Practical Human Resource Allocation in Software Projects Using Genetic Algorithm

Jihun Park, Dongwon Seo, Gwangui Hong, Donghwan Shin, Jimin Hwa, Doo-Hwan Bae
Department of Computer Science
Korea Advanced Institute of Science and Technology
{jhpark, dwseo, gwangui.hong, donghwan, jmhwa, bae}@se.kaist.ac.kr

Abstract

Software planning is becoming more complicated as the size of software project grows, making the planning process more important. Many approaches have been proposed to help software project managers by providing optimal human resource allocations in terms of minimizing the cost. Since previous approaches only concentrated on minimizing the cost, there has not been a study that considers the practical issues affecting project schedule in practice.

We elicited the practical considerations on the human resource allocation problem by communicating with a group of software project experts. In this paper, we propose an approach for solving the human resource allocation problem using a genetic algorithm (GA) reflecting the practical considerations. Our experiment shows that our algorithm considers the practical considerations well, in terms of continuous allocation on relevant tasks, minimization of developer multitasking time, and balance of allocation.

1 Introduction

As the size of software project increases, software planning process becomes more complicated and important. In addition, an inappropriate software plan often results in the failure of a project [15]. For these reasons, software project managers can significantly benefit from the human resource allocation technique. The human resource allocation technique automatically allocates each employee to the tasks to make an optimal plan of the project in terms of time and money.

Many researches have been proposed to deal with the human resource allocation problem. For example, Alba et al. [1] and Chang et al. [6] suggested a genetic algorithm (GA) approach for minimizing the project duration and project cost. Chang et al. [7] suggested a fine-grained representation of a human resource allocation result and used GA to find a schedule minimizing payment and delay penalty.

While these approaches only considered minimizing the cost in terms of time or money, they did not consider practical issues which can affect the actual development process when the plan is applied in a real world. We discussed with a group of software experts to elicit practical issues that affect the project schedule in practice. We then suggest a GA approach to minimize the inefficient assignments which can delay the schedule.

The rest of this paper is organized as follows. Section 2 explains the practical considerations for the human resource allocation problem and Section 3 describes our genetic algorithm. Section 4 presents a case study, Section 5 discusses threats to validity, Section 6 introduces related work, and Section 7 summarizes our findings.

2 Practical Considerations

By consulting with an expert group, we elicited practical issues which can affect the project schedule. The expert group consisted of managers and developers from a software development and consulting company, military software experts from a research institute, and a professor and graduate students. The derived practical considerations are described below.

C1. Short project plan The basic objective of human resource allocation problem is to generate a project plan that can be finished within the minimum time span, in order to reduce the entire project cost in terms of time and money.

C2. Minimization of multitasking time In practice, a developer can work on more than one task at the same time, while many researches assumed that a developer can work on only one task at a time [3, 9, 18]. Thus, we allow assigning developers to work on multiple tasks at the same time. If a developer is involved in too many tasks at a certain moment, however, productivity will decrease since the developer cannot concentrate on one task due to the frequently switching tasks. In addition, a developer involved in multiple tasks at the same time is more likely to introduce bugs. [11].

C3. Assignment on relevant tasks Since the task precedence relationship indicates closely related tasks, assigning a developer to both of the pre-task and the post-task is efficient in terms of minimizing the context-switching cost. If a developer should work on a series of tasks that are not related to each other, the developer must learn the new context for every assigned task, e.g., requirements or design of an unfamiliar module.

C4. Balance of allocation Task size should be considered in the human resource allocation problem. On the one hand, if a few developers are assigned to a huge task, developers will be overwhelmed by the heavy workload. On the other hand, if too many developers are assigned to a small task, the high communication overhead causes inefficiency. The staff level (e.g., director/manager/engineer) of developers also should be considered. Developers with high staff level have more experiences than lower level developers, so they will manage a task rather than concentrating on the implementation, which is the main work of a low staff level developer. To manage each task efficiently, developers having different staff levels should be assigned together.

Previous project scheduling algorithms do not reflect the inefficiencies caused by these practical issues, but in practice, project managers consider them very important. Our GA approach takes into account of these practical considerations by representing them as a part of the fitness function.

