SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:551–577
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1030

Constraint-based human resource allocation in software projects

Dongwon Kang[1,*,†], Jinhwan Jung[2] and Doo-Hwan Bae[1]

[1]*Department of Computer Science, College of Information Science and Technology, Korea Advanced Institute of Science and Technology (KAIST), 373-1 Guseong-dong, Yuseong-gu, Daejeon, Korea*
[2]*Defense Agency for Technology and Quality, Cheongnyang PO Box 276, Dongdaemun-gu, Seoul 130-650, Korea*

SUMMARY

Resource allocation in a software project is crucial for successful software development. Among various types of resources, human resource is the most important as software development is a human-intensive activity. Human resource allocation is very complex owing to the human characteristics of developers. The human characteristics affecting allocation can be grouped into individual-level characteristics and team-level characteristics. At the individual level, familiarity with tasks needs to be taken into account as it affects the performance of developers. In addition, developers have different levels of productivity, depending on their capability and experience; the productivity of developers also varies according to tasks. At the team level, characteristics such as team cohesion, communication overhead, and collaboration and management also affect human resource allocation. As these characteristics affect the efficiency of project execution, we treat them as constraints of human resource allocation in our approach. We identify individual-level constraints and team-level constraints based on the literature and interviews with experts in the industry. With these constraints, our approach optimizes the scheduling of human resource allocations, resulting in more realistic and efficient allocations. We also provide a guideline supporting various factors, with respect to roles and module characteristics, to estimate the productivity of developers based on COCOMO II. As productivity data are hard to obtain and manage, our guideline can provide a useful direction for human resource allocation in case of software projects. To validate our proposed approach, we document a case study using real project data. Copyright © 2011 John Wiley & Sons, Ltd.

Received 14 August 2009; Revised 7 July 2010; Accepted 27 September 2010

KEY WORDS: human resource allocation; constraints; productivity estimation; task allocation

1. INTRODUCTION

The failure of software development projects is often a result of inadequate planning and allocation of human resources [1]. The main objective of human resource allocation is to determine what task is to be performed by whom. Without this, the efficiency of a software project cannot be achieved because developers may be involved in tasks in which their capabilities are not maximized.

Resource allocation has been researched in various areas, such as operations research and management science, including load distribution, production planning, and computer scheduling [2]. Compared to these areas, human resource allocation in software projects is especially complex because the human characteristics of developers affect the allocation. We identified the following characteristics related to developers and categorized them into individual-level characteristics and team-level characteristics.

*Correspondence to: Dongwon Kang, Department of Computer Science, College of IT, KAIST, Korea.
†E-mail: dwkang@se.kaist.ac.kr

- Individual-level characteristics
  - *Learning curve*: When a developer performs a task, the performance gradually improves along the learning curve [3]. Thus, it would be efficient to allocate developers to tasks with which they are familiar.
  - *Different productivity levels*: Unlike machines which may have identical productivity, developers have different levels of productivity, which vary according to their capability and experience. Moreover, the productivity of a developer varies according to the types of tasks assigned.
- Team-level characteristics
  - *Need for team cohesion*: A software development team needs to act as one cohesive unit to increase the efficiency of development [4]. Thus, team members should work closely together and support one another for effective development.
  - *Overhead in communication*: When developers work together, communication between developers lowers the efficiency of a project [5–8]. Moreover, communication overhead is higher when communication occurs between teams [9].
  - *Need for collaboration and management*: As software development is performed in teams, collaboration is required among developers, and the performance of a developer may be affected by other developers. In particular, a manager in a software development team ensures that developers do the correct work and get the tasks done in the most efficient and least problematic way [10]. Thus, it is important to allocate capable developers to each team to lead the development and guide other developers.

As these developer-related characteristics affect the efficiency of project execution, we accommodate them as constraints of human resource allocation in our approach. We identify five constraints based on the literature and interviews with experts in the industry. These constraints are categorized at the individual and team levels following the characteristics mentioned above. They are described as follows:

- Individual-level constraints
  - *Constraint on phase-level continuity*: As mentioned above, familiarity with tasks affects the performance of developers. Thus, this needs to be accounted for in phase-level allocation. Allocating a developer to different modules in each phase may lower productivity as the developer would need to spend a considerable amount of effort in understanding and analyzing the results from the previous phases. Thus, we encourage maintaining phase-level continuity in the allocation.
  - *Constraint on increment-level continuity*: In incremental software development, if developers participate in developing modules in which they were not involved during the previous increment, it may lower productivity as the developer would need to spend additional effort in understanding and analyzing the results from the previous increments. Thus, we encourage maintaining increment-level continuity in the allocation.
- Team-level constraints
  - *Constraint on sharing developers*: According to the need for team cohesion, team members need to be unified into a working group acting as a unit [4]. In addition, the communication cost between teams is high [9]. Hence, developers are basically not shared between teams in practice, except for some tasks such as acceptance testing that need cooperation. Sharing of developers between teams lowers efficiency because those developers are under the control of different teams. This causes difficulty in scheduling teams and balancing workload for shared developers, and leads to commitment problems when each team requires high commitment from the shared developers. For this reason, sharing of developers between teams is discouraged.
  - *Constraint on the number of developers*: If too many developers participate in developing a module, the productivity is lowered because of the increase in communication overhead. Thus, our approach restrains excessive participation of developers in a module.

DOI: 10.1002/spe

- *Constraint on novice teams*: If a team consists of only novice developers, the quality of product cannot be guaranteed and a considerable amount of rework may be required. Thus, our approach restrains the team composition with novices only.

With these constraints, our approach optimizes the scheduling of human resource allocations, using accelerated simulated annealing (ASA) [11], which is a variation of simulated annealing (SA) [12]. As the constraints impose additional overhead on the project duration, they are handled as soft constraints which lead to penalty on the duration when violated. In the optimization, we use different levels of productivity with respect to roles and module characteristics in addition to applying the constraints. Thus, our approach can provide more realistic and efficient allocation results.

In addition, we provide a guideline to estimate productivity. Although many human resource allocation approaches have been proposed, it is not appropriate to use them if developers' productivity data are not available. Because it is difficult to establish a productivity scheme and accumulate a large historical dataset, especially for low-maturity organizations, we provide a guideline to estimate productivity by using the existing effort estimation model COCOMO II [13]. As manual productivity estimation is time-consuming and prone to bias, this guideline can be helpful when using human resource allocation approaches for software projects.

One of the limitations of existing approaches is that they are not validated with real project data. As these approaches generally use artificial data in experiments, their applicability is questionable. In this paper, we describe a case study using real project data to show the usefulness of our approach.

The remaining sections are organized as follows. Section 2 introduces the basics of SA and ASA algorithms. Section 3 describes the meta-models for human resource allocation in a software project. Section 4 presents the constraints used in this paper in detail, illustrating them with examples. In Section 5, we explain the constraint-based human resource allocation method. As productivity information is used as input for human resource allocation, a guideline to estimate productivity is also described in this section. Section 6 describes a case study to show the usefulness of our approach. In Section 7, we introduce the related approaches and compare them with our approach. Finally, we conclude in Section 8 with a summary and discussion of the future direction.

2. BACKGROUND: SIMULATED ANNEALING

The SA algorithm is based on the analogy between the simulation of the annealing of solids and the problem of solving large combinatorial optimization problems [14]. It works by emulating the physical process whereby a solid is slowly cooled so that when eventually its structure is *frozen*, this happens at a minimum energy configuration [15]. It was introduced first in its original form [16] as a numerical minimization method and then modified [12] to incorporate a cooling schedule.

SA finds a new solution by changing an existing one and determines its quality using a cost function. One of the most important characteristics of SA is that it can avoid premature convergence to local optimum by accepting non-improving moves.

So far, many variations of SA have been proposed to improve the efficiency. Among them, we used the ASA [11] in our approach. The algorithm of ASA is presented as Algorithm 1.

In the INITIALIZE function, the initial temperature (T), the initial solution (X), and the number of internal loops (L) are decided. The ASA algorithm has two loops. In the internal loop, the ASA algorithm creates a new solution (Y) nearby the current solution X using the PERTURB function. If the cost of Y given by the COST function is lower than the current solution X, then the new solution is accepted as the current solution. When the cost of Y is higher than X, Y is accepted with a certain probability to prevent an accepted solution from remaining in a local optimum. In this case, the probability of accepting Y is decided based on the costs of X and Y and T using an exponential function (exp). The lower the T, the lower the probability of accepting Y. Then, if the cost of the accepted solution is lower than that of the best solution (X_best), the new solution is stored as the best solution.

DOI: 10.1002/spe

Algorithm 1 Algorithm ASA

```
1: INITIALIZE (X,T,L);
2: X_best ← X;
3: Counter1 ←0;
4: Counter2 ←0;
5: repeat
6:    Costold ←COST(X);
7:    Check ←0;
8:    for i ←1 to L do
9:       Y ← PERTURB(X);
10:      if (COST(Y) < COST(X)) or (exp((COST(X) − COST(Y))/T)>random(0,1)) then
11:         X ← Y; {accept the movement}
12:      if COST(X) < COST(X_best) then
13:         X_best ← X;
14:         Counter2 ←0;
15:         Check ←1;
16:      else
17:         Counter2 ←Counter2+1;
18:   Costnew ← COST(X);
19:   if Check=1 or Costnew<Costold then
20:      T =αT;
21:   if  Costnew=Costold  then
22:      Counter1 ←Counter1+1;
23:   else
24:      Counter1 ←0;
25: until Counter1 > M or Counter2 > N
```

In SA algorithms, a temperature is lowered when the system reaches a steady state (quasi equilibrium) by accepting at least some fixed number of solutions at the temperature [12, 17]. Thus, if ASA finds a better solution in the internal loop, then ASA cools down the temperature as it implies that a high-quality solution has been found in the current temperature and ASA can take the next step in the lower temperature. Otherwise, ASA executes the internal loop with the same temperature. In this manner, the number of internal loops (L) can be set to a small number, and ASA can thus prevent executing the internal loop too many times with a large L value. Commonly, T is cooled down using a cooling factor α as $T_k = \alpha T_{k-1}$. Alpha is set to a value close to 1; a typical value is 0.95 [18].

