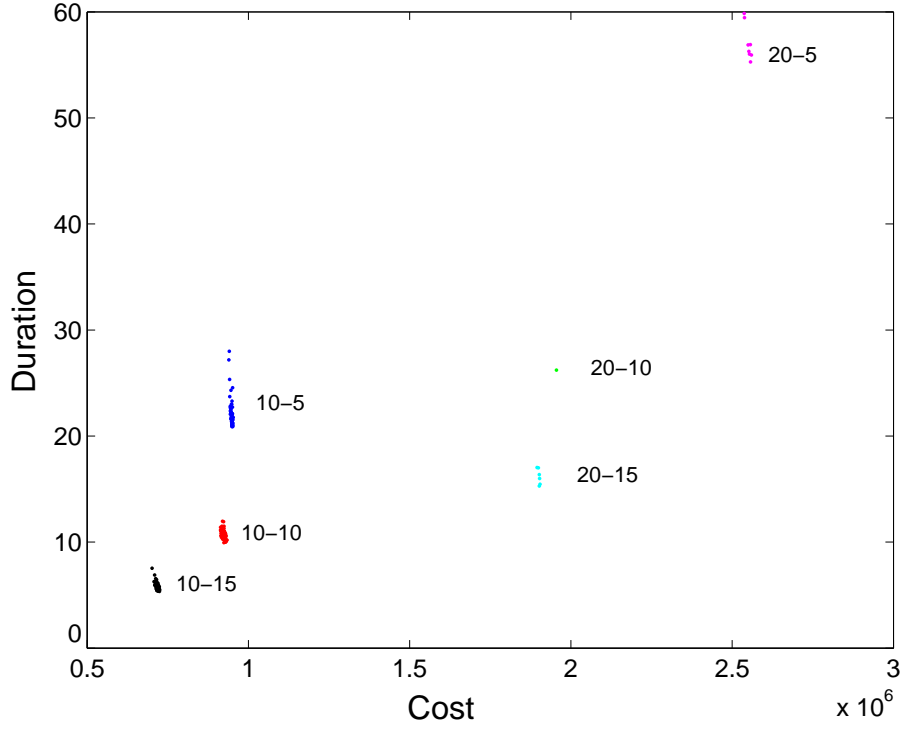


Fig. 13. Results with 10 skills. Labels show the number of tasks and employees of the instance.

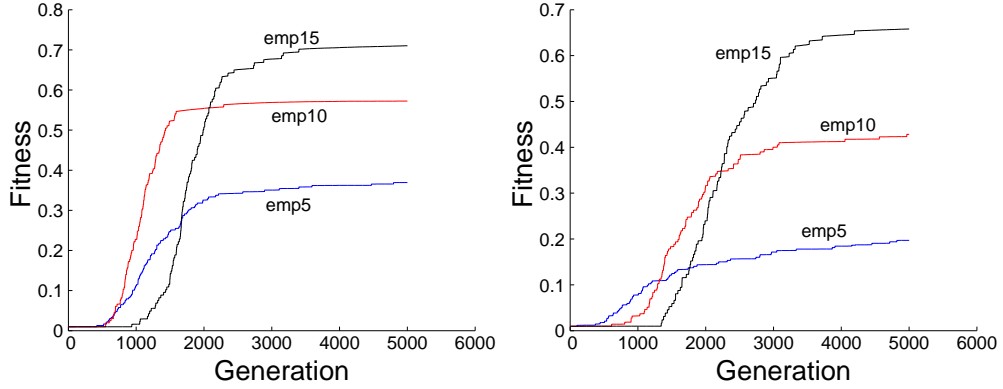


Fig. 14. Average best fitness evolution of the instances with 10-tasks/5-skills (left) and 10-tasks/10-skills (right). The label  $\text{emp}_i$  identifies the instance with  $i$  employees.

The two final plots (Fig. 16) show the evolution in the instances with 5 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 maintaining their fitness values below 0.01.