3 Human Resource Allocation with GA

3.1 Problem Description

Our problem is described by the tasks and developers. A software project consists of a set of tasks $T = \{t_1, t_2, \cdots, t_n\}$. We use the task precedence graph (TPG) to represent the precedence relationship between tasks. Figure 1 shows an example of a TPG. In Figure 1, task t_1 must be completed before t_2, t_3 and t_4 begin. Each task $t_i (i = 1, 2, \cdots, n)$ is defined by the following attributes.

- *type*$_i$: The type of the task. In our research, four task types which are the basic phases for software process models are used: analysis, design, implementation, and testing.

- $effort_i$: The effort estimated by the project manager, which is assumed to be available as an input. Project manager can use existing effort estimation techniques, such as COCOMO models [4, 5], or analogy-based software effort estimation [16]. The unit of effort is man-hours.

- *PT*$_i$: A set of preceding tasks for the task. A task cannot begin if any of the preceding task is not completed. TPG is constructed based on the precedence relationship.

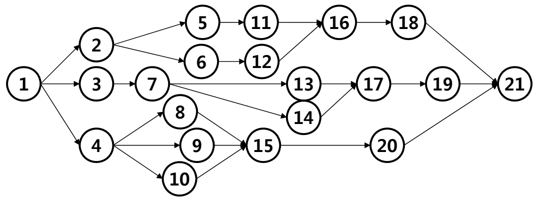


Figure 1. A Task Precedence Graph (TPG)

A set of developers $D = \{d_1, d_2, \cdots, d_m\}$ is allocated to tasks. We assume that the developer information is managed by a database, and the project manager can access it for planning a project. Developer $d_j (j = 1, 2, \cdots, m)$ is defined by the following attributes

- *sl*$_j$: The staff level of the developer. In our problem, we assume that there are three hierarchical staff levels —director, manager, and engineer.

- *ability*$_j^k$: The ability of the developer for task type k. Developers have different levels of proficiency on each task type. If a developer has 0.7 as the ability value on the *design* task type, he decreases 0.7 remaining manhour per a time unit when he is only working on the task.

A single allocation for a task is represented by $\{t_i, \{d_k, \cdots, d_l\}\}$, and *the assignment result* consists of the allocations for each task.

3.2 Genetic Algorithm

A genetic algorithm (GA) is an evolutionary search-based heuristic algorithm introduced by Holland [12]. Many researches tried to utilize the GA to find an optimal solution for the human resource allocation problem, which has a very large search space [1, 2, 6, 7]. We develop a GA approach reflecting the practical considerations to minimize inefficient allocations that can delay the schedule in practice. Figure 2 shows the pseudocode of our GA. Each step of the algorithm is explained in this section.

Representation We define a chromosome to represent an assignment result as a fixed length of genes. The chromosome has $|T|$ number of genes, where each gene records the assigned developers for each task. For example, the first gene contains a set of developers who are assigned to the task t_1. Figure 3 shows an example of a chromosome.

Initial population The first step of our GA algorithm is to generate an initial population of chromosomes. The initial

```
//Initial population
Generate initial population $P_0$
Initialize generation counter $g \leftarrow 0$

while($g$ < the maximum number of generation counter){
  Generate empty population $P_{g+1}$
  //Assessment
  Evaluate each chromosome in population $P_g$

  //Elitism Selection
  Copy a chromosome $c_{best}$ which has the highest fitness
     value from $P_g$

  Copy $c_{best}$ into $P_{g+1}$
  while(the number of chromosome in $P_{g+1}$ != the number
     of chromosome in $P_g$ ){

    //Tournament Selection
    Select $c_1$ and $c_2$ from $P_0$ using tournament selection

    //Crossover
    Generate $c_3$ from Crossover $c_1$, $c_2$

    //Mutation
    for ($gene_i$ in $c_3$)
      if (rand(0,1) < $mutation\ rate$) Mutate $gene_i$

    Copy $c_3$ into $P_{g+1}$
  }
}
Evaluate each chromosome in population $P_g$
Copy $c_{best}$ which has the highest fitness value from $P_g$

return $c_{best}$
```

