An Ant Colony Optimization Approach to the Software Release Planning with Dependent Requirements

Jerffeson Teixeira de Souza, Camila Loiola Brito Maia,
Thiago do Nascimento Ferreira, Rafael Augusto Ferreira do Carmo,
and Márcia Maria Albuquerque Brasil

Optimization in Software Engineering Group (GOES.UECE),
State University of Ceará (UECE)
60740-903 Fortaleza, Brazil
jeff@larces.uece.br,
{camila.maia,thiagonascimento.uece@,carmorafael}@gmail.com,
marcia.abrasil@gmail.com

Abstract. Ant Colony Optimization (ACO) has been successfully employed to tackle a variety of hard combinatorial optimization problems, including the traveling salesman problem, vehicle routing, sequential ordering and timetabling. ACO, as a swarm intelligence framework, mimics the indirect communication strategy employed by real ants mediated by pheromone trails. Among the several algorithms following the ACO general framework, the Ant Colony System (ACS) has obtained convincing results in a range of problems. In Software Engineering, the effective application of ACO has been very narrow, being restricted to a few sparse problems. This paper expands this applicability, by adapting the ACS algorithm to solve the well-known Software Release Planning problem in the presence of dependent requirements. The evaluation of the proposed approach is performed over 72 synthetic datasets and considered, besides ACO, the Genetic Algorithm and Simulated Annealing. Results are consistent to show the ability of the proposed ACO algorithm to generate more accurate solutions to the Software Release Planning problem when compared to Genetic Algorithm and Simulated Annealing.

Keywords: Ant Colony Optimization, Search Based Software Engineering, Software Release Planning.

1 Introduction

The Search Based Software Engineering (SBSE) field [1] has been benefiting from a number of general search methods, including, but not limited to, Genetic Algorithms, Simulated Annealing, Greedy Search, GRASP and Tabu Search. Surprisingly, even with the large applicability and the significant results obtained by the Ant Colony Optimization (ACO) metaheuristic, very little has been done regarding the employment of this strategy to tackle software engineering problems modeled as optimization problems.

Ant Colony Optimization [2] is a swarm intelligence framework, inspired by the behavior of ants during food search in nature. ACO mimics the indirect communication strategy employed by real ants mediated by pheromone trails, allowing individual ants to

M.B. Cohen and M. Ó Cinnéide (Eds.): SSBSE 2011, LNCS 6956, pp. 142–157, 2011.

adapt their behavior to reflect the colony´s search experience. Ant System (AS) [3] was the first algorithm to follow the ACO general framework. First applied to tackle small instances of the Traveling Salesman Problem (TSP), AS was not able to compete with state-of-the-art algorithms specifically designed for this traditional optimization problem, which stimulated the development of significant extensions to this algorithm. In particular, the Ant Colony System (ACS) algorithm [4] improved AS by incorporating an elitist strategy to update pheromone trails and by changing the rule used by ants to select the next movement. These enhancements considerably increased the ability of the algorithm to generate precise solutions to hard and different combinatorial problems, including the TSP [4], vehicle routing [5], sequential ordering [6] and timetabling [7].

The application of ACO to address software engineering problems has been very narrow, but relevant. To this date, as detailed in the next section of this paper, the literature has reported works using ACO on software testing [8], model checking [9] and requirement engineering [10,11]. Thus, it seems that the full potential of the ACO algorithm is far from being completely explored by the SBSE research community.

In Requirement Engineering, some important problems have been shaped and addressed as optimization problems, including the Software Requirement Prioritization problem [12], the Next Release problem (NRP) [13] and a multi-objective version of the NRP [14], to mention some. Finally, a search based representation of the Software Release Planning problem (SRP) was formulated in [15], dealing with the allocation of requirements to software releases, or increments, by considering the importance of the requirements as specified by the stockholders, and respecting dependency and effort constraints.