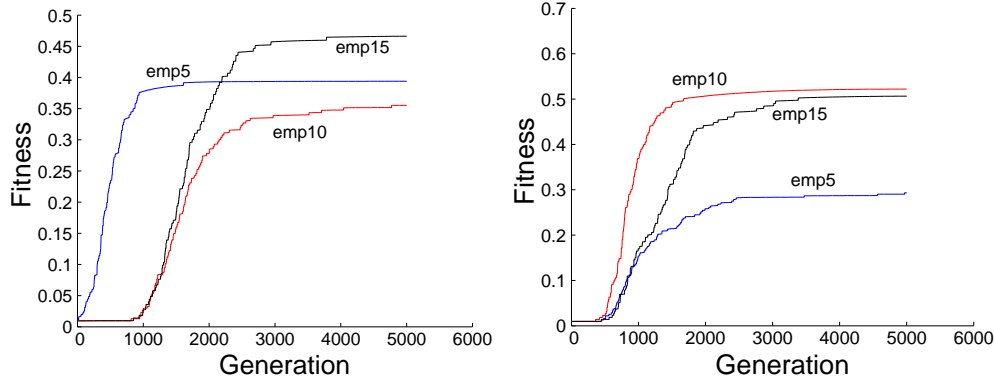


Fig. 15. Average best fitness evolution of the instances with 10-tasks/4-5-skills per employee (left) and 10-tasks/6-7-skills per employee (right). The label  $\text{emp}_i$  identifies the instance with  $i$  employees.

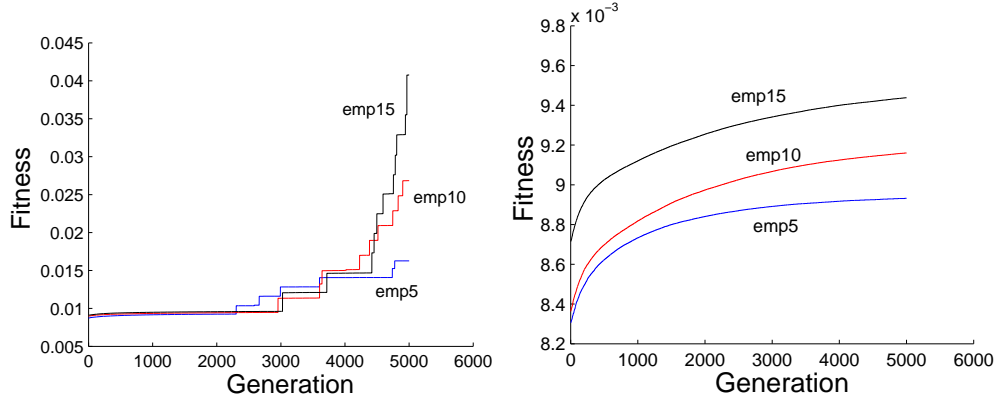


Fig. 16. Average best fitness evolution of the instances with 20-tasks/5-skills (left) and 30-tasks/5-skills (right). The label  $\text{emp}_i$  identifies the instance with  $i$  employees.

## 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 to take 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 nearly 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 solved 48 different project scenarios and performed 100 independent runs for each test 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 driven to a successful end in a shorter time. However, the relationship between employees and cost is not that 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 first study).

## 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 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 get 12 graphs in this way, that are 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 increases in general 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 over 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 above the 10-skills instances one. This was already discussed in Section 6 when we observed that projects involving 10 skills were more difficult to solve than those requiring 5 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.

