# An Evolutionary Hyper-heuristic for the Software Project Scheduling Problem

Xiuli Wu[1(✉)], Pietro Consoli[2], Leandro Minku[3], Gabriela Ochoa[4], and Xin Yao[2(✉)]

[1] Department of Logistics Engineering, School of Mechanical Engineering, University of Science and Technology Beijing, Beijing, China
wuxiuli@ustb.edu.cn
[2] CERCIA, School of Computer Science, University of Birmingham, Birmingham, UK
{P.A.Consoli,X.Yao}@cs.bham.ac.uk
[3] Department of Computer Science, University of Leicester, Leicester, UK
leandro.minku@leicester.ac.uk
[4] Computing Science and Mathematics, University of Stirling, Stirling, UK
gabriela.ochoa@cs.stir.ac.uk

**Abstract.** Software project scheduling plays an important role in reducing the cost and duration of software projects. It is an NP-hard combinatorial optimization problem that has been addressed based on single and multi-objective algorithms. However, such algorithms have always used fixed genetic operators, and it is unclear which operators would be more appropriate across the search process. In this paper, we propose an evolutionary hyper-heuristic to solve the software project scheduling problem. Our novelties include the following: (1) this is the first work to adopt an evolutionary hyper-heuristic for the software project scheduling problem; (2) this is the first work for adaptive selection of both crossover and mutation operators; (3) we design different credit assignment methods for mutation and crossover; and (4) we use a sliding multi-armed bandit strategy to adaptively choose both crossover and mutation operators. The experimental results show that the proposed algorithm can solve the software project scheduling problem effectively.

**Keywords:** Software project scheduling · Hyper-heuristics · Adaptive operator selection · Sliding multi-armed bandit

## 1 Introduction

The Software Project Scheduling Problem (SPSP) relates to the decision of who does what task during a software project lifetime [1]. It plays an important role in reducing the duration and the cost of a software project [1, 15]. In China alone, it was reported that more than 40 % of unsuccessful software projects failed because of the inefficient planning of project tasks and human resources [8]. The SPSP, hence, is an important issue for IT companies.

However, the SPSP is particularly challenging when the project is large. The space of possible allocations of employees to tasks is enormous, and providing an optimal


J. Handl et al. (Eds.): PPSN XIV 2016, LNCS 9921, pp. 37–47, 2016.
DOI: 10.1007/978-3-319-45823-6_4

allocation of employees to tasks becomes a very difficult task [14]. It is impractical to use exact methods to solve medium or large SPSP instances. Evolutionary algorithms have been employed to solve the SPSP [1, 5, 12, 14, 16, 17]. Other metaheuristics have also been used, such as ant colony optimization and its variants [4, 6, 15]. A column generation approach was presented in [12], embedded within a branch-and-price procedure.

In those algorithms, different search operators (e.g., different types of crossover and mutation) may be good for different problem instances. However, little is known about which operators are most adequate for which types of instances. This motivates us to design an evolutionary algorithm capable of choosing the most effective operator automatically. Moreover, given a single problem instance, different search operators may be good at different stages of the search. As a result, it is very difficult to choose/design the operators to be used beforehand. Ideally, we would like an algorithm that can automatically choose which operators to use during the evolutionary process, and thus liberate practitioners from this difficult task [7]. This motivates our study of adaptive operator selection for the SPSP.

As a recent trend in optimization, hyper-heuristics search the space of heuristics rather than the space of solutions of the given problem, and use limited problem specific information to control the search process [9]. A hyper-heuristic is an automated methodology for selecting or generating heuristics to solve computational search problems [2]. We propose an evolutionary hyper-heuristic to solve the SPSP. Different from previous work on hyper-heuristics, our approach can be used to select both mutation and crossover operators, rather than being used to select only crossover or only mutation. We design different credit assignment methods for these two types of operators because mutation is typically used to exploit the solution space while crossover is typically used to explore it.

In summary, our novelty lies in the following: (1) this is the first work to adopt an evolutionary hyper-heuristic for SPSP; (2) this is the first work for adaptive selection of both crossover and mutation operators; (3) we design different credit assignment methods for the two types of operators: mutation and crossover; and (4) we use a sliding multi-armed bandit strategy to adaptively choose both crossover and mutation operators. We use a 3-sized crossover pool and a 3-sized mutation pool. Our experiments show that our approach is effective in selecting crossover and mutation operators for the SPSP.