As an important Requirement Engineering problem, several strategies have been applied in the attempt to solve the Software Release Planning problem, including Genetic Algorithms and Simulated Annealing. However, no attempt has been made, thus far, to adapt and apply the Ant Colony Optimization metaheuristic to this problem, even though such an application seems reasonably justified, given the nature of the Software Release Planning problem which can be interpreted as a Multiple Knapsack Problem and the fact that ACO has been reported to perform significantly well in solving such problem [18,19,20].

Therefore, this paper will report the results of a work aimed at answering the following research questions:

> RQ1. – *ACO for the Software Release Planning problem*: how can the ACO algorithm be adapted to solve the Software Release Planning problem in the presence of dependent requirements?
> RQ2. – *ACO versus Other Metaheuristics*: how does the proposed ACO adaptation compare to other metaheuristics in solving the Software Release Planning problem in the presence of dependent requirements?

The remaining of the paper is organized as follows: Section 2 summarizes prior works on the application of ACO to solve software engineering problems. Section 3 formally defines the Software Release Planning problem. Next, in Section 4, the proposed ACO algorithm for the Software Release Planning problem is presented, starting with the encoding of the problem, followed by a detailed description of the algorithm. Section 5 discusses, initially, the design of the experiments aimed at answering our second research question, presents the results from the experiments and describes the

analyses of these results. Finally, Section 6 concludes the work and points out some future directions to this research.

2 Previous Works

As mentioned earlier, very few works have been performed on the application of ACO to solve software engineering problems. Next, some of these researches are summarized.

An ACO-based algorithm for error tracing and model checking is designed and implemented in [8]. In [9], the ACOhg metaheuristic, an ACO algorithm proposed to deal with large construction graphs, is combined with partial order reduction to reduce memory consumption. This combination is employed to find safety property violations in concurrent models.

The first proposal for the application of ACO in requirement engineering, more specifically to tackle the NRP, appeared in [10]. This proposal was later fully developed in [11], where an ACS algorithm is established to search for a subset of requirements with maximum satisfaction and subject to a certain effort bound. When compared to the Simulated Annealing and the Genetic Algorithm metaheuristics, over a real software problem with 20 requirements, ACO was the only algorithm which found the known best solution to the problem every time.

3 The Software Release Planning Problem Definition

This section formally defines the Software Release Planning problem (SRP) as it will be considered on this paper.

Let $R = (r_1, r_2, \dots, r_N)$ be the set of requirements to be implemented and allocated to some software release, with N representing the number of requirements. The implementation of each requirement r_i demands certain implementation cost, denoted by $cost_i$. In addition, each requirement r_i has an associated risk, given by $risk_i$. Let $C = (c_1, c_2, \dots, c_M)$ be the set of clients, whose requirements should be delivered. Each client c_j has a given importance to the organization, defined as wt_j.

Consider, yet, $S = (s_1, s_2, \dots, s_P)$ to be the set of software releases to be planned, with P representing the number of releases. Each release s_k has an available budget ($budgetRelease_k$) which should be respected. Additionally, different clients have different interests in the implementation of each requirement. Therefore, $importance(c_j, r_i)$ defines the importance, or business value, client c_j attributes to requirement r_i.

Therefore, the Software Release Planning problem can be mathematically formulated with the following objective and restriction functions:

$$Maximize \ \sum_{i=1}^{N} (score_i \,.\, (P - x_i + 1) - risk_i \,.\, x_i) \,.\, y_i$$

subject to

$$\sum_{i=1}^{N} cost_i \,.\, f_{i,k} \ \leq \ budgetRelease_k, \ for \ all \ k \in \{1, \dots, P\}$$

$$x_b \ \leq \ x_a \,, \forall (r_a \ \rightarrow \ r_b) \,, where \ r_a, r_b \in R$$

where the boolean variable y_i indicates whether requirement r_i will be implemented in some release, variable x_i indicates the number of the release where requirement r_i is to be implemented, and the boolean value $f_{i,k}$ indicates whether requirement r_i will be implemented in release k.