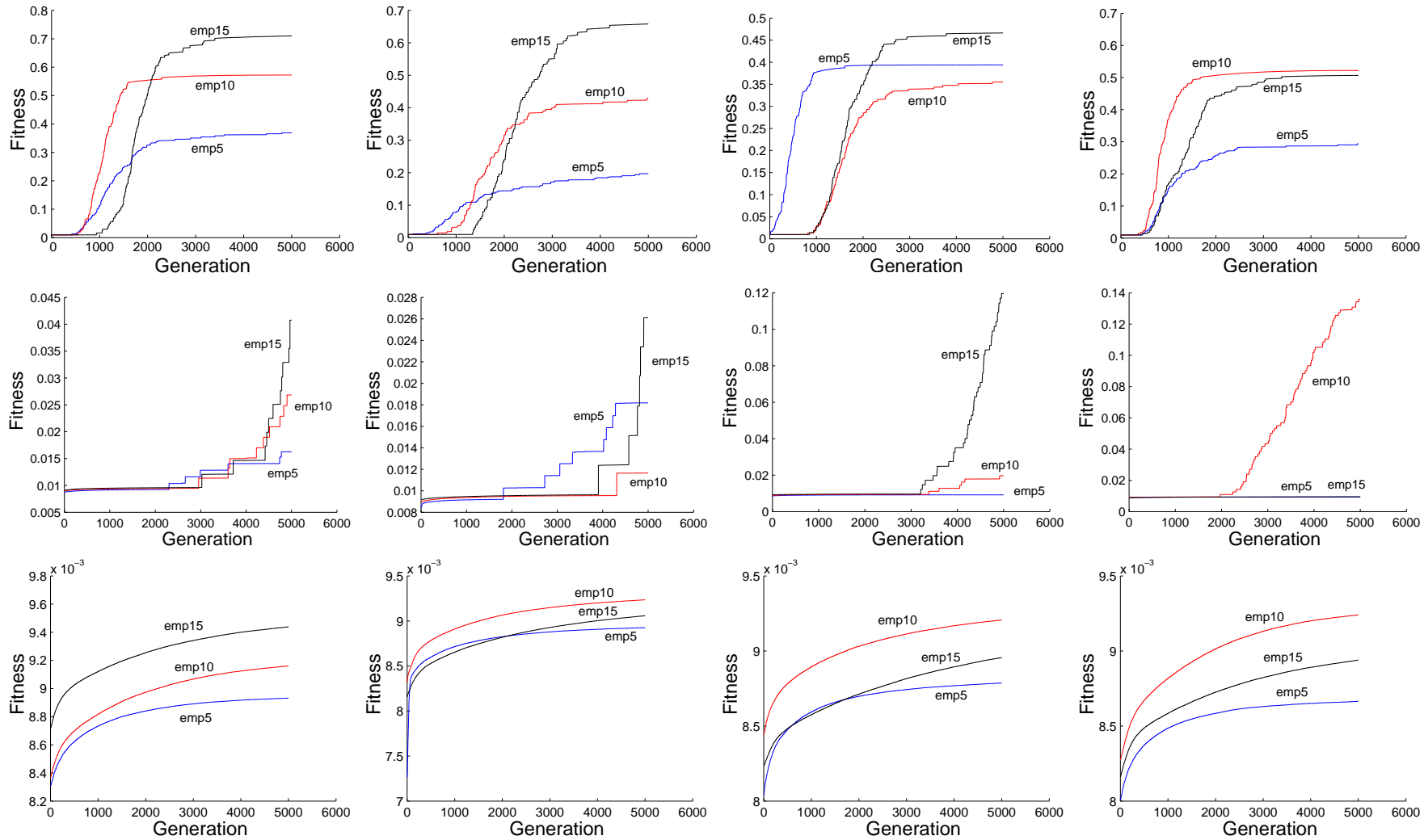


Fig. A.1. Tasks and skills fixed (horizontal: 5, 10, 10/4-5, 10/6-7 skills, vertical: 10, 20, 30 tasks).