The rest of this paper is organized as follows. Section 2 formulates the problem. Section 3 proposes an evolutionary hyper-heuristic for the SPSP. Section 4 reports the experimental results. Section 5 concludes the paper.

## 2 Formulation of SPSP

In this section, we explain the formulation of the SPSP [1, 11]. The notations adopted in the definitions are summarized in Table 1. A software project is composed of $N$ tasks. A Task Precedence Graph (TPG) describes the precedence relations among tasks. It is used together with the decision variable and the task required efforts in order to determine the start and finishing time of each task ($st_j$ and $ed_j$). This is done by creating a Gantt chart based on Algorithm 1 described in [11], which is omitted here

due to space constraints. Each task $t_j$ requires a set of skills $req_j$ and has an estimated effort $eff_j$. There are $M$ employees involved in the project. Each employee $e_i$ can be described as a three-tuple array $(e_i^{skill}, e_i^{max}, e_i^{norm\_sal})$. A project requires a total of $s$ skills. Each $skill^k$ ($k$ = 1, 2, …, $s$) represents a kind of software development skill in the project, such as system analysis, designing, coding, algorithm, database, quality check, testing, etc.

Employees can work on several tasks simultaneously, as indicated by their dedication to certain tasks. The dedication $x_{ij} \in \{0/k, 1/k, \ldots, k/k\}$ of employee $e_i$ to task $t_j$ is the fraction of the employee's time devoted to that particular task. $k \in \mathbb{N}$ represents the granularity of the problem. A dedication of $x_{ij} = 1$ indicates that the employee $e_i$ spends all his or her working time on task $t_j$. $x_{ij} = 0$ indicates that $e_i$ does not spend any time on $t_j$.$0 < x_{ij} < 1$ indicates that $e_i$ spends part of his or her working time on $t_j$. The matrix $\boldsymbol{X} = (x_{ij})$ of $M \times N$, where $x_{ij} \geq 0$, is the decision variable and represents a solution to the problem. This problem formulation [1, 11] assumes a static environment where employees will always be available during the lifetime of a project, i.e., they will not leave or be absent from work, and the task effort is fixed. As in [1, 11], we will also assume that $e_i^{max} = 1$ for all employees.

**Table 1.** SPSP notations

| | Description |
|---|---|
| $M$ | The number of the employees involved in the project |
| $e_i$ | The $i$-th employee |
| $e_i^{skill}$ | $e_i^{skill} = \{pro_i^1, pro_i^2, \ldots, pro_i^s\}$, $pro_i^k$ ($k$ = 1, 2, …, $s$) is a binary variable indicating whether the employee $e_i$ possesses the skill $skill^k$ |
| $e_i^{max}$ | The max dedication of $e_i$ to the project indicating the percentage of a full time employee $e_i$ is able to dedicate to the project. |
| $e_i^{norm\_sal}$ | The monthly salary for an employee $e_i$ for his or her full normal working time |
| $N$ | The software project is composed of $N$ tasks |
| $t_j$ | The $j$-th task |
| $eff_j$ | The estimated effort for the task $t_j$ |
| $req_j$ | The required skills for the task $t_j$ |
| TPG | The task precedence graph is an acyclic directed graph with tasks as nodes and task precedence as edges |
| $x_{ij}$ | The decision variable to determine the degree of dedication of employee $e_i$ to task $t_j$. |
| $st_j$ | The starting time of task $t_j$ |
| $ed_j$ | The finishing time of task $t_j$ |

The SPSP is the problem of assigning employees to tasks in a software project so as to minimize the completing time (i.e., duration of the project as defined by Eq. (2)), and the cost (i.e., the total amount of salaries paid as defined by Eq. (3)). Equation (1) is the mixed objective, where $w_1$ and $w_2$ are the weights for the completing time and the cost, respectively. The assignment of employees to tasks is to determine the decision variable $x$.