Figure 2. Genetic algorithm

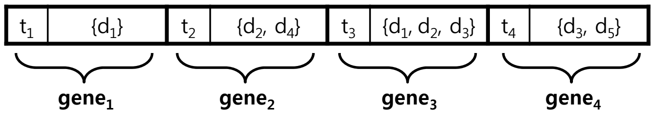


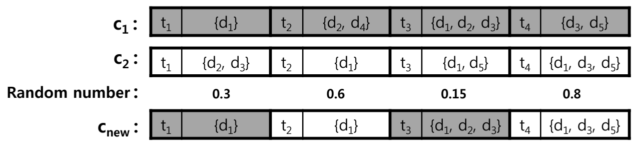
Figure 3. An example of a chromosome

population comprises the $populationSize$ number of chromosomes. We randomly assign a developer to a task, until every task has at least one assigned developer.

Assessment We evaluate each chromosome to identify a superior one among the population. The fitness function assesses a chromosome by simulating the solution and inspecting the assignment structure. Details of the fitness function is described in Section 3.3.

Selection Selection step identifies superior chromosomes that should survive until the next generation and pass their genes down to the next generation. On the one hand, the elitism selection passes on the fittest chromosome of the current generation to the next generation. It prevents degradation of the fittest chromosome. On the other hand, the tournament selection chooses a chromosome to be a parent of the next generation. It randomly selects k (tournament size) chromosomes from the current population, and picks the fittest one among them. In this study, k=5 is used.

Crossover Crossover step generates chromosomes for the next generation using two parent chromosomes selected by the tournament selection. We use uniform crossover which generates a new chromosome by randomly taking each gene from the parents. We repeat the tournament selection and crossover steps to make new chromosomes until the number of chromosomes in the next generation becomes equal to the predefined population size. Figure 4 shows an example of the uniform crossover.



c_1:
t_1 $\{d_1\}$ t_2 $\{d_2, d_4\}$ t_3 $\{d_1, d_2, d_3\}$ t_4 $\{d_3, d_5\}$
c_2:
t_1 $\{d_2, d_3\}$ t_2 $\{d_1\}$ t_3 $\{d_1, d_5\}$ t_4 $\{d_1, d_3, d_5\}$
Random number:
0.3 0.6 0.15 0.8
c_{new}:
t_1 $\{d_1\}$ t_2 $\{d_1\}$ t_3 $\{d_1, d_2, d_3\}$ t_4 $\{d_1, d_3, d_5\}$

Figure 4. An example of the uniform crossover

Mutation Mutation step maintains genetic diversity. With a certain probability of the mutation ($mutation\ rate$), each gene is mutated using one of the three mutation operators: *assigning* a random developer to the task, *removing* a random developer from the task, and *replacing* an assigned developer with a random developer. In this study, we use 0.05 for the mutation rate.

3.3 Fitness Function

A fitness function assesses each chromosome to identify a superior one among the population. The fitness function encodes the requirements of our GA, which means the better human resource allocation results in the higher fitness score. The formula for our fitness function is as follows.

$$FitnessScore = w_1 * CMscore + w_2 * CEscore + w_3 * CCscore + w_4 * BAscore$$

Each w_i represents the weight for each score, and each score encodes our practical considerations. CM (Cost Minimization) score assesses whether the project cost is minimized in terms of time, and CE (Concentration Efficiency) score represents how many developers are involved in multitasking at each time unit. CM and CE scores are calculated with the scheduling simulation. CC (Concentration Consideration) score assesses whether developers are assigned to tasks that have precedence relationships, and BA (Balance of Allocation) score represents whether developers are evenly allocated to tasks considering the task size and the staff level. The CM, CE, CC, and BA scores reflect the practical consideration *C1*, *C2*, *C3*, and *C4*, respectively.

3.3.1 Scheduling Simulation

We inspect the estimated time span of an assignment result and how assigned developers perform each task, using

```
AllocationResult solution
List<Task> remainingTasks
List<Task> enabledTasks
List<Task> runningTasks
List<Task> completedTasks

Add tasks that do not have pre-task to enabledTasks
while(completedTasks.size() != totalNumOfTasks) {
  TimeTick++
  Assign developers to enabled tasks using solution
  running task ← enabled task

  for(runingTask in runningTasks){
    // Perform task
    Decrease remaining man-hour of running tasks
    if (runningTask.remainingMH <= 0){
      // Complete the task
      completed task ← running task}
  }
  for(remainingTask in remainingTasks){
    if (all preTask of a remainingTask is completed)
      // Enable the task
      enabled task ← remaining task}
  }
}
```

Figure 5. Scheduling simulation algorithm

a scheduling simulation algorithm (Figure 5).