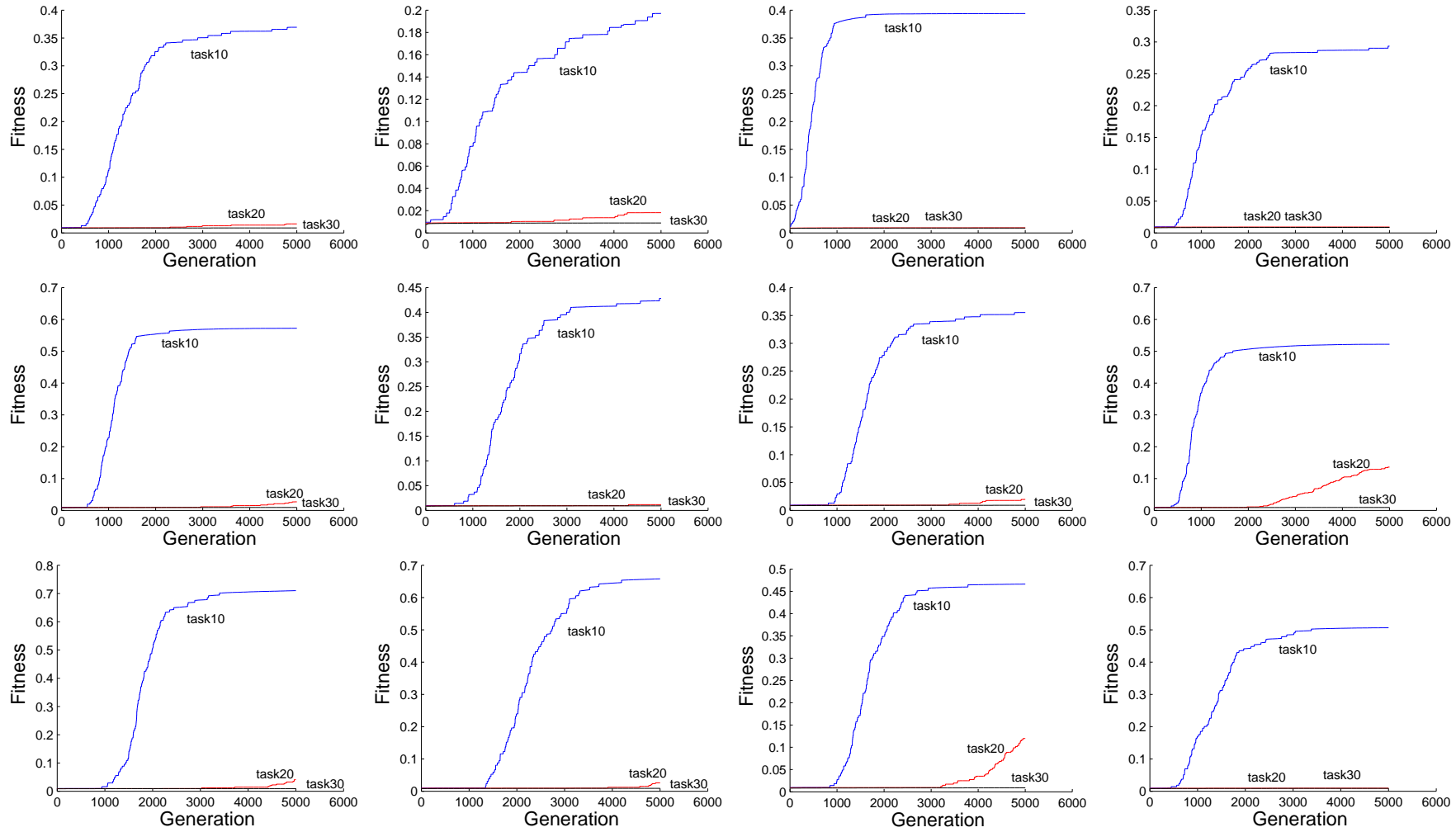


Fig. A.2. Employees and skills fixed (horizontal: 5, 10, 10/4-5, 10/6-7 skills, vertical: 5, 10, 15 employees).