The problem is subject to the aforementioned assumptions and the following two constraints: employees can only work on a task $t_j$ if all employees working together have all the skills to perform the task (Eq. (4)); and employees should not exceed their maximum dedication to the tasks that are active at any given time moment $t$ (Eq. (5)).

$$\text{Minimize}_x f(x) = w_1 f_1(x) + w_2 f_2(x) \tag{1}$$

$$f_1(x) = \max_j(ed_j), \tag{2}$$

where $ed_j$, $\forall j$, is obtained with Algorithm 1 from [11].

$$f_2(x) = \sum_{i=1}^{M} \sum_{j=1}^{N} \left( eff_j \Big/ \sum_{k=1}^{M} x_{kj} \right) x_{ij}\ e_i^{norm\_sal}, \tag{3}$$

s.t.

$$\text{req}_j \subseteq \bigcup_{i=1}^{n} \{skill_i | x_{ij} > 0\} \tag{4}$$

$$\sum_{j \in \text{active\_tasks}(\tau)} x_{ij} \leq e_i^{max},\ \forall i, \tau, \tag{5}$$

where active_tasks$(\tau)$ are all tasks active at time $\tau$ according to the Gantt chart generated using Algorithm 1 from [11]

$$x_{ij} \in [0, 1] \tag{6}$$

It is worth noting that the SPSP is related to the Resource-Constrained Project Scheduling problem (RCPS), but there are some key differences [1]: (1) the SPSP has a cost associated to each employee; (2) SPSP has only one type of resource (employee); and (3) each activity in RCPS requires different quantities of different resources, whereas the SPSP requires different skills, which are not quantifiable entities.

## 3 The Evolutionary Hyper-heuristic

### 3.1 Hyper-heuristic Framework

The evolutionary hyper-heuristic (Fig. 1) chooses an operator to apply at each search stage. The high level algorithm is based on a $(\mu + \lambda)$-EA, i.e. it maintains a population of μ candidate solutions and λ parents are selected at each generation. Before each evolutionary cycle, the adaptive operator selection function is called twice (lines 3 and 4) to choose the crossover and the mutation operator, respectively. At the end of each iteration, the two credit assignment functions: diversity-credit and improvement-credit, are called (lines 12 and 13) to assign a credit to the currently chosen crossover and mutation operator, respectively.

```
1: initialize a population pop with μ candidate solutions
2: repeat
3:   crossover= Operators-selection(crosscredit)
4:   mutation= Operators-selection(mutationcredit)
5:   for i=1:2:λ
6:      select 2 parents x(1) and x(2) from pop at random
7:      apply crossover to x(1) and x(2) to generate  x'(1) and x'(2) with
probability Pc
8:      apply mutation to x'(1) and x'(2) with probability Pm
9:      pop=pop∪(x'(1),x'(2))
10:  end for
11:  Select the best μ solutions from pop to survive to the next gener-
ation
12:  Update crosscredit= diversity-credit(pop)
13:  Update mutationcredit= improvement-credit(pop)
14: until termination criteria are met
15: output the best candidate solution in pop.
```

**Fig. 1.** The evolutionary hyper-heuristic for SPSP

### 3.2 Adaptive Operator Selection

Adaptive operator selection (AOS) performs on-line selection of evolutionary operators to produce each new offspring, based on the recent known performance of each of the available operators. An adaptive operator selection is typically composed of a credit assignment and an operator selection rule. The former assigns a reward to an operator and the latter determines the operator to be chosen at each step. In the AOS framework, the performance of an operator in a very early stage may be irrelevant to its current performance [10]. More attention should be paid on the recent performance. We propose a sliding multi-armed bandit (SMAB) following the approach in [10]. The credit assignment and the operator selection rules adopted are as follows.

**Credit Assignment of SMAB.** To determine the credit assignment, one needs to make a decision on how to measure the impact in the search process caused by the application of an operator. We propose two credit assignment methods according to the main role each operator plays during the search process.

Considering that the main role of crossover is to explore the solution space, we employ the population diversity to evaluate the performance of one on-duty crossover operator. The diversity of the population is measured by the "population diversity", inspired by the entropy concept. It is calculated in Eq. (7) by computing the standard deviation of the same amount of dedication among solutions in the population.

$$ent = \sum_{i=1}^{M} \sum_{j=1}^{N} \sqrt{\frac{1}{\mu} \sum_{k=1}^{\mu} (x_{ij}^{(k)} - \frac{1}{\mu} \sum_{k=1}^{\mu} x_{ij}^{(k)})^2} \tag{7}$$

Considering that one mutation operator plays the role to guide a local search, we use the fitness improvements caused by the recent application of the operator under assessment. The fitness improvement is defined in Eq. (8).

$$r = (f_{old} - f_{new})/f_{old} \tag{8}$$

Where $f_{old}$ and $f_{new}$ is the fitness of the individual before and after the application of the operator respectively.

A sliding window with a fixed size $W$ is used to store the fitness improvement values of the recently used operators. The sliding window is a two-dimensional list of 2 rows and $W$ columns. The first row records the operator index number and the second row records the corresponding fitness improvement. It is organized as a first-in-first-out (FIFO) queue.