ASA algorithm stops when the cost is not changed in M external loops or the best cost is not changed in N internal loops. Using the number of non-improving moves in both loops as the stopping criterion, ASA can reduce the time wasted in non-improving loops at low temperatures.

Because of these characteristics of ASA, it was empirically validated that ASA required much less computation time than the conventional SA while their solutions had the same level of quality [11].

3. SPECIFICATION OF META-MODELS FOR HUMAN RESOURCE ALLOCATION IN SOFTWARE PROJECTS

This section describes the meta-models for the human resource allocation problem in software projects. The overview of the meta-model structure is presented in Figure 1.

DOI: 10.1002/spe

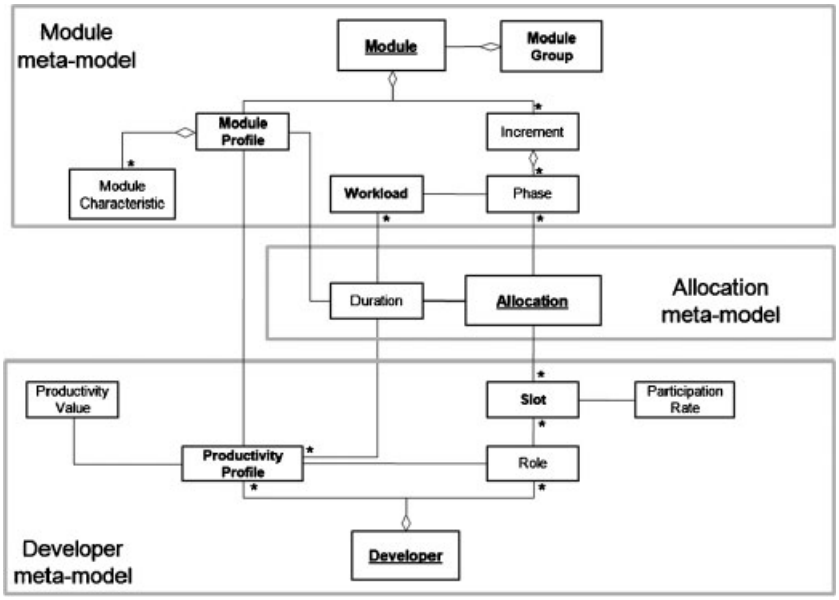

Figure 1. The meta-models for human resource allocation.

3.1. Module meta-model

A module is mainly described with workload and a module profile. As a module may be developed in several increments, each of which consists of several development phases, workload given at the phase level is the basic unit of allocation in this paper. Therefore, workload is described as $wload(m, i, p)$, where m, i, and p represent a module, an increment, and a phase, respectively. $wload$ presents the amount of effort required when development is performed by developers with average productivity. In other words, it is the estimated effort without considering the developers' productivity.

A module profile specifies the characteristics of a module and is represented as a combination of characteristics that differentiates the productivity of developers regarding the module. These characteristics can be development languages, types of functionality or technologies used in development. The characteristics can be defined by experts according to project environments. For example, suppose that Modules A and B are to be developed in C++ and provide visualization functionality, while Module C is to be developed in Java and provides data-processing functionality. In this case, languages and functionality can be the characteristics in a module profile because a developer may have different levels of productivity according to them. In this case, a module profile for A is represented as mpf(C++,Visualization).

Note that our approach uses a phase in developing a module as the unit for allocation. There can be other phases such as acceptance testing which are applied to whole modules. In this case, the effect of optimization is not significant because choosing the best developers for these phases is enough. Hence, our approach only deals with phases related to modules, similar to [19].

As developers collaborate as a team, they can be first grouped into teams before performing individual-level allocation. For team-level allocation, we propose the concept of a *module group*. It is defined as a set of modules which becomes a unit of team allocation. The composition of a module group can be conducted by experts considering module profiles, functional relationship of modules, or size of modules.

3.2. Developer meta-model

A developer mainly has two types of information: productivity profiles and slots. A productivity profile is used to support various productivity levels of a developer according to a role and a module

DOI: 10.1002/spe

profile. As there can be various combinations of characteristics for module profiles and several roles can be used in a software project, a developer has a set of productivity profiles. A productivity profile can be described as $prod(d, r, mpf)$, where d, r, and mpf represent a developer, a role, and a module profile, respectively. In this paper, it is assumed that a role of a software project is given for each development phase. Given these inputs, $prod$ presents a productivity value. Generally, productivity is measured as the ratio of size over effort. As size metrics vary according to each software development organization, we simply use a normalized productivity value for the productivity measure to describe the differences in the capability of developers, as used by Ngo-The *et al*. [19]. For example, if a developer d has an average capability for a given module profile mpf and a role r, then $prod(d, r, mpf) = 1.0$.

In addition to productivity profiles, a developer has slots to support partial allocation, because s/he may be allocated to multiple modules [7, 20, 21]. A slot is described as $slot(d, i, p, k)$, where d, i, p, and k represent a developer, a phase, an increment, and a slot number, respectively. Given these inputs, $slot$ represents a participation rate, which specifies the degree of participation of a slot. The sum of participation rates of slots in a phase should be maintained at 100%.

3.3. Allocation meta-model

Modules are allocated to the slots of developers in each phase in the allocation procedure. Thus, allocation is described as $allo(d, m, i, p, k)$, where d, m, i, p, and k represent a developer, a module, an increment, a phase, and a slot number, respectively. It takes a value of 1 if and only if d is allocated to m in the kth slot of p in i. Otherwise, it takes 0.

When the allocation is performed for all developers and modules, the estimated duration for a project can be obtained using productivity profiles, workload, and module profiles. First, given a set of developers D and a role r which is related to p, the estimated duration of a phase p in developing a module m in an increment i can be obtained by the following equation:

$$Duration(m, i, p) = \frac{wload(m, i, p)}{\sum_{j=1}^{|D|} \sum_{k=1}^{S(j,i,p)} allo(D(j), m, i, p, k) \times slot(D(j), i, p, k) \times prod(D(j), r, mpf)},$$

where mpf and $S(j, i, p)$ represent the module profile of m and the number of slots of $D(j)$ in p of i, respectively.

Then, given a set of phases P, the duration of a module m in an increment i is obtained by summing up the phase duration as follows:

$$Duration(m, i) = \sum_{j=1}^{|P|} Duration(m, i, P(j)).$$

Given a set of modules M, the duration of the project in an increment i is calculated by the following equation:

$$Duration(i) = Max(Duration(M(1), i), \ldots, Duration(M(n), i)) \quad \text{where } n = |M|.$$

The total duration of a project is obtained by summing up the estimated duration for all increments.

In the equation for $Duration(i)$, it is assumed that modules are developed in parallel. This is based on the interview results with experts in the industry that the dependency of modules does not much influence the sequence of development by establishing interfaces among modules.

The goal of this paper is to provide an optimized human resource allocation for software projects in the schedule perspective. Thus, we try to minimize the total duration of a project based on the equations above. However, productivity and workload may not be enough to produce a feasible allocation because there exist human-related constraints related to software projects. In the following section, these constraints are explained in detail.

DOI: 10.1002/spe

4. HUMAN RESOURCE ALLOCATION CONSTRAINTS IN SOFTWARE PROJECTS

To achieve a reasonable human resource allocation in software projects, characteristics related to developers need to be considered. As these characteristics affect the efficiency of project execution, we consider them as allocation constraints in our approach.

We have identified the constraints from two sources: literature and interviews with experts in the industry. The interviews were performed with three experts in two software development companies who had more than 10 years of experience in the industry. In the interviews, we examined the allocation information of real project data with the experts. As the allocation is performed according to experts' experience rather than fixed rules, we extracted potential rules based on the project data, and the experts confirmed that they generally hold in software projects.

The constraints are categorized as individual-level constraints and team-level constraints. Each constraint is explained below with an illustrative example, which has been made up to compare different allocation policies regarding the constraint. For simplicity of describing examples in this section, we assumed that all developers have equal productivity for all module profiles and roles, unless additionally specified.

4.1. Individual-level constraints

As software development is a human-intensive activity, effort is required to understand tasks. Thus, it is important to maintain the continuity of allocation in order to minimize the effort needed to understand the result of the previous development. For this reason, we considered allocation continuity at the phase level and increment level in this approach.

- Constraint on phase-level continuity

According to the effect of learning curve [3], it is required to allocate developers to tasks with which they are familiar to increase the efficiency of performing tasks. Development of a module consists of several development phases, such as requirement analysis, design, implementation, and testing. If a developer is allocated to different modules in each phase, the developer should spend a considerable amount of time to understand and analyze the results from previous phases. This situation may occur when we only focus on minimizing the duration of each phase. We compare allocation policies regarding phase-level allocation with an example in Table I. Productivity of developers in this example is described in Table II.

As mentioned in Section 3, the duration for a phase is calculated by dividing the workload with the sum of productivity of participating developers, and the total duration is decided as the maximum duration among modules.

Table I. Comparison of allocation policies regarding phase-level continuity.

		Allocation 1		Allocation 2	
	Phases (Workload)	Developers	Duration	Developers	Duration
Module 1	Req. analysis (1.5)	A	1.0	A	1.00
	Design (1.0)	A	0.67	B	1.00
	Implementation (2.5)	A, D	1.0	A, C	0.83
	Testing (2.0)	A, D	0.8	B, D	1.00
	Subtotal		3.47		3.83
Module 2	Req. analysis (1.0)	B	1.0	B	1.00
	Design (1.2)	B	1.2	A	0.80
	Implementation (2.0)	B, C	0.8	B, C	0.80
	Testing (2.5)	B, C	1.0	C, D	1.00
	Subtotal		4.0		3.60
	Total		4.0		3.83

DOI: 10.1002/spe

Table II. Productivity matrix for Table I.

Developer	Req. analysis	Design	Implementation	Testing
A	1.5	1.5	1.5	1.5
B	1	1	1	1
C	—	—	1.5	1.5
D	—	—	1	1

Table III. Comparison of allocation policies regarding increment-level continuity.