The allocation result (variable solution) is an input of this algorithm. At the beginning of the algorithm, every task is added to remainingTasks and then the task which does not have any pre-task is added to enabledTasks.

At the beginning of the while loop, the time tick increases. The start and finish time of each task are calculated with this time tick. The finish time of the last task is regarded as total time span of a project.

For each task in enabledTasks, developers are assigned as recorded in solution, and the task is moved to runningTasks. The remaining man-hour of each task in runningTasks is decreased by the sum of the assigned developers' ability. The ability of the developer is assumed to exist as an input. If a developer is working on more than one task, his ability on each task is divided by the number of assigned tasks, because his capability is divided among the tasks.

After decreasing the remaining man-hours of running tasks, running tasks with its remaining man-hour less than or equal to zero is moved to completedTasks. Tasks in remainingTasks that every pre-task is completed are moved to enabledTasks.

3.3.2 Fitness Scores

Cost Minimization (CM) Score The CM score assesses whether the solution finishes early. (*C1*) The formula for CM score is as follows.

$$CM\ score = \frac{minTS(S)}{TS(S)}$$

$$minTS(S) = \sum_{k\in types}\frac{\sum_{\{t_i\in T|type_i=k\}} effort_i}{max(ability_1^k,\cdots,ability_m^k)*|D|}$$

The $minTS(S)$ refers to the minimum time span, and $TS(S)$ refers to the time span of the solution S which can be calculated by the simulation algorithm. The $minTS(S)$ is calculated with the assumption that every developer has the highest ability among developers at the given task type and there is no multitasking overhead or communication cost. The CM score becomes higher if the time span of the solution comes to the minimum time span.

Concentration Efficiency (CE) Score To assess the burden of multitasking (*C2*), we calculate the number of tasks that a developer has to deal with at each time unit. At a certain moment, a developer might work on more than one task because the assigned tasks are performed at the same time, but such multitasking can cause inefficiency in practice, which delays the project progress behind the expected schedule. The formula for the CE score is as follows.

$$CE\ score = \frac{1}{|D|}\sum_{d_j\in D}\frac{|\{\forall t \,|\, |\#involve(d_j,t)|\geq 1\}|}{\sum_{t=1}^{finishTime}|\#involve(d_j,t)|}$$

$|\#involve(d_j,t)|$ refers to the number of tasks that developer d_j is involved in at time t. For each developer, we divide the total working time units by the sum of the number of involved tasks at those time units to get the partial CE score. For example, if a developer performs [2, 1, 0, 4] tasks on time unit 1 to 4, the developer level CE score is calculated by 3 / 2+1+4. We average all the partial scores for each developer to get the CE score. The CE score comes to 1.0 if every developer performs only one task at every time unit that he works.

Continuity Consideration (CC) Score The CC score assesses how well the human resource allocation result considers the precedence relationship between tasks (*C3*). We count the number of unit allocation $ua_x = \{t^x, d^x\}$ that do not consider continuity, in which the developer is not assigned any pre-tasks of the task t^x. The solution of human resource allocation S' is represented by a set of unit allocation ($S' = \{ua_1, ua_2, ..., ua_n\}$), and the CC score is calculated with the formula below.

$$CC\ score = \frac{|\{ua_x\in S' \,|\, (\exists(ua_k)\in S')\wedge(t^k\in PT^x)\wedge(d^k=d^x)\}|}{|S'|}$$

The numerator of the formula counts how many times the developer of a unit allocation (d^x) is assigned to any pre-task of the task (PT^x). The CC score comes to 1.0 if every unit assignment considers continuity.