**Selection Rule of SMAB.** Based on the received credit values, the operator selection scheme selects one operator for generating new solutions. This paper uses a bandit-based operator selection scheme. Our scheme is similar to that in [10]. The major difference is that we use the entropy *ent* for the crossover operator and the fitness improvement value $r$ as the quality $\hat{q}_{i,t-1}$ instead of the average of all the rewards received so far for an operator. The operator that maximizes Eq. (9) will be chosen as the on-duty operator.

$$argmax_{i=1,\ldots k}\left(\hat{q}_{i,t-1} + C\sqrt{\frac{2\log\sum_k n_{k,t}}{n_{i,t}}}\right) \tag{9}$$

where $\hat{q}_{i,t-1}$ is the empirical reward (the best result achieved by the operator in last $W$ iterations) of the $i$-th arm (operator), $C$ is a scale parameter, $n_{i,t}$ is the times that the $i$-th arm has been tried till the $t$-th iteration during the recent $W$ applications.

### 3.3 The Low-Level Heuristics Pool

**Crossover Operator Pool.** There are 3 operators in the crossover operator pool.

*Crossover 1: Swap-Row Crossover.* For each employee, select its corresponding dedications to tasks from one randomly chosen parent to generate an offspring. This can be seen as changing some employees' dedication to tasks [11].

*Crossover 2: Swap-Column Crossover.* For each task, select its corresponding employees' dedications from one randomly chosen parent to compose an offspring. This can be seen as exchanging some tasks' resource assignment [11].

*Crossover 3: Swap-Block Crossover.* The Swap-Block Crossover [1] is a 2-D single point crossover applied to matrices. It randomly selects a row and a column (the same in the two parents) and then swaps the elements in the upper left quadrant and the lower right quadrant in both solutions.

**Mutation Operator Pool.** There are 3 operators in the mutation operator pool.

*Mutation 1: Mutate-Position* [11]. An individual is mutated by changing each entry $x_{ij}$ of the dedication matrix to a random times of 1/7 with mutation probability, independently from other entries.

*Mutation 2: Mutate-Row.* An individual is mutated by swapping randomly dedications in each row. The selected positions to swap must be the nonzero dedication to keep the balance.

*Mutation 3: Mutate-Column.* An individual is mutated by swapping randomly dedications in each column.

## 4 Experimental Study

We run a number of experiments to determine whether the use of the proposed AOS technique is beneficial with respect to the adoption of a simple random selection of the crossover and mutation operators. We also compare the results of these algorithms with those achieved by a state-of-the-art approach [11]. We label the three algorithms for convenience GA-SMAB, GA-randAOS and GA.

We performed our experiments on a benchmark dataset of 48 instances used in [1]. For each instance, we provide the average results from 30 independent runs of the algorithms. In order to reduce the impact of different parameter settings on the results, we adopt the same parameter settings as those used in literature [1, 11] (Table 2) except for the mutation probability, which was set to a value that guaranteed the application of the mutation operator, necessary to observe its performance.

**Table 2.** Parameters setting

| Parameter | Value | Description |
|---|---|---|
| $\mu$ | 64 | The size of the population |
| $\lambda$ | 64 | The size of the offspring |
| *Pc* | 0.75 | The crossover probability |
| *Pm* | 0.1 | The mutation probability |
| *maxg* | 200 | The number of generations |
| $w_1$ | $10^{-1}$ | The weight of the duration |
| $w_2$ | $10^{-6}$ | The weight of the cost |
| *winsize* | 7 | The sliding window size |
| *ScalingC* | 60 | The scaling factor for crossover operators |
| *ScalingM* | 110 | The scaling factor for mutation operators |

Table 3 reports the average results achieved by the GA-SMAB, GA-randAOS and GA algorithms on the 48 instances of the benchmark set. For every algorithm we report the average fitness, its standard deviation, the best result, the average cost and the average completion time. The results indicate how the fitness achieved by GA-SMAB is the lowest of the three. In particular, the average fitness of GA-SMAB is slightly lower than the best of GA. Similarly, GA-randAOS also shows a comparable improvement with respect to the GA. It is also worth noting that the cost has increased, while the average completing time has improved considerably. This is likely to be a result of the weights $w_1$ and $w_2$ used in the fitness function.