		Allocation 1		Allocation 2		Allocation 3	
	Module (Workload)	Developers	Duration	Developers	Duration	Developers	Duration
Inc. 1	Module 1 (4)	A,B	1.54	A,B	1.54	A,B	1.54
	Module 2 (7)	C,D,E,F	1.75	C,D,E,F	1.75	C,D,E,F	1.75
	Subtotal		1.75		1.75		1.75
Inc. 2	Module 1 (7)	A,B	2.69	A,B,E	1.94	C,D,E,F	1.54
	Module 2 (5)	C,D,E,F	1.25	C,D,F	1.67	A,B	1.92
	Subtotal		2.69		1.94		1.92
	Total		4.44		3.69		3.67

In the example, two developers are involved from the beginning, and two additional developers are allocated from the implementation phase according to the different staffing levels required for each phase.

In Allocation 1, developers are continuously involved in each module. Thus, the effort required to understand previous phases is minimized. On the other hand, Allocation 2 is performed regardless of the module worked on in previous phases. Although the productivity-based calculation indicates Allocation 2 to have a shorter estimated duration, this type of allocation is discouraged in practice as it requires effort to understand previous phases. To prevent this situation, we propose the constraint on phase-level continuity. Given a developer d, a module m, an increment i, this constraint can be formally described as the following condition:

- if $allo(d, m, i, P_d(j)) = 1$, then $allo(d, m, i, P_d(j-1)) = 1$,
  where P_d is the set of phases in which d participates and $1 < j \leq |P_d|$.

In the equation above, $allo(d, m, i, p) = 1$ if there exists any slot k such that $allo(d, m, i, p, k) = 1$. Else, $allo(d, m, i, p) = 0$. It is also used in describing other constraints below.

- Constraint on increment-level continuity

Incremental software development is gaining wide acceptance as an approach to handling delivering parts of the software product early [13]. Thus, we consider the characteristics of incremental development in this approach. Similar to what we explained previously, developers need to be continuously involved in modules over increments.

However, there could be an allowable exception. As features of a module to be developed for each increment are determined by business decisions, workload of a module may vary per increment. In this case, it may be required to allocate some developers of the previous increment to other modules that require more developers in later increments. We describe this situation in the following example in Table III and compare the allocation policies to deal with this situation. In the example, productivity of Developers A and B is set to 1.3, and productivity for other developers is set to 1.0.

Allocation 1 does not allow for changes between increments. In this case, the efficiency of the project is reduced because the change in the module workload cannot be addressed in Increment 2.

The result from Allocation 2 is preferable because allocation in Increment 2 is adjusted according to the change in the workload while preserving the allocation of the previous increment as much

DOI: 10.1002/spe

Table IV. Comparison of allocation policies regarding sharing developers.

	Allocation 1		Allocation 2	
Module (Workload)	Developers	Duration	Developers	Duration
Module 1 (7)	A (40%), B (50%), C (100%)	3.64	A (60%), B (60%), E (50%)	3.58
Module 2 (5)	A (60%), B (50%)	3.62	A (40%), B (40%), C (40%)	3.78
Module 3 (8)	D (30%), E (100%), F (50%)	4.23	D (60%), E (50%), F (60%)	4.10
Module 4 (6)	D (70%), F (50%)	3.97	C (60%), D (40%), F (40%)	4.05
Total		4.23		4.10

as possible. Thus, Increment 2 can be performed efficiently with minimal additional effort required to understand the previous result.

In the case of Allocation 3, allocation is performed to minimize the estimated duration of each increment and has the shortest estimated duration of the three cases. However, it is impractical to remove developers of the previous increment and to add new developers simultaneously because it increases the overhead costs involved in understanding the results of the previous increment. In addition, it is difficult to transfer knowledge from the previous developers to the new developers. These can be overhead to the project and may increase the project duration. To prevent this situation, we propose the constraint on increment-level continuity. Given a set of developers D, a module m, a set of increments I, and a phase p, this constraint can be more formally described as follows:

- if there exists $d_1 \in D$ such that $allo(d_1, m, I(j-1), p) = \alpha$ and $allo(d_1, m, I(j), p) = \beta$, there exists no $d_2 \in D$ such that $allo(d_2, m, I(j\text{-}1), p) = \beta$ and $allo(d_2, m, I(j), p) = \alpha$, where $1 < j \leq |I|$, $\alpha + \beta = 1$, and $\alpha = 0$ or 1.

4.2. Team-level constraints

In most software development organizations, there is seldom a one-to-one mapping between developers and development tasks [7]. It means that software development is performed at a team level. To deal with characteristics related to team composition, we identified the following constraints:

- Constraint on sharing developers

Software development is performed by a set of teams. Each team is managed independently by a supervisor, and the members need to be unified into a working group acting as a unit [4]. In addition, the communication cost between teams is high [9]. Hence, developers are basically not shared between teams in practice, except for some tasks such as acceptance testing that need cooperation.

We describe the situation of team composition in the example in Table IV and compare the allocation policies for dealing with this situation. In Table IV, a percentage in parentheses represents the participation rate of a developer. In the example, Modules 1 and 2 are to be developed in C++ and provide visualization functionality. In other words, Modules 1 and 2 have the same module profile. On the other hand, Modules 3 and 4 are to be developed in Java and provide data processing functionality. We assumed that the number of developers is limited, thus some developers are allocated to two modules by providing two slots for each developer for all phases. Productivity of developers in this example is described in Table V.

In Allocation 1, Modules 1 and 2 share developers, and one additional developer is involved according to the workload difference. This situation is acceptable because sharing is limited to Modules 1 and 2, and the two modules have similar characteristics with the same module profile. Thus, they can be regarded as one team which develops similar modules. This is also applied to Modules 3 and 4.

DOI: 10.1002/spe

Table V. Productivity matrix for Table IV.

Developer	C++ and visualization	Java and data processing
A	1.3	1.0
B	1.2	1.1
C	0.8	0.8
D	0.9	1.3
E	0.9	0.9
F	1.0	1.2

Table VI. Comparison of allocation policies regarding the number of developers.

	Allocation 1		Allocation 2	
Module (Workload)	Developers	Duration	Developers	Duration
Module 1 (8)	A (100%), B (100%)	3.07	A (50%), B (50%) C (50%), D (50%)	3.47
Module 2 (11)	C (100%), D (100%) E (100%)	3.66	A (50%), B (50%), C (50%) D (50%), E (100%)	3.33
Total		3.66		3.47

In contrast, Developers C and E are shared between the two teams in Allocation 2. In this type of allocation, team scheduling is harder to manage because each team has to consider developers who are under the control of other teams. For example, if additional requirements are given, it is harder to adjust the workload of developers because other teams may need to deal with the unexpected workload of the shared developers. In addition, commitment problems may occur when each team requires high commitment from the shared developers. Thus, these can be overhead to the project and may increase the project duration. To prevent this situation, we propose the constraint on sharing developers. Given a developer d, an increment i, and a phase p, this constraint is described as follows:

- there exist no modules $m1$ and $m2$ such that $allo(d, m1, i, p) = 1$ and $allo(d, m2, i, p) = 1$, where $m1$ and $m2$ are not in the same module group.

- Constraint on the number of developers

As the number of developers increases, the efficiency of development gets lowered according to the communication cost [5–8]. To prevent this situation, an appropriate number of developers must be allocated to each module.

Generally, approaches using optimization algorithms can allocate an appropriate number of developers to each module when one developer is allowed to be allocated to one module only. However, if we allow allocating a developer to more than one module, some counterexamples can be shown. Consider the example in Table VI in which productivity of Developers A and B is set to 1.3 and productivity for other developers is set to 1.0.

Modules have at most three developers in Allocation 1, whereas modules have four to five developers in Allocation 2. Although Allocation 2 has a shorter estimated duration, as shown in Table VI, it may not be optimal as it increases the communication cost. Thus, the number of developers for each module needs be controlled in the human resource allocation procedure. To prevent the situation in Allocation 2, we propose the constraint on the number of developers. Given a set of developers D, a module m, an increment i, and a phase p, this constraint is described as follows:

- $\sum_{k=1}^{|D|} allo(D(k), m, i, p) \leq App\_Num(m, i, p)$,
  where $App\_Num(m, i, p)$ is the appropriate number of developers for m in p of i.

DOI: 10.1002/spe

Table VII. Comparison of allocation policies regarding novice teams.

		Allocation 1		Allocation 2	
Team	Module (Workload)	Developers	Duration	Developers	Duration
Team 1	Module 1 (10)	A, C, D	3.70	A, B, C	3.03
	Module 2 (5)	E, F	3.57	D, E	3.57
	Subtotal		3.70		3.57
Team 2	Module 3 (7)	B, G	3.50	F, G, H	3.33
	Module 4 (5)	H, I, J	2.38	I, J	3.57
	Subtotal		3.50		3.57
	Total		3.70		3.57

The appropriate number of developers for a module is decided according to the project environment. A guideline for setting the number is explained in Section 5, which describes the implementation of our allocation approach.

- Constraint on novice teams

A major role of a manager in a software development team is to make developers do the correct work and get the tasks done in the most efficient and least problematic way [10]. Thus, it is important to allocate an expert developer to each team to lead the development and guide other developers. If a team consists of novice developers only, the quality of the product cannot be guaranteed and the total duration of the project may increase because of a considerable amount of rework. We describe the situation related to allocating experts in Table VII and compare the allocation policies to deal with this situation. In this example, Developers A and B are experts and have a productivity of 1.3. Others are novices and have a productivity of 0.7.

In Allocation 1, each team has one expert who leads the development. In contrast, Team 2 in Allocation 2 consists of novices only. Although Allocation 2 requires a shorter estimated duration from the calculation, modules developed by Team 2 may have low quality and cause a considerable amount of rework. These can be overhead to the project and may increase the project duration. Thus, we propose the constraint on novice teams to prevent this situation. Given a module group mg, an increment i, and a phase p, this constraint is described as follows:

- $Exp\_Num(mg, i, p) \geq 1$, where $Exp\_Num(mg, i, p)$ is the number of expert developers allocated to any module of mg in p of i.

For the use of this constraint, our approach additionally manages the rank information of developers. For simplicity, our approach deals with two levels of ranks: *novice* and *expert*. The details are explained further in the following section.

5. CONSTRAINT-BASED HUMAN RESOURCE ALLOCATION APPROACH

In this section, we describe the constraint-based human resource allocation approach. The entire procedure is illustrated in Figure 2.