Balance of Allocation (BA) Score We assess how evenly the developers are allocated to tasks considering the task size and staff level using the BA score (*C4*). The BA score uses *Shannon entropy* [17] which assesses the uncertainty in a distribution. The normalized *Shannon entropy* is defined as: $H = -\sum_{i=1}^{n}(p_i * log_n p_i)$, where p_i is the probability that each element occurs ($p_i \geq 0$ and $\sum_{i=1}^{n} p_i = 1$). The value is maximized when all p_is have the same probability, i.e., $p_i = 1/n, \forall i \in 1, 2, ..., n$. We calculate the BA score with the formula below.

$$BA\ score = \frac{\sum_{s \in staff\ level} Entropy^s}{|\{staff\ level\}|}$$

$$Entropy^s = -\sum_{i=1}^{n}(p_i^s * log_n p_i^s)$$

$$p_i^s = \frac{\frac{\#assigned_i^s}{effort_i}}{\sum_{t_k \in T} \frac{\#assigned_k^s}{effort_k}}$$

$Entropy^s$ represents the entropy value at each staff level, and p_i^s represents the normalized value for the number of assigned developers of the staff level s at task t_i ($\#assigned_i^s$) divided by the effort of the task ($effort_i$). The avearge of the entropy values for each staff level is calculated to assess the BA score. The BA score becomes 1.0 if all p_i are the same, meaning that the number of assigned developers to a task is proportional to the effort of the task.

4 Case Study

We assess how well our GA reflects the practical considerations, by comparing the result when the GA only considers cost minimization ($Case_{time}$, weights for the fitness score = $\{1, 0, 0, 0\}$), and the result when considering all the objectives ($Case_{all}$, weights for the fitness score = $\{0.25, 0.25, 0.25, 0.25\}$).

4.1 Experimental Setup

We generate three experiment sets. Set1, Set2, and Set3 consist of 11 tasks / 7 developers, 11 tasks / 10 developers, and 21 tasks / 10 developers respectively. Table 1 and Table 2 describe a task set T_1 with 11 tasks and a developer set D_1 with 7 developers. The detail of a developer set with 10 developers and a task set with 21 task set is omitted because of the space limit.

Our GA uses 100 population sizes, and we set the maximum generation count to 400. We inspect the tendency of the fitness value along different GA parameters, then we determine the appropriate parameters that GA explores enough search space to get the optimal solution. Selecting the optimal parameters is beyond the scope of this paper.

We compare 1) *the time span of the result*, 2) *the average time units that each developer works on more than one task*, 3) *the number of assignments not considering the task precedence*, 4) *the number of tasks that only one level developers are assigned*, and 5) *the mean and variance of (the number of assigned developers / $effort_i$) for each task.* We repeat each experiment 100 times to compare the average value. Table 3 summarizes the results.

ID	$type_i$	$effort_i$	PT_i
t_1	Analysis	400	
t_2	Design	320	1
t_3	Design	240	1
t_4	Design	240	1
t_5	Implementation	240	2
t_6	Implementation	600	3
t_7	Implementation	160	4
t_8	Test	100	5
t_9	Test	80	6
t_{10}	Test	70	7
t_{11}	Test	80	8,9,10

Table 1. Task set T$_1$

ID	sl_j	$ability_j^k$			
		$analysis$	$design$	$implementation$	$test$
d_1	3	1.25	1	1.25	1.25
d_2	3	1.25	1	1.25	0.75
d_3	2	0.75	0.75	1	1
d_4	2	0.75	1	1	0.75
d_5	1	1	0.75	0.75	1
d_6	1	0.75	1	0.75	0.75
d_7	1	1	0.75	1	0.75

Table 2. Developer set D$_1$

4.2 Experimental results

The time span of the result We find that $Case_{time}$ take 5.99% to 12.07% shorter time span than $Case_{all}$. We simulate the time span of each solution, but in practice, several issues can affect the actual time span, such as multitasking overhead, context switching cost, and hierarchical management efficiency. Considering that we minimize these practical issues that can delay the project schedule, we conclude that the difference is acceptable.

The average multitasking time We investigate how long developers are involved in multitasking, by studying the average time that a developer works on more than one task. We find that the average multitasking time of each developer in $Case_{time}$ is 3.51 to 6.94 times of $Case_{all}$. High multitasking time represents that developers should work more than one task at the same time for a longer time in $Case_{time}$ schedule than in $Case_{all}$.