Table 4 summarizes the results of the Wilcoxon Rank-Sum (significance level of 0.05) test performed for each instance to determine the number of instances for which each algorithm yields statistically better (column W) results, comparable results (column T) and statistically worse results (column L). We also provide the p-values relative to the Wilcoxon Signed-Rank (significance = 0.05) test performed on the average fitness achieved by the three algorithms across the 48 instances of the benchmark set where instances with comparable results (according to the Wilcoxon Rank-Sum test) are treated as ties.

**Table 3.** Average results achieved by GA-SMAB, GA-randAOS and GA algorithms

| | GA-SMAB | GA-RANDAOS | GA |
|---|---|---|---|
| Average fitness | 4.5351 | 4.5552 | 4.6369 |
| Standard dev of fitness | 0.0856 | 0.1009 | 0.0509 |
| Best fitness | 4.3617 | 4.3709 | 4.5534 |
| Average cost | 1,830,377.1444 | 1,830,223.0918 | 1,830,037.0659 |
| Average completing time | 27.0460 | 27.2475 | 28.0686 |

**Table 4.** Win/tie/losses obtained based on Wilcoxon Rank-Sum tests for each instance, p-values of the Wilcoxon Signed-Rank test across instances and average Cohen's d effect size

| | GA-SMAB | | | GA-randAOS | | | GA | | |
|---|---|---|---|---|---|---|---|---|---|
| | w | t | l | w | t | l | w | t | l |
| GA-SMAB | | | | 25 | 23 | 0 | 44 | 4 | 0 |
| | | | | 1.23E-05 | | | 7.62E-09 | | |
| | | | | 0.6368 | | | 1.5688 | | |
| GA-randAOS | 0 | 23 | 25 | | | | 37 | 11 | 0 |
| | 1.23E-05 | | | | | | 1.14E-07 | | |
| | 0.6368 | | | | | | 0.9814 | | |
| GA | 0 | 4 | 44 | 0 | 11 | 37 | | | |
| | 7.62E-09 | | | 1.14E-07 | | | | | |
| | 1.5688 | | | 0.9814 | | | | | |

The three algorithms produce statistically significantly different results across problem instances, as shown by the Wilcoxon Sign-Rank tests. The average Cohen's d effect sizes vary from 0.6368 (medium) to 1.5688 (large). Together with the win-tie-losses, this shows that it is beneficial to adopt GA-SMAB instead of GA-randAOS or GA. GA-SMAB achieved similar or better fitness than GA-randAOS and GA on all problem instances. In particular, GA-SMAB achieved statistically better fitness on 25 instances when compared to GA-randAOS, and similar fitness on 23 instances. This suggests that there are instances which do not require an AOS strategy, where GA-SMAB performs similarly to GA-randAOS. The improvement obtained through the use of GA-SMAB is also confirmed by the Cohen's d Effect size included in Table 4. However, an AOS strategy is needed for other problem instances.

When compared to the results achieved by GA, GA-SMAB outperforms its results on 44 instances. This difference can be explained by the interaction between the random parent selection and the SMAB strategy, as the increased exploration ability of the algorithm created more favorable conditions for the GA-SMAB.

In order to show the behavior of the algorithm, we provide the selection rates of the crossover and mutation operators for the instance *inst-employees20* respectively in Fig. 2. It is possible to notice trends in the search as one operator is preferred to the others during different periods of the search. This is particularly clear in the selection rates of the mutation operator, where operator Mutate-Column has a higher selection rate for most of the search, with the exception of some periods where the other two operators are preferred. In the plot relative to the crossover operator, on the other hand, it is possible to notice shorter trends over the course of the search, although Operator Swap-Block seems to be the one selected most of the times. This might be explained by the fact that the algorithm favors a frequent alternation of the crossover operators, as the repeated use of a single operator might cause a decrease of the population diversity.