As inputs, developers and modules are used according to the structure proposed in Section 3. By the constraint on sharing developers, modules are subject to be grouped into module groups to compose teams, as explained in the previous section. In addition, rank information is given to developers to handle the constraint on novice teams.

Based on these inputs, our approach optimizes the allocation with the proposed constraints in the schedule perspective using ASA. As our approach supports incremental development, the allocation is performed for each increment, and the allocation result from a previous increment is given to the next increment as input.

The allocation approach in an increment consists of two phases: team allocation and individual allocation. In the team allocation, developers are grouped as a team by allocating them to a module

DOI: 10.1002/spe

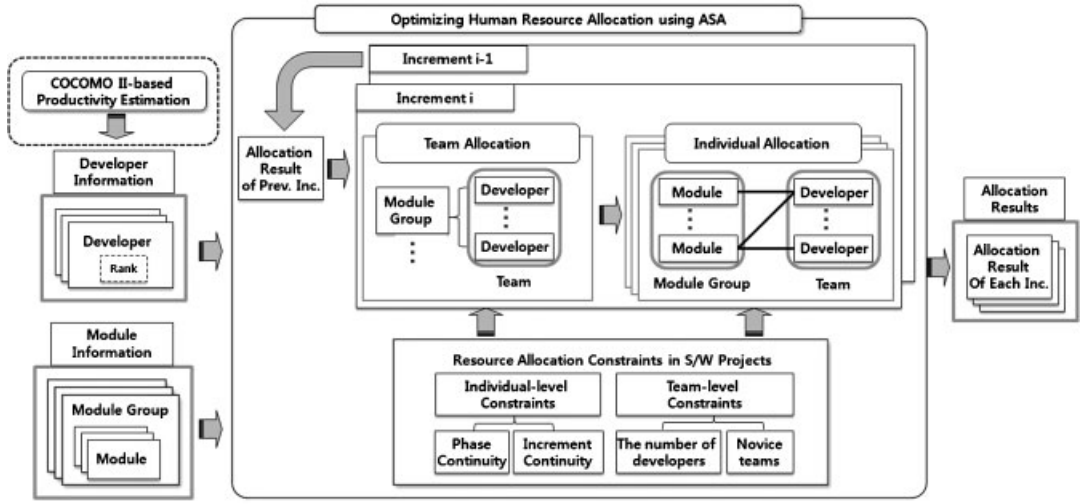

Figure 2. Overview of the human resource allocation procedure.

group. In the individual allocation, developers are allocated to modules in a module group for each development phase. In addition, a developer may be allocated to one or more modules using slots.

We propose the two-phase procedure for the following reasons. First, this two-phase procedure naturally prevents violation of the constraint on sharing developers, because developers are allocated fully to each team in team allocation. Thus, no developer can be allocated across teams. Second, the speed of optimization can be accelerated through this two-phase procedure. In team allocation, the unit of allocation is a module and a developer, rather than a phase and a slot. Thus, the convergence speed of allocation can be improved upon by reducing the search space for team level. Detailed allocation is handled in individual allocation, but the size of the problem is now smaller than that of the original problem. As this type of the allocation is known as NP-hard [22], partitioning the problem can improve the speed of optimization.

One of the main difficulties in using human resource allocation approaches lies in the lack of productivity data. To obtain productivity data, we need to establish a scheme to measure productivity and accumulate a large dataset for the more accurate evaluation of productivity. However, productivity data may not be always available, as measurement process is immature in many organizations [23]. When unavailable, productivity data need to be estimated, but estimation is prone to bias and is time-consuming as the number of developers increases. Thus, we provide a guideline for estimating productivity using the COCOMO II model. This is explained in the following subsection. Then we present the constraint-based optimization method.

5.1. A guideline to estimate productivity

To relieve the difficulty of obtaining productivity data, we propose a guideline to estimate productivity by using an existing estimation model, COCOMO II [13]. As COCOMO II provides various factors related to developers' capabilities in its cost drivers, it can be a valuable tool for estimating the productivity of developers. A number of studies have used the COCOMO II model for productivity estimation [24–28], but none has provided individual-level productivity estimation. Thus, we devised an individual-level productivity estimation guideline using COCOMO II. Basically, the values of the cost drivers in COCOMO II are supposed to be measured at the team level. In our study, we extended the use of the cost drivers to individual-level productivity estimation by assuming that the productivity of a team is proportional to the sum of the productivity of the team members.

As a developer has different capabilities for roles and types of modules, our guideline provides productivity estimation according to roles and module profiles based on COCOMO II. At first, we extract cost drivers related to developers' productivity from COCOMO II and categorize them into

DOI: 10.1002/spe

Table VIII. Description of productivity-related cost drivers of COCOMO II.

	Rating levels	Very low	Low	Nominal	High	Very high
ACAP	Descriptors	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile
	Effort multipliers	1.42	1.19	1.00	0.85	0.71
PCAP	Descriptors	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile
	Effort multipliers	1.34	1.15	1.00	0.88	0.76
APEX	Descriptors	≤2 months	6 months	1 year	3 years	6 years
	Effort multipliers	1.22	1.10	1.00	0.88	0.81
PLEX	Descriptors	≤2 months	6 months	1 year	3 years	6 years
	Effort multipliers	1.19	1.09	1.00	0.91	0.85
LTEX	Descriptors	≤2 months	6 months	1 year	3 years	6 years
	Effort multipliers	1.20	1.09	1.00	0.91	0.84

role-related factors and module-related factors. Then, productivity according to roles and modules can be calculated using the values of these factors. The details of productivity estimation are described as follows.

5.1.1. Extraction and categorization of productivity-related cost drivers of COCOMO II. For productivity estimation, we have chosen five cost drivers from the personnel factors of COCOMO II. Each of them is described as follows [13]:

- *Analyst capability (ACAP)*: This measures analyst capability, considering analysis and design ability, thoroughness, and the ability to communicate and cooperate. This rating should consider the capability of analysts, rather than experience, which is rated with APEX, LTEX, and PLEX.
- *Programmer capability (PCAP)*: This measures programmer capability, considering programming ability, efficiency and thoroughness, and the ability to communicate and cooperate. This rating should consider the capability of programmers, rather than experience, which is rated with APEX, LTEX, and PLEX.
- *Application experience (APEX)*: This measures the level of applications experience developing the software system or subsystem. The ratings are defined in terms of the equivalent level of experience with this type of application.
- *Platform experience (PLEX)*: This measures the productivity influence of platform experience by recognizing the importance of understanding the use of more powerful platforms, including more graphic user interface, database, networking, and distributed middleware capabilities.
- *Language and tool experience (LTEX)*: This measures the level of experience regarding programming language and computer-aided software engineering (CASE) tools used in the development.

The values related to productivity for these cost drivers are described in Table VIII. A value (Effort Multiplier) represents the effect of the cost driver on the effort. For example, if the ACAP value is low, then the effort is estimated to increase by 19%.

To obtain the productivity for each role and module profile, we categorized the cost drivers of COCOMO II into two groups: role-related factors and module-related factors. Role-related factors provide the productivity influence of performing phases, such as design and implementation. Module-related factors provide the productivity influence according to the developers' experience about a specific type of module. The categorization is shown in Table IX.

APEX can be used for all phases as it deals with the experience of an application type and its effect cannot be divided for each phase. LTEX is also used for all phases as it measures CASE tool experience for all phases as well as language experience. PLEX affects phases from design because requirement analysis is not affected much by platforms. ACAP is used for requirement

DOI: 10.1002/spe

Table IX. Categorization of COCOMO II personnel factors related to developers' productivity.

Factors	Category	Affected phases
ACAP	Role	Requirement analysis, design
PCAP	Role	Implementation, testing
APEX	Module	All phases
PLEX	Role/module	Design, implementation, testing
LTEX	Module	All phases

analysis and design phases according to its definition. In the case of PCAP, we assume that PCAP can be used for the testing phase as well as the implementation phase, as COCOMO II does not provide the effect of allocating testers and programmers are involved in the testing phase in many organizations.

When using this productivity estimation guideline, the module-related factors form module profiles. Thus, application types of APEX, platforms of PLEX, and languages and tools of LTEX can be the characteristics for describing module profiles.

5.1.2. Calculating productivity for roles and module profiles. According to the categorization of cost drivers in Table IX, we can calculate developers' productivity to each role and module profile. Basically, productivity is calculated from an inverse of the product of related effort multipliers of the cost drivers because productivity is in inverse proportion to the required effort. This is described in the following equations:

Given a developer d and a module profile p,

$$prod(d, Analyst, p) = 1/(\text{APEX}(d, p) \times \text{LTEX}(d, p) \times \text{ACAP}(d)),$$

$$prod(d, Designer, p) = 1/(\text{APEX}(d, p) \times \text{LTEX}(d, p) \times \text{PLEX}(d, p) \times \text{ACAP}(d)),$$

$$prod(d, Programmer, p) = 1/(\text{APEX}(d, p) \times \text{LTEX}(d, p) \times \text{PLEX}(d, p) \times \text{PCAP}(d)),$$

$$prod(d, Tester, p) = 1/(\text{APEX}(d, p) \times \text{LTEX}(d, p) \times \text{PLEX}(d, p) \times \text{PCAP}(d)).$$

The definition of $prod$ is the same as explained in Section 3. As shown in the equations, values of module-related factors need to be obtained for each developer by the number of module profiles. For example, if some modules are supposed to be developed using Java, and other modules are developed in C++, two sets of data need to be gathered for each developer.

In the equations above, values of role-related factors need to be adjusted because effort multipliers of COCOMO II drivers are supposed to estimate the influence on all phases. For instance, PCAP measures the capability of programmers which is related to the implementation and testing phases, but the PCAP values express the influence of programmers on the entire effort. As PCAP values are applied only to two roles in the equation, the effect of PCAP is decreased by the percentage of the implementation and testing phases over the entire effort.