Assignments not considering precedence relationship We assess whether the human resource allocation result reduces the inefficiency of context switching, by comparing the number of unit assignment (see section 3.3.3) which do not consider the task precedence relationship. We find that the number of assignments not considering precedence relationship in $Case_{time}$ is 2.31 to 2.72 times of $Case_{all}$. The result indicates that developers are assigned to a task of different context from previous task more often in $Case_{time}$

Metric	Set1 #task=11 #dev=7		Set2 #task=11 #dev=10		Set3 #task=21 #dev=10	
	All	Time	All	Time	All	Time
Time (h)	342.71	322.17	239.70	225.04	846.30	744.14
Multitasking Time (h)	28.96	101.76	12.49	54.94	58.38	404.93
# no precedence assignments	7.91	18.31	10.92	27.08	15.11	41.17
# tasks only one level asisgned	3.73	3.05	1.69	3.07	5.74	6.56
Mean (# assigned devs / *effort*$_i$)	2.15e-02	3.50e-02	2.80e-02	4.70e-02	1.07e-02	1.88e-02
Variance (# assigned devs / *effort*$_i$)	1.71e-04	7.47e-04	2.69e-04	1.48e-03	4.29e-05	1.24e-04

Table 3. Experiment results

than in $Case_{all}$

The number of tasks that only one staff level of developers are allocated To investigate whether developers having different staff levels are evenly assigned to each task, we compare the number of tasks that only one staff level of developers are assigned in $Case_{time}$ and $Case_{all}$. We find that the number is higher in $Case_{time}$ than in $Case_{all}$ for Set2 and Set3, but the result is reversed on Set1. Because the number of high level developer is limited and our GA minimize multitasking time in $Case_{all}$, there is no significant difference of the number in $Case_{all}$ and $Case_{time}$. We find that the average multitasking time and the number of tasks with only one staff level developers are inversely proportional. The result shows that our GA in $Case_{all}$ considers the staff level slightly better than $Case_{time}$, but minimize multitasking time in $Case_{all}$ significantly.

The mean and variance of (the number of assigned developers / $effort_i$) for each task To identify how many developers are assigned to each task proportional to the effort, and how stable the value is, we investigate the mean and variance of *# of assigned developers / $effort_i$* . Overall, the means and variances are higher in $Case_{time}$ than in $Case_{all}$. This results indicate that more developers are assigned when we only consider time, which can cause communication overhead and longer multitasking time. Moreover, large variances indicate there are more cases that too many developers are assigned to a small task or small number of developers are assigned to a large task.

5 Discussion

We use the task type to represent a type of the task, rather than defining a skill set needed to do a task. Many previous approaches [7, 8, 18] used skill sets to represent required capability of a task, but we abstract the skill sets using the task type. Our assumption is that defining and inputting each skill set and the capability of developers for each skill is difficult for project managers, especially when the number of skill is large. A project manager can use our tool by categorizing tasks into several types, and defining proficiency of developers on each category type, which is simpler than a skill set representation.

Our approach generates only one solution which is the fittest one, not considering relative proportion of each subscores. We can apply MOEA approach [13], which generates a set of non-dominated solutions when the fitness function comprises the weighted sum of multiple sub-scores.

6 Related Work

Duggan et al. [9] suggested a GA approach as an multi objective evolutionary approach to consider package precedence, full team utilization, and cross-communication overhead. They only considered that the successor package should be developed after the predecessor, not considering that the developers should be assigned to tasks with a precedence relationship to lessen the context switching cost. Barreto et al. [3] suggested a optimization-based approach to find teams satisfying constraints established by the software development organization. These work assumed that developers can work on only one task at a time, which was unrealistic in practice. Their approaches required a fine-grained project plan in which a task size was small enough to be finished by a developer.

Chang et al. [7] proposed a human resource allocation technique based on GA with the concept of time-line. The assignment was represented by a three-dimensional array with time, tasks, and employee axes. As time goes with the pre-defined time unit, the human resources were allocated to tasks. Genetic algorithm was used to search the schedule that minimized payment and delay penalties. Chen et al. [8] improved Chang et al.'s work [7] by introducing the event-based scheduler and the ant colony optimization technique. These approaches could allocate human resources in a fine-grained manner, but the allocation was often too fragmented. In addition, they only concentrated on minimizing the cost, not considering practical issues that we consider in this paper.