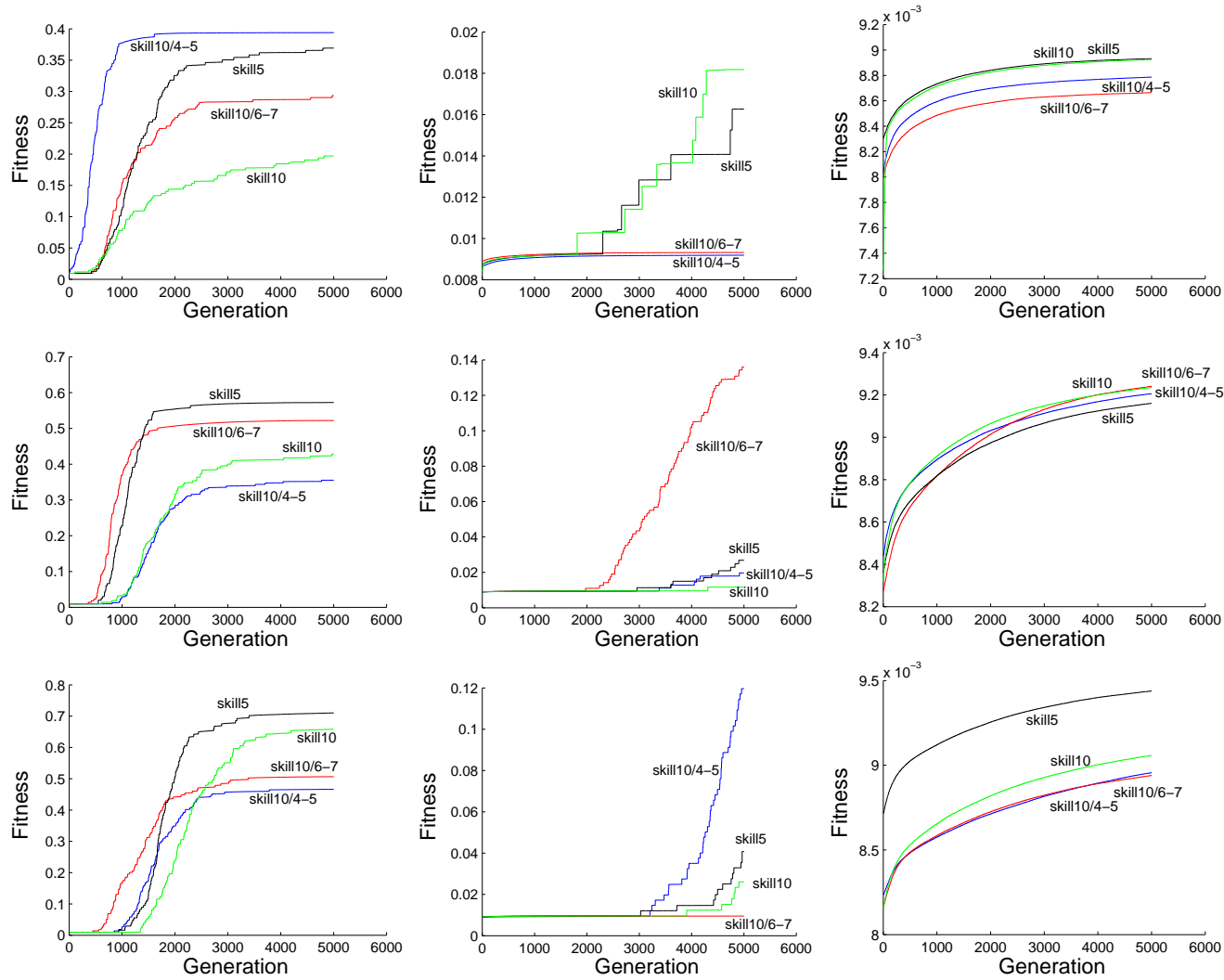


Fig. A.3. Employees and tasks fixed (horizontal: 10, 20, 30 tasks, vertical: 5, 10, 15 employees).

## 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 Guilherme Horta Travassos for easing the access to several publications and to an anonymous review for her/his constructive comments.

## 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. Jones, H.-H., D. 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 Trans. on Soft. Eng.* 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: Proc. 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. Jones, S. Yang, Genetic algorithm based software integration with minimum software risk, *Inf. and Soft. Tech.* 48 (3) (2006) 133–141.