To help in a better understanding of this situation, suppose that the entire development requires 100 man-months and the implementation and testing phases require a 50% effort in the entire development. In estimating the productivity of developers of PCAP value low, if we use the original value of PCAP, then it would be 0.87 from dividing 1 by 1.15. In this case, the total duration is calculated to be 107.5 from $(100 \times 50\% \div 0.87 + 100 \times (1 - 50\%))$. However, this result conflicts with the calculation using the definition of PCAP because the total duration with low PCAP is calculated to be 115 by multiplying 100 with 1.15.

Thus, values of these factors need to be adjusted to maintain their effect on the entire development by considering effort distribution of phases related to the roles. Given a role-related factor R, this is described in the following equation:

$$AdjustedValue(R) = \frac{originalValue(R) - (1 - \sum related\_phase\_distribution(R))}{\sum related\_phase\_distribution(R)}.$$

DOI: 10.1002/spe

In the equation, $AdjustedValue(R)$ is a corrected value of a role-related factor R. $Related\_phase\_distribution(R)$ represents the percentage of effort taken by phases related to R over the entire effort.

Using the equation above, PCAP values were adjusted from (Very Low 1.34, Low 1.15, Nominal 1.0, High 0.88, Very High 0.76) to (Very Low 1.68, Low 1.30, Nominal 1.0, High 0.76, Very High 0.52) in the previous example. In this case, the total estimated effort was maintained the same: 115 man-months from the calculation $(100 \times 50\% \times 1.30 + 100 \times (1 - 50\%))$. As shown in the example, we can obtain the influence of role-related cost drivers in estimating individual productivity while preserving their influence on all phases using the proposed equations.

5.2. Constraint-based optimization using ASA

We propose the two-phase procedure to optimize scheduling of human resource allocation using ASA while preserving the constraints mentioned above.

There are two styles to deal with constraints using SA or ASA. The first allows perturbation of solutions only in the feasible solution space. In this case, we can obtain feasible solutions all the time. However, it makes the perturbing procedure more complex and takes more time in perturbation as the new solution is generated by considering all constraints. Moreover, this can be used for hard constraints only, as this removes the possibility of violating constraints.

The other style allows infeasible solutions in perturbation and uses some methods to filter these out. In this case, solutions can be generated in a relatively short time, and the possibility of staying in a local optimum can be reduced. Therefore, many studies have allowed searching in the infeasible solution space and have offered penalties when constraints are violated [17].

In human resource allocation of software projects, violation of the constraints causes additional development overhead, rather than prohibition of development. Thus, one may sacrifice the constraints to some extent for reducing the estimated duration. With this in mind, our approach allows some violations of the constraints in perturbation and gives penalties in the cost function when the constraints are violated.

We devised penalty functions from discussions with experts. As experts do not use those penalty functions in practice, we have tried to extract reasonable penalty functions based on the constraints and confirmed with the experts that they are acceptable in practice.

To apply SA or ASA to real-world problems, we need to adjust its generic functions to the specific problem domain. This section explains the design of three main functions—initialize, perturb, and cost functions—and how constraints are applied in the SA-based optimization approach for team allocation and individual allocation.

5.2.1. Team allocation. In the team allocation, developers are grouped into teams by allocating them to each module group, which is a unit of team allocation. To group developers in a team, all slots of a developer are allocated to one module group. This needs to be reflected in the perturb function. For the cost function, we need to revise the definition of cost according to the module groups. In addition, the penalty for violating constraints is to be considered in the cost function.

In the following, we describe the design of these two functions as well as the initialize function.

- Initialize function for team allocation

The initialize function decides the parameters of ASA algorithm and an initial solution. Parameters such as the number of loops, the cooling factor, and the initial temperature are usually determined by repeating execution of the algorithm. When these parameters are set to high values, the quality of solutions improves, but generating solutions takes more time. Thus, we need to choose reasonable values for the parameters from the initial experiments.

An initial solution should be carefully decided upon because it largely affects the quality of solutions and convergence time [29]. Thus, we devised a greedy heuristic for deciding the initial allocation. Basically, the heuristic sorts the developers and module groups according to average productivity and workload, respectively, and allocates the developer with the highest productivity

DOI: 10.1002/spe

to the module group with the highest workload to minimize the total cost. In addition, it is guided to conform to the constraints in the allocation procedure.

The heuristic consists of two parts to deal with the constraint on increment-level continuity. At first, we present the allocation heuristic for the first increment in the team allocation. It consists of three steps:

- *Step 1*: Repeat allocating the expert with the highest productivity to the module group with the highest workload until every module group has an expert.
- *Step 2*: Repeat allocating the remaining expert with the highest productivity to the module group of the longest duration.
- *Step 3*: Repeat allocating the remaining novice with the highest productivity to the module group of the longest duration.

According to the constraint on novice teams, experts are allocated first. In Step 1, each module group is guaranteed to have an expert. In our approach, it is assumed that the number of experts is equal to or higher than the number of module groups to satisfy the constraint on novice teams. After allocating one expert to each module group, the remaining experts are allocated according to the duration of module groups in Step 2. In each allocation of a remaining expert, the duration for each module group is updated, and the total cost is reduced by allocating more developers to the module group with the longest duration. Finally, novice developers are allocated in the same way as in Step 2.

Next, the allocation heuristic for the second or later increments in the team allocation consists of six steps:

- *Step 1*: Repeat allocating developers to module groups to which they were allocated in the previous increment.
- *Step 2*: Repeat allocating the remaining expert with the highest productivity to the module group with the highest workload which has no expert yet. This is repeated until every module group has an expert or there is no expert available.
- *Step 3*: If there is any module group which does not have any expert yet, move one expert from a module group which already has more than one expert to the module group having no expert. This is repeated until every module group has at least one expert.
- *Step 4*: If there are remaining experts after conducting Step 2, repeat allocating the remaining expert with the highest productivity to the module group of the longest duration.
- *Step 5*: Repeat allocating the remaining novice with the highest productivity to the module group of the longest duration.
- *Step 6*: Improve the cost of allocation by adjusting allocated developers.

According to the constraint on increment-level continuity, allocation in the previous increment is reflected in Step 1. There may exist new module groups introduced in this increment, or there may exist developers allocated to the module groups which are completed in the previous increment. Those developers and module groups are handled from Steps 2 to 5. After allocating all the developers, the duration of a module group may be considerably different from that of other module groups according to the workload difference of module groups over increments. For example, suppose that a module group has high workload in Increment 1 and low workload in Increment 2. Then, a large number of developers will be allocated to the module group in Increment 1 and also Increment 2 by Step 1. Thus, the algorithm tries to make the distribution of duration of module groups as uniform as possible to reduce the entire cost in Step 6. For this purpose, the module group with the minimal effect of removing a developer sends one developer to the module group with the longest duration, unless that increases the total cost. Step 6 is repeated until no further improvement is possible.

- Perturb function for team allocation

For team allocation, two kinds of perturbing operations are used here with equal probability: *exchange* and *move*.

DOI: 10.1002/spe

Exchange operation exchanges developers between module groups. In the procedure, two developers are chosen randomly. If module groups of the two developers are different, then the developers are exchanged between the two module groups.

Move operation sends a developer to another module group. At first, a developer is chosen randomly. Then a new module group to be allocated is chosen randomly. If the module group which currently has the developer and the new module group are different, then the developer is allocated to the new module group.

- Cost function for team allocation

The cost function for team allocation is defined as the maximum duration among all module groups. The duration of a module group is estimated as the sum of duration of all the modules in the module group and penalty from violating the constraints. Given a set of module groups MG and an increment i, the following equations describe the cost function in the team allocation:

$$Cost(i) = Max(Duration(MG(1), i), \ldots, Duration(MG(n), i)) \text{ where } n = |MG|,$$

$$Duration(MG(j), i) = \sum_{k=1}^{|M|} Duration(M(k), i) + Penalty(MG(j), i) \quad \text{where } M(k) \in MG(j).$$

Duration of a module is the same as defined in Section 3, and penalty of a module group is decided as the sum of penalty for the constraints applied in the team allocation.

When calculating the duration of a module group, we assume that all slots of the developers in the module group are allocated to each module. In other words, we handle the workload of all modules in the module group as a whole for each phase.

For team allocation, the constraints on increment-level continuity and novice teams are considered. The constraint on phase-level continuity is not violated in this phase because each developer is fully allocated to all modules in a module group, and the constraint on sharing developers is handled by two-level optimization itself. In the case of the constraint on the number of developers, it is naturally conformed to by the team allocation because the appropriate number of developers is assigned to each module group by optimization. In other words, if too many developers are allocated to one module group, the estimated duration for other module groups may increase because it implies that less developers are allocated to other module groups.

Note that the team allocation result is not the final result but the input of individual allocation. Thus, the penalty in team allocation is not used for the final results, as they are considered again in detail in individual allocation.

The usages of penalty for the two constraints are explained as follows:

- Constraint on increment-level continuity

When this constraint is violated in a module group mg in an increment i, a penalty is given in proportion to the duration of the module group and the minimum number of added ($\#Of\,Added\,Devs$) and removed ($\#Of\,Removed\,Devs$) developers. This is described as

$$Increment\_Penalty(mg, i) = Duration(mg, i) \times Increment\_Penalty\_Constant \times \frac{Min(\#Of\,Removed\,Devs(mg, i), \#Of\,Added\,Devs(mg, i))}{\#Of\,Developers(mg, i)}.$$

The *Increment_Penalty_Constant* is determined according to the organizational policy or the project environment.

- Constraint on novice teams

When this constraint is violated in a module group mg in an increment i, a penalty is given in proportion to the estimated duration for the module group. This is described in the following equation:

$$Novice\_Penalty(mg, i) = Duration(mg, i) \times Novice\_Penalty\_Constant.$$

DOI: 10.1002/spe

The *Novice_Penalty_Constant* is determined according to the organizational policy or the project environment. Note that the equation above is calculated only if violation occurs. Otherwise, it is set to zero. The same rule is applied to other constraints.

5.2.2. Individual allocation. Using the allocation result from the team-level allocation, developers of a team are allocated to modules in the module group for each development phase. Thus, individual allocation is conducted for each module group. In this allocation, developers may be allocated to multiple modules using slots, and participation rates for slots should be controlled for optimization. In addition, the constraints on phase-level continuity and the number of developers need to be considered. This section describes the design of perturb and cost functions supporting these characteristics of the individual allocation as well as the initialize function.