Gerasimou et al. [10] investigated the human resource allocation problem using a particle swarm optimization technique. Their fitness function evaluated whether each task finished before the deadline and whether an appropriate developer was assigned to a task considering the ability of the developer. Their approach allowed developers to work on more than one task at the same time, but did not limit or manage the number of tasks on which a developer can work.

Kang et al. [14] proposed a constraint-based approach considering constraints that affected the project schedule. Their constraints included continuous allocation to related tasks, minimization of sharing developers among tasks, restriction on the number of developers assigned to a task, and avoiding novice teams. These constraints encoded some of practical considerations of ours, but they assumed that each program module can always be developed in parallel, not considering the precedence between modules and the integration of the modules.

7 Conclusion

Many approaches have been proposed to find optimal human resource allocations. The majority of them only considered minimizing the expected cost of the plan, and not practical issues that can affect the actual schedule in practice. With a group of software experts, we elicited the practical considerations for the human resource allocation problem. We design a genetic algorithm to reflect the practical issues, by encoding them as a part of the fitness function. The fitness function consists of weighted sum of the four fitness scores considering cost minimization, concentration efficiency, continuity consideration, and balance of allocation. Our experiment shows that when the four fitness scores are considered altogether, our GA generates a practical human resource allocation result, in terms of less multitasking time, less assignments not considering task precedence, and more even allocations than an algorithm which the objective only aims at minimizing the time span.

As future work, we will find the optimal parameters and operations for GA, by studying the effect of mutation rate, tournament size, and weight values for the fitness function, and investigating different selection, crossover, mutation approaches. We will also identify more practical issues that can be applied to our approach.

Acknowledgements

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

References

[1] E. Alba and J. Francisco Chicano. Software project management with gas. *Information Sciences*, 177(11):2380–2401, 2007.

[2] G. Antoniol, M. Di Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 240–249, Washington, DC, USA, 2005. IEEE Computer Society.

[3] A. Barreto, M. Barros, and C. Werner. Staffing a software project: A constraint satisfaction approach. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[4] B. W. Boehm. Software engineering economics. 1981.

[5] B. W. Boehm, R. Madachy, B. Steece, et al. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[6] C. K. Chang, M. J. Christensen, and T. Zhang. Genetic algorithms for project management. *Ann. Softw. Eng.*, 11(1):107–139, Nov. 2001.

[7] C. K. Chang, H. yi Jiang, Y. Di, D. Zhu, and Y. Ge. Time-line based model for software project scheduling with genetic algorithms. *Information and Software Technology*, 50(11):1142 – 1154, 2008.

[8] W.-N. Chen and J. Zhang. Ant colony optimization for software project scheduling and staffing with an event-based scheduler. *Software Engineering, IEEE Transactions on*, 39(1):1–17, Jan 2013.

[9] J. Duggan, J. Byrne, and G. J. Lyons. A task allocation optimizer for software construction. *IEEE Softw.*, 21(3):76–82, May 2004.

[10] S. Gerasimou, C. Stylianou, and A. S. Andreou. An investigation of optimal project scheduling and team staffing in software development using particle swarm optimization. In *ICEIS (2)*, pages 168–171, 2012.

[11] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[12] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[13] H. Ishibuchi and T. Murata. A multi-objective genetic local search algorithm and its application to flowshop scheduling. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 28(3):392–403, Aug 1998.

[14] D. Kang, J. Jung, and D.-H. Bae. Constraint-based human resource allocation in software projects. *Software: Practice and Experience*, 41(5):551–577, 2011.

[15] H. R. Kerzner. *Project Management-Best Practices: Achieving Global Excellence*. John Wiley & Sons, 2014.

[16] J. Li, G. Ruhe, A. Al-Emran, and M. M. Richter. A flexible method for software effort estimation by analogy. *Empirical Software Engineering*, 12(1):65–106, 2007.

[17] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[18] V. Yannibelli and A. Amandi. A knowledge-based evolutionary assistant to software development project scheduling. *Expert Systems with Applications*, 38(7):8403–8413, 2011.