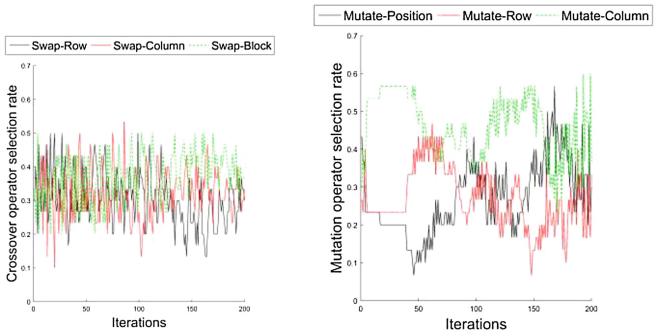


**Fig. 2.** Selection rates of the crossover (left) and selection rates of the mutation (right)

## 5 Conclusions

This paper proposes an evolutionary hyper-heuristic to address the SPSP. The hyper-heuristic uses an EA as a high level strategy and adapt automatically both mutation and crossover operators during evolution. A sliding window MAB strategy is used to adaptively select both operators during the search. The experiments performed on a set of 48 benchmark instances showed that the proposed algorithm can solve the SPSP effectively and outperform a strategy based on a simple random selection of the operators as well as a state-of-the-art approach from the literature. Future work includes a detailed analysis of the behavior of the proposed algorithm and the reasons for its ability to generate better solutions; an extension of the proposed algorithm in order to deal with the dynamic SPSP [13]; the use of alternative AOS strategies; and the inclusion of more aspects that could affect software projects into the problem formulation.

**Acknowledgements.** This paper was partly supported by the National Natural Science Foundation of China under Grant (Grants. 51305024 and 61329302) and EPSRC (Grant No. EP/J017515/1). Xin Yao was supported by a Royal Society Wolfson Research Merit Award.

## References

1. Alba, E., Francisco, C.: Software project management with GAs. Inf. Sci. **177**, 2380–2401 (2007)
2. Burke, E., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., Woodward, J.: A classification of hyper-heuristic approaches. In: Gendreau, M., Potvin, J.-Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research & Management, vol. 146, pp. 449–468. Springer, Berlin (2010)
3. Blazewicz, J., Lenstra, J., Rinnooy, K.: Scheduling subject to resource constraints: classification and complexity. Discret Appl. Math. **5**, 11–24 (1983)
4. Crawford, B., Soto, R., Johnson, F., Monfroy, E., Paredes, F.: A max-min ant system algorithm to solve the software project scheduling problem. Expert Syst. Appl. **41**, 6634–6645 (2014)
5. Chang, C., Jiang, H., Di, Y., Zhu, D., Ge, Y.: Time-line based model for software project scheduling with genetic algorithms. Inf. Softw. Tech. **50**, 1142–1154 (2008)
6. Chen, W., Zhang, J.: Ant colony optimization for software project scheduling and staffing with an event-based scheduler. IEEE Trans. Softw. Eng. **39**(1), 1–17 (2013)
7. Consoli, P.A., Minku, L.L., Yao, X.: Dynamic selection of evolutionary algorithm operators based on online learning and fitness landscape metrics. In: Dick, G., et al. (eds.) SEAL 2014. LNCS, vol. 8886, pp. 359–370. Springer, Heidelberg (2014)
8. Ding, R., Jing, X.: Five principles of project management in software companies. In: Project Management Technology, vol. 1 (2003). (in Chinese)
9. Jorge, A., Alcaraz, S., Ochoa, G., Swan, J., Carpio, M., Puga, H., Burke, E.: Effective learning hyper-heuristics for the course timetabling problem. Eur. J. Oper. Res. **238**, 77–86 (2014)
10. Li, K., Fialho, A., Kwong, S., Zhang, Q.: Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. IEEE Trans. Evol. Comput. **18**(1), 114–130 (2014)
11. Minku, L., Sudholt, D., Yao, X.: Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis. IEEE Trans. Softw. Eng. **40**(1), 83–102 (2014)
12. Montoya, C., Bellenguez-Morineau, O., Pinson, E., Rivreau, D.: Branch-and-price approach for the multi-skill project scheduling problem. Optim. Lett. **8**, 1721–1734 (2014)
13. Shen, X., Minku, L., Bahsoon, R., Yao, X.: Dynamic software project scheduling through a proactive-rescheduling method. IEEE Trans. Softw. Eng. 24 December 2015. doi:10.1109/TSE.2015.2512266
14. Penta, M., Harman, M., Antoniol, G.: The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. Softw. Pract. Exper. **41**, 495–519 (2011)
15. Xiao, J., Ao, X., Tang, Y.: Solving software project scheduling problems with ant colony optimization. Comput. Oper. Res. **40**, 33–46 (2013)

16. Xiao, J., Osterweil, L.J., Wang, Q., Li, M.: Dynamic resource scheduling in disruption-prone software development environments. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 107–122. Springer, Heidelberg (2010)
17. Yannibelli, V., Amandi, A.: A knowledge-based evolutionary assistant to software development project scheduling. Expert Sys. Appl. **38**, 8403–8413 (2011)