- Initialize function for the individual allocation

In deciding an initial solution for the individual allocation, most parts of the heuristic algorithm is the same as that of the team allocation. The main difference is in dealing with the constraint on novice teams. As this constraint is applied in team allocation only, our approach does not consider ranks of developers in individual allocation. In addition, if the number of developers in a team is not enough to allocate each of them to only one module, each developer is allocated to multiple modules using slots, and the equal participation rates are given to slots for each phase.

- Perturb function for individual allocation

For individual allocation, three kinds of perturbing operations are used here with equal probability: *exchange*, *move*, and *change participation rate*. Exchange and move operations in individual allocation are similar to those in team allocation, but they are applied to slots of developers. Thus, these operations additionally choose a phase and slots in the phase randomly as well as developers and modules in the individual allocation.

In addition to exchange and move operations, we additionally devised the change participation rate operation for the individual allocation. This operation chooses two slots of a developer in a phase, then adjusts the participation rates between the slots randomly while maintaining the sum of participation rates to 100%. For example, if a developer has two slots in a phase which have 50% participation rate each, then they can be adjusted to 60 and 40%, or 30 and 70%.

If the participation rate for a slot is too small, it would not be practical. For example, if a developer is allocated to a module with a participation rate of 1%, it may only cause communication overhead. Therefore, our approach sets the lower bound of participation rates to prevent this situation. This value needs to be determined according to the organizational policy or the project environment.

- Cost function for individual allocation

In individual allocation, the cost function for a module group is refined as the maximum duration among all modules. The duration of a module is estimated as the sum of duration and penalty from violating the constraints for all phases. Given a module group MG, an increment i, and a set of phases P, the following equations describe the cost function in the individual allocation:

$$Cost(i) = Max(Cost(MG(1), i), \ldots, Cost(MG(n), i)) \quad \text{where } n = |MG|,$$

$$Cost(MG(j), i) = Max(Duration(M(1), i), \ldots, Duration(M(m), i))$$
$$\text{where } M(1), \ldots, M(m) \in MG(j) \quad \text{and} \quad m = |MG(j)|,$$

$$Duration(M(k), i) = \sum_{l=1}^{|P|} (Duration(M(k), i, P(l)) + Penalty(M(k), i, P(l))).$$

Duration for a phase is the same as defined in Section 3. Penalty of a phase in developing a module is decided as the sum of penalty for the constraints applied in individual allocation.

DOI: 10.1002/spe

For individual allocation, all constraints except the constraint on sharing developers are reflected in the cost function. The constraints applied in individual allocation are described as follows:

- Constraint on phase-level continuity

When this constraint is violated in a phase of developing a module, a penalty is given in proportion to the ratio of violating developers over the number of allocated developers. Given a module m, an increment i, and a phase p, this is described as follows:

$$Phase\_Penalty(m,i,p) = Duration(m,i,p) \times Phase\_Penalty\_Constant \times \frac{\#Of\,Violating\,Developers(m,i,p)}{\#Of\,Developers(m,i,p)}.$$

The *Phase_Penalty_Constant* is determined according to the organizational policy or the project environment.

- Constraint on increment-level continuity

In applying this constraint in individual allocation, we considered the situation when some developers are moved to other teams in the second or later increments in team-level allocation. In this case, other team members may help in the development of the module that the moved developers were involved in. This is not allowed in the original form of the constraint as developers are added and removed simultaneously. However, this could be acceptable in practice since team members work together in developing modules in a module group which have characteristics similar to each other. Thus, our approach allows moving developers within a team, while preventing moving developers to other teams and adding developers from outside the team simultaneously. Given a module m, an increment i, and a phase p, this constraint is applied as follows:

$$Increment\_Penalty(m,i,p) = Duration(m,i,p) \times Increment\_Penalty\_Constant \times \frac{Min(\#Of\,RemovedDevs\_OT(m,i,p), \#Of\,AddedDevs\_OT(m,i,p))}{\#Of\,Developers(m,i,p)}.$$

$\#Of\,RemovedDevs\_OT$ and $\#Of\,AddedDevs\_OT$ represent the numbers of moved developers to other teams and added developers from other teams, respectively.

The *Increment_Penalty_Constant* is the same as described for team allocation.

- Constraint on the number of developers

To determine whether this constraint is violated, the appropriate number of developers first needs to be determined for each module. In our approach, it is determined in proportion to the ratio of the workload of the module over the workload of the module group. As the individual allocation deals with phase level, the appropriate number of developers for a module is decided for each phase. Given a module m, a module group mg which m belongs to, an increment i, and a phase p, this is described as follows:

$$App\_Num(m,i,p) = \left\lceil (1+buffer) \times \frac{wload(m,i,p)}{wload(mg,i,p)} \times \#of\,Developers(mg,i,p) \right\rceil.$$

In the equation, App_Num represents the appropriate number of developers.

We introduced an adjustment factor *buffer* in the equation because the number of developers for a phase could be increased according to the effect of cooperation. For instance, if there are 10 developers and 10 modules with identical workload, a module should only be developed by one developer without buffer, and cooperation is prohibited. The buffer value is determined according to the project environment. We suggest that if the number of developers is much larger than the

DOI: 10.1002/spe

number of modules, one should set the buffer to zero or a sufficiently small value close to zero because it is then possible to maintain independent developers for each module.

Based on the appropriate number for a module, we can determine whether this constraint is violated. If the number of allocated developers to a module is larger than the appropriate number, a penalty is given to the module. The penalty is in proportion to the number of excessive developers. Given a module m, an increment i, and a phase p, this is described as follows:

$$Dev\_Num\_Penalty(m, i, p) = Duration(m, i, p) \times Dev\_Num\_Penalty\_Constant \times (\#of\ Developers(m, i, p) - App\_Num(m, i, p)).$$

The equation above is triggered only if violation occurs. Thus, there is no situation where the penalty has a negative value. The *Dev_Num_Penalty_Constant* is determined according to the organizational policy or the project environment.

- Constraint on novice teams

As this constraint is applied at the team level, the penalty value is calculated if and only if it is violated in team allocation. Given a module m, an increment i, and a phase p, in the case of violated team allocation, the penalty value is obtained as follows:

$$Novice\_Penalty(m, i, p) = Duration(m, i, p) \times Novice\_Penalty\_Constant.$$

The *Novice_Penalty_Constant* is the same as described for the team allocation.

6. CASE STUDY

We performed two kinds of experiments in this case study. The first experiment was conducted to show the effect of different penalty settings. If the penalty constants of the constraints are set to zero, the constraints are to be violated without any restriction. This may show the effect of other approaches which do not consider the constraints introduced in this paper. If penalty is set very high, we can observe any possible conflict between the constraints.

The second experiment was conducted to validate the effectiveness of our approach in the schedule optimization. For the comparison, we used two benchmarks: the actual duration of a project and the result from the heuristic used in the initialize functions.

6.1. Data collection

For this case study, the project data from the development of a government information system (over 10 000 function points) were used. The system was developed using the incremental development method, with increments by two. The system consists of 24 modules; 17 modules are assigned to the first increment while 20 modules are assigned to the second increment. It means that some modules are completed within an increment, whereas others are incrementally developed. The modules are grouped into nine module groups by considering the functional relationships, size of modules, and the actual allocation results in the project. The average number of developers was 39 (29 novices and 10 experts). Java or C++ was used for each module depending on the module characteristics. Each developer participated in all the development phases of this project.

To gather productivity data, we carried out a survey of the five COCOMO II cost drivers, and information about 22 of the 39 developers (16 novices and 6 experts) was retrieved. For the experiments, we randomly replicated the collected information for the remaining 17 developers.

To obtain workload from the effort for each phase, we normalized the effort by the average productivity of developers. As the effort is estimated by considering developers' productivity, the effect of productivity can be applied in a duplicated manner without normalization.

In addition, we gathered the information about planned duration and actual duration of the project to compare them with the result from our approach in the second experiment.

DOI: 10.1002/spe

6.2. Experimental environment

The parameters used in the experiments are described as follows:

- Penalty constants and other parameters: These values are set after discussion with the experts who participated in the project.
  - *Phase_Penalty_Constant*: a maximum value, as the violation was not permitted due to the organizational policy
  - *Increment_Penalty_Constant*: 50%
  - *Dev_Num_Penalty_Constant*: 10%
  - *Novice_Penalty_Constant*: a maximum value, as the violation was not permitted due to the organizational policy
  - The number of slots for each developer: 2 for each phase
  - The lower bound of a participation rate: 20%
  - *Buffer* in determining the appropriate number of developers: 30%
- Parameters of ASA: These values are set after conducting initial experiments.
  - T(the initial temperature): 100
  - L(the number of internal loops): 500
  - M(the external loop control parameter): 8
  - N(the internal loop control parameter): 2000
  - α(the cooling factor): 0.95.

The algorithm was implemented in Java, and all computations were made on a Pentium IV 2.66 GHz computer.

6.3. Experiment 1: effect on different penalty settings for the constraints

6.3.1. Experimental design. In this experiment, we conducted the allocations with three different penalty settings: expert-given penalty, no penalty, and maximum penalty. First, we performed allocation using the penalty constants given by experts. This can produce the feasible allocation result regarding the project environment.

In the allocation with no penalty, all penalty constants are set to zero. In this case, the allocation result is produced without considering the overhead of violating constraints because no penalty is given when the constraints are violated. In addition, we removed the consideration of the constraints in the initialize functions. Thus, the result of this case represents the allocation approaches which do not consider the human-related constraints proposed in this paper.

In the allocation with maximum penalty, all penalty constants are set to a maximum value. In this case, the allocation is guided to conform to the constraints in a very strict manner. Thus, we can observe the effect when the constraints are used as hard constraints which must be conformed to. In addition, we can figure out whether there are conflicts between constraints. If any, the result may become infeasible since the maximum penalty drastically increases the cost.

We obtained 30 allocation results for each penalty setting to prevent extreme case representation.

6.3.2. Results. The average of 30 allocations for each penalty setting is summarized in Table X. The table shows the estimated duration in months and the percentage of violation for each constraint from the individual allocation, which produces the final result. In the table, P, I, D, and N in the second row represent constraints on phase-level continuity, increment-level continuity, the number of developers, and novice teams, respectively. As previously mentioned, the constraint on sharing developers is handled by the two-phase approach itself; thus, it is not shown in the table.

For the constraint on phase-level continuity, the percentages are calculated for developers (i.e. the average percentage of violating phases of all developers). For the constraint on novice teams, the percentages are calculated at the module group level (i.e. the average percentage of violating module groups of all module groups) as they are applied in team allocation. For other constraints, the percentages are calculated at the module level (i.e. the average percentage of violating phases of all modules).

DOI: 10.1002/spe

Table X. Comparison of various penalty settings.

Condition	Inc.	Duration (Mth)	Violation (%)			
			P	I	D	N
No_ penalty	Inc.1	11.51	28.83	0.0	17.10	21.85
	Inc.2	10.66	50.67	61.46	28.25	45.19
Expert_ penalty	Inc.1	11.86	0.0	0.0	0.79	0.0
	Inc.2	11.48	0.0	15.83	3.67	0.0
Maximum_ penalty	Inc.1	12.05	0.0	0.0	0.0	0.0
	Inc.2	11.75	0.0	0.0	0.0	0.0

P: phase-level continuity/I: increment-level continuity.
D: the number of developers/N: novice teams.

In Table X, violation of the increment-level continuity constraint is zero for the first increment since the constraint is only applied to the second or later increments.

In the result, the allocations with no penalty have the highest degree of violation for all constraints. Violations occurred more frequently in the second increment as the project has more modules in this increment. In the schedule perspective, the estimated duration was minimized in comparison with those of other conditions because of no restrictions on the constraints. Although the allocations with no penalty show the best schedule among the three conditions, it is hard to use them in real project environments due to high violation rates for the constraints. In order to illustrate this point, we applied the penalty constants given by experts to the allocations with the no penalty condition. As the expert-given penalty constants represent the effect of the constraints on the schedule of the project used in this case study, the possible effect of giving no penalty can be shown by applying the expert-given penalty constants. As a result, the allocations were infeasible in that they violated the constraints on phase-level continuity and novice teams which were given maximum penalty. When we relaxed these constraints by setting the penalty constants to 50%, which was the same as *Increment_Penalty_Constant*, the schedule for each increment was increased to 15.5 and 19.45 months on average, respectively. This implies that allocation approaches may produce impractical allocations if they focus on minimization of the total duration using productivity-based calculations only.

When penalty constants are given by experts, it is observed that some constraints were sacrificed for optimization, but the violations occurred less frequently than the allocations with no penalty. This is quite obvious, because the penalty plays a role to guide the allocations to conform to the constraints. In the schedule perspective, the allocations have longer estimated duration than the allocations with no penalty. The difference is bigger in the second increment because the constraint on increment-level continuity has restrained the diversity of possible allocations. However, since the allocations were produced by considering the constraints, we can conclude that the allocations in this condition are more practical than those with no penalty.

In the case that a maximum value is given to all penalty constants, the allocations strictly conform to all the constraints. This implies that the constraints do not conflict with each other and the constraints can be used all together as hard constraints. In addition, the required duration is slightly longer than that of the second condition, but the difference is not very significant: increase by 0.46 months out of 23.34 months. Thus, when it is difficult to set appropriate penalty constants, we may use the constraints as hard constraints as the schedule of these allocations is reasonable in comparison to those with other conditions.

6.4. Experiment 2: effectiveness of the approach in a schedule perspective

6.4.1. Experimental design. In this experiment, we compared the result from the allocations with the expert-given penalty constants in the first experiment with the actual duration of the project and the result from the heuristic used in the initialize functions.

DOI: 10.1002/spe

Table XI. Comparison of the result from our approach, the result from the heuristic, and the actual duration.

		Our approach		
Increment	Actual (Mth)	Average (Mth)	Best (Mth)	Heuristic (Mth)
Increment 1	15	11.86	11.46	22.30
Increment 2	13	11.48	10.92	18.37
Total	28	23.34	22.38	40.67

The actual duration is used as a benchmark to judge the effectiveness of our approach in the real-world environment. The actual duration of the project was the same as the planned duration in the project.

The heuristic of the initialize function is used in this experiment as another benchmark for the comparison. As the allocation in planning was performed according to the project manager's personal experience, and the planned schedule was forced to be kept, an additional benchmark is needed to show the performance of the optimization. By comparing the allocation result from our approach with that from this heuristic, we can observe the degree of improvement in our approach. In addition, as the heuristic may represent the possible behavior of project managers, we can anticipate the performance of manual allocation using the heuristic result.

6.4.2. Results. Results are shown in Table XI. Duration for each case is described in months.

In the best case scenario of our approach, it is shown that the duration can be reduced by 5.62 months (out of 28 months) compared to the actual duration. For further analysis, we conducted a one-sample *t*-test to show that our approach consistently produced a more efficient schedule than the actual duration. The null hypothesis is that the average estimated duration of our approach is longer than the actual duration. At 1% level of significance, the null hypothesis is rejected as $p = 0.00$. In comparison with the result from the heuristic, it is observed that our approach has improved the schedule of the heuristic by about 45%. Thus, we can conclude that our approach is effective in reducing the project duration.

In the case of the heuristic, there was much difference in the estimated duration among modules, and this caused long duration. This shows the necessity of the optimization method in the allocation problems.

- Discussion

As the resource allocation problem has high computational complexity, it is important to generate efficient solutions in reasonable time. One of the goals in devising the two-phase procedure is to reduce convergence time by partitioning the problem into team and individual allocations. Thus, we conducted an additional experiment to compare the efficiency of computation. In the experiment, whole modules were categorized into one group; thus only individual allocation was conducted. As only one team was considered, the constraints on increment-level continuity, sharing developers, and novice teams were not applied. While it took 4.36 min on average to generate an allocation result in the original setting, 16.38 min were spent on average in the allocation with one module group. This result shows the efficiency of the two-phase procedure of this approach. The average schedule from the allocation with one module group was not better than that from the original setting: 23.83 for the two increments.

6.5. Threats to validity

One issue that needs to be addressed for this case study is the replication of developer information in data collection. As productivity information about all developers is not available, productivity information collected for 22 developers is randomly duplicated for the remaining 17 developers. Generally, the degree of optimization increases with an increasing number of developer profiles because the diversity of solutions in the optimization is increased. Thus, we suggest that our

DOI: 10.1002/spe

approach might have produced better solutions in the case study if we could have obtained the productivity of all the developers.

Another issue concerns the practical feasibility of the proposed solution. As additional constraints can be found in practice, there could be some argument about the practical usefulness of the proposed solution. We therefore discussed the practicality of the solutions with the experts participating in the project and are in agreement that the proposed solutions are practical.

7. RELATED WORK

Resource allocation is a traditional research area in operations research and management science. Many optimization techniques have been proposed, such as parallel machine scheduling, flow shop scheduling, and job shop scheduling, to solve specific resource allocation problems [30]. Among these techniques, unrelated parallel machine scheduling is the most similar to human resource allocation in software projects because it can handle developers performing tasks in parallel, with the productivity of each developer varying according to the task characteristics. Unrelated parallel machine scheduling is known as NP-hard [22], and a number of approximation approaches and heuristics have been developed [31–36]. However, directly applying these approaches to software projects is not feasible because these approaches mainly deal with machine allocation and do not take human-related characteristics into consideration.

In the field of software engineering, a number of approaches have been proposed to address human resource allocation in software projects. Chang *et al*. [20] proposed optimizing human resource allocation in software projects using genetic algorithm in order to minimize the total time span, labor costs, and overtime with any given software project task network and developers' skill set. This approach does not, however, consider the different productivity levels of developers. In addition, this approach may produce the case in which a developer's time is fragmented over too many tasks in allocation, which decreases the efficiency of project execution [7]. Their recent work [5] improved the project scheduling by breaking down development tasks into smaller time-sliced activities and allocating developers according to the time-line. This enables a more fine-grained optimization of allocation. In addition, productivity according to experience and the effect of learning in the projects are considered in the approach.

Alba and Chicano [21] proposed a similar approach to [20]. This approach optimizes the duration and cost of a software project using genetic algorithm, and considers the working overtime of developers and the skills required to perform tasks of a project. They performed various trade-off experiments by devising an instance generator. Similar to [20], the limitation of this approach is that it is not appropriate to deal with the different productivity levels of developers.

Duggan *et al*. [6] proposed optimizing the estimated duration and quality of software in the implementation phase of software development by using different productivity factors for each developer according to the required skills of each package, which is a unit of allocation. In addition, this approach supports various options for package precedence, full team utilization, and cross-communication overhead. In spite of these advantages, this approach also has some limitations. First, it does not cover the entire software development life cycle. This approach may not be appropriate if developers of previous phases are continuously involved in the implementation phase because problems related to the continuity of allocation may arise. Second, this method may produce a solution in which a developer's time is fragmented over too many packages in allocation, which increases the required effort.

Ngo-The *et al*. [19] provided an optimized human resource allocation approach for release planning. Their approach optimizes the allocation with the two-phase method which combines the strength of integer linear programming and genetic algorithm, hence the produced solutions are close to the optimal solution. It also considers the different productivity levels of developers according to the phases in allocation. However, this approach does not support collaboration as it assumes that a task is performed by one developer only, with each developer working on one task at a time. As it is frequently necessary to concurrently allocate developers to multiple tasks and to

DOI: 10.1002/spe

assign more than one individual to work cooperatively on a single task [7], this assumption may lower the method's practical usefulness. In addition, the method does not support the different productivity levels of developers with respect to feature characteristics, which is needed to obtain efficient solutions because developers' productivity may vary according to the required skills of each feature.

Barreto *et al*. [8] proposed a resource optimization approach which provides various trade-off analyses for the cost, team size, and completion time by considering the salaries, skills, and availability of developers. Similar to [19], it does not support collaboration among developers. In addition, difference in productivity levels according to module characteristics is not considered in this approach.

Xiao *et al*. [37] provided an approach to optimize multiple projects. It scores each project based on the cost and schedule and obtains the optimized allocation for all projects using genetic algorithm. This approach also does not support the different productivity levels as per the skills required for each of the projects.

One of the common limitations of these approaches is the lack of consideration of human-related characteristics in allocation. Some approaches deal with effect of learning [5] and communication overhead from the number of developers in a team [5, 6, 8], but other characteristics such as continuity of allocation, communication between different teams, and the role of team leaders are not considered in the approaches above. Another limitation is that these approaches are not validated with real project data, which makes demonstrating the effectiveness of optimization difficult.

There are other approaches for dealing with the human resource allocation problem of software development. Fenton *et al*. used Bayesian networks to help in resource decisions in software projects [38]. Padberg explained the influence of human resource allocation strategies with a system dynamics model [39], and proposed a probabilistic scheduling model for software projects [40]. Lee *et al*. suggested a resource management method using simulation modeling for multi-project resource allocation [41]. Antoniol *et al*. proposed an approach for estimating the staffing level for maintenance projects, using queuing theory [42]. For testing resource allocation, a number of approaches have been proposed to optimize the staffing level in the testing phase [43–47]. These approaches do not deal with allocation at the individual level, hence they are not appropriate to determine who performs which tasks.

8. CONCLUSION

Human resource allocation is one of the crucial factors for success of a software project as relevant allocation of human resources helps to maximize the efficiency of software development. However, human resource allocation in software projects is very complex and depends on the characteristics of developers. As these characteristics affect the efficiency of project execution, we consider them as constraints of human resource allocation in our approach. We identified individual-level constraints and team-level constraints based on the literature and interviews with experts. With these constraints, our approach optimized the scheduling of human resource allocation through a two-phase procedure. We showed that our approach can produce more realistic and efficient allocation results.

In addition, we developed a guideline to estimate the productivity of developers based on COCOMO II. This guideline supports various factors with respect to performing roles and module profiles. As productivity data are hard to obtain and manage, especially for low-maturity organizations, our guideline can provide help in using human resource allocation approaches for software projects.

Using real project data, we validated that our approach can effectively reduce the violation of constraints and produce efficient allocation results. Also, it was shown that our two-phase approach is efficient in reducing the time required in the optimization. The practicality of the allocation result was confirmed by the experts participating in the project.

DOI: 10.1002/spe

Our approach provides the following advantages for software projects: first, more practical human resource allocation can be provided since we apply software project constraints. Second, the time to market of software projects can be accelerated through the optimization. Third, efficient allocation can be obtained in a relatively short time with the two-phase approach.

More constraints related to human resource allocation of software projects can be applied depending on organization-specific project environments. Hence, in the future research, we will identify additional constraints and apply these to our allocation approach.

ACKNOWLEDGEMENTS

This research was partially supported by the Defense Acquisition Program Administration and the Agency for Defense Development under the contract.

REFERENCES

1. Tsai H, Moskowitz H, Lee L. Human resource selection for software development projects using Taguchi's parameter design. *European Journal of Operational Research* 2003; **151**:167–180.
2. Ibaraki T, Katoh N. *Resource Allocation Problems*: *Algorithmic Approaches* (*Foundations of Computing*). The MIT Press: Cambridge, MA, 1988.
3. Jones C. *Assessment and Control of Software Risks*. Yourdon Press: Upper Saddle River, NJ, U.S.A., 1994.
4. Humphrey WS. *Introduction to the Team Software Process*. Addison-Wesley: Reading, MA, 1999.
5. Chang CK, Jiang H, Di Y, Zhu D, Ge Y. Time-line based model for software project scheduling with genetic algorithms. *Information and Software Technology* 2008; **50**:1142–1154.
6. Duggan J, Byrne J, Lyons GJ. A task allocation optimizer for software construction. *IEEE Software* 2004; **21**(3):76–82.
7. Smith RK, Hale JE, Parrish AS. An empirical study using task assignment patterns to improve the accuracy of software effort estimation. *IEEE Transactions on Software Engineering* 2001; **27**(3):264–271.
8. Barreto A, Barrros MO, Werner CML. Staffing a software project: A constraint satisfaction and optimization-based approach. *Computers and Operations Research* 2008; **35**:3073–3089.
9. Begel A, Nagappan N. Coordination in large-scale software teams. *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, Vancouver, Canada, 2009; 1–7.
10. Stellman A, Greene J. *Applied Software Project Management*. O'Reilly Media Inc.: Sebastopol, CA, 2006.
11. Yoon BS, Cho GY. Acceleration of simulated annealing and its application for virtual path management. *Journal of the Korean Operations Research and Management Science Society* 1996; **21**(2):125–140.
12. Kirkpatrick S, Gelatt Jr CD, Vecchi M. Optimization by simulated annealing. *Science* 1983; **220**:671–690.
13. Bohem BW, Abts C, Brown AW, Chulani S, Clark BK, Horowitz E, Madachy R, Reifer D, Steece B. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR: Englewood Cliffs, NJ, 2000.
14. Laarhoven P, Aarts E. *Simulated Annealing*: *Theory and Applications*. Kluwer Academic Publishers: Dordrecht, 1989.
15. Bertsimas D, Tsitsiklis J. Simulated annealing. *Statistical Science* 1993; **8**(1):10–15.
16. Metropolis N, Rosenbluth A, Rosenbluth M, Teller A. Equations of state calculations by fast computing machines. *Journal of Chemical Physics* 1953; **21**(6):1087–1092.
17. Aarts E, Korst J. *Simulated Annealing and Boltzmann Machines*: *A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley: New York, 1989.
18. Jiménez A, Ríos-Insua S, Mateos A. Interactive simulated annealing for solving imprecise discrete multiattribute problems under risk. *Pesquisa Operacional* 2002; **22**(2):265–280.
19. Ngo-The A, Ruhe G. Optimized resource allocation for software release planning. *IEEE Transactions on Software Engineering* 2009; **35**(1):109–123.
20. Chang CK, Christensen MJ, Chang T. Genetic algorithms for project management. *Annals of Software Engineering* 2001; **11**:107–139.
21. Alba B, Chicano F. Software project management with GAs. *Information Science* 2007; **177**:2380–2401.
22. Hochbaum DS. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company: Boston, MA, U.S.A., 1997.
23. Berry M, Jeffery R. An instrument for assessing software measurement programs. *Empirical Software Engineering* 2000; **5**:183–200.
24. COPROMO. Available at: http://csse.usc.edu/csse/research/COPROMO/.
25. De Rore L, Snoeck M, Poels G, Dedene G. Cocomo II as productivity measurement: A case study at KBC. *FBE Research Report KBI 0829*, 2008; 1–66.
26. De Rore L, Snoeck M, Poels G, Dedene G. Deducing software process improvement areas from a COCOMO II-based productivity measurement. *Proceedings of the Fifth Software Measurement European Forum*, Milan, Italy, 2008; 163–174.

DOI: 10.1002/spe

27. Yun SJ, Simmons DB. Continuous productivity assessment and effort prediction based on Bayesian analysis. *Proceedings of the 28th Annual International Computer Software and Applications Conference*, Hong Kong, 2004; 44–49.
28. Choi K, Bae DH. Dynamic project performance estimation by combining static estimation models with system dynamics. *Information and Software Technology* 2009; **51**:162–172.
29. Johnson DS, Aragon CR, Mcgeoch LA, Schevon C. Optimization by simulated annealing: An experimental evaluation; Part I. Graph partitioning. *Operations Research* 1989; **37**:865–892.
30. Baker KR. *Introduction to Sequencing and Scheduling*. Wiley: New York, 1974.
31. Lenstra JK. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming* 1990; **46**:259–271.
32. Kim D, Kim K, Jang W, Chen FF. Unrelated parallel machine scheduling with setup times using simulated annealing. *Robotics and Computer Integrated Manufacturing* 2002; **18**:223–231.
33. Glass CA, Potts CN, Shade P. Unrelated parallel machine scheduling using local search. *Mathematical and Computer Modeling* 1994; **20**(2):41–52.
34. Lawler EL, Labetoulle J. On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM* 1978; **25**(4):612–619.
35. Herrmann J, Proth JM, Sauer N. Heuristics for unrelated machine scheduling with precedence constraints. *European Journal of Operational Research* 1997; **102**:528–537.
36. Anagnostopoulos GC. A simulated annealing algorithm for the unrelated parallel machine scheduling problem. *Proceedings of the Fifth Biannual World Automation Congress*, Orlando, FL, 2002; 115–120.
37. Xiao J, Wang Q, Li M, Yang Q, Xie L. Value-based multiple software projects scheduling with genetic algorithm. *Proceedings of the International Conference on Software Process*, Vancouver, Canada, 2009; 50–62.
38. Fenton N, Marsh W, Neil M, Cates P, Forey S, Tailor M. Making resource decisions for software projects. *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, U.K., 2004; 397–406.
39. Padberg F. A study on optimal scheduling for software projects. *Software Process Improvement and Practice* 2006; **11**:77–91.
40. Padberg F. Scheduling software projects to minimize the development time and cost with a given staff. *Proceedings of the Eighth Asia-Pacific Software Engineering Conference*, Macau, China, 2001; 187–194.
41. Lee B, Miller J. Multi-project management in software engineering using simulation modelling. *Software Quality Journal* 2004; **12**:59–82.
42. Antoniol G, Cimitile A, Di Lucca GA, Di Penta M. Assessing staffing needs for a software maintenance project through queuing simulation. *IEEE Transactions on Software Engineering* 2004; **30**(1):43–58.
43. Ohtera H, Yamada S. Optimal allocation & control problems for software-testing resources. *IEEE Transactions on Reliability* 1990; **39**(2):171–176.
44. Dai YS, Xie M, Poh KL, Yang B. Optimal testing-resource allocation with genetic algorithm for modular software systems. *Journal of Systems and Software* 2003; **66**(1):47–55.
45. Yamada ST, Nishiwaki IM. Optimal allocation policies for testing-resource based on a software reliability growth model. *Mathematical and Computer Modelling* 1995; **22**(10-12):295–301.
46. Hou RH, Kuo SY, Chang YP. Needed resources for software module test using the hyper-geometric software reliability growth model. *IEEE Transactions on Reliability* 1996; **45**(4):541–549.
47. Leung YW. Dynamic resource-allocation for software-module testing. *Journal of Systems and Software* 1997; **37**(2):129–139.

DOI: 10.1002/spe