The first component of the objective function expresses the weighted overall satisfaction of the stakeholders, where

$$score_i = \sum_{j=1}^{M} wt_j . importance(c_j, r_i)$$

represents the aggregated business value for requirement r_i. The other component deals with risk management, expressing those requirements with higher risk should be implemented earlier. Finally, the two restrictions will, respectively, limit the implementation cost of requirements in each release and guarantee that the precedence among requirements will be respected. In this second restriction, $r_a \rightarrow r_b$ represents that requirement r_b is a precedent of requirement r_a, indicating that r_b should be implemented earlier or at the same release as r_a.

4 An ACO Algorithm for Software Release Planning

This section describes the main contribution of this work, the proposed ACO algorithm for the Software Release Planning problem. First, however, it outlines the problem encoding which will allow the application of such algorithm.

4.1 Problem Encoding

To be able to apply an Ant Colony Optimization strategy to the Software Release Planning problem, a proper encoding is required. For that purpose, a graph will be generated with the Release Planning problem instance information, as follows. Two types of edges will be created, representing mandatory and optional moves for a particular ant. Mandatory moves will be created to ensure that precedence constraints are respected.

This way, the problem will be encoded as a directed graph, $G = (V, E)$, where $E = E_m + E_o$, with E_m representing mandatory moves, and E_o representing optional ones.

i. each vertex in V represents a requirement r_i;
ii. a directed mandatory edge $(r_i, r_j) \in E_m$, if $(r_i \rightarrow r_j)$;
iii. a directed optional edge $(r_i, r_j) \in E_o$, if $(r_i, r_j) \notin E_m$ and $i \neq j$.

Consider $overall\_cost_i$ to represent the cost of implementing requirement r_i, that is, $cost_i$, in addition to the cost of implementing all unvisited precedent requirements of requirement r_i. That means, $overall\_cost_i = cost_i$ if requirement r_i has no precedent requirements and $overall\_cost_i = cost_i + \sum overall\_cost_j$, for all unvisited requirements r_j where $r_i \rightarrow r_j$, i.e, requirement r_j is a precedent of r_i.

When visiting a node r_i, an ant k, which will select requirements do release k, will define two sets, $mand\_vis_k(i)$ and $opt\_vis_k(i)$, representing, respectively, mandatory and optional moves. These two sets are defined as follows:

$$mand\_vis_k(i) = \{r_j | (r_i, r_j) \in E_m \text{ and } visited_j = False\}$$

That is, the set of mandatory moves for an ant k visiting node r_i will contain the requirements r_j in the graph which respect the two conditions below:

1. r_j can be reached from r_i using a mandatory edge, that is, $(r_i, r_j) \in E_m$;
2. node r_j has not been previously visited, i.e., $visited_j = False$.

Additionally, $opt\_vis_k(i)$ can be defined as:

$$opt\_vis_k(i) = \{r_j | (r_{i,} r_j) \in E_o, effort(k) + overall\_cost_j \leq budgetRelease_k \text{ and } visited_j = False\}$$

Similarly, set $opt\_vis_k(i)$ is defined with the same rules above, except, in rule 1., node r_j can be reached from r_i using an optional edge, in other words, $(r_{i,} r_j) \in E_o$, and by the addition of a third rule, which verifies whether a new requirement can be added to the release without breaking the budget constraint, in a way that the added effort for implementing r_j, does not exceed the defined budget for release k, that is, $effort(k) + overall\_cost_j \leq budgetRelease_k$.

The heuristic information of a particular node, represented by a productivity measure for the respective software requirement, is defined as:

$$w_i = \mu \cdot (score_i * risk_i)$$

where μ is a normalization constant.

At each step, in case the random value $q \leq q_0$ (where q_0 is a parameter), an ant k visiting a node r_i will have the probability of moving to the node r_j given by:

$$p_{ij}^k = \frac{\tau_{ij}^\alpha \cdot w_j^\beta}{\sum_{r_j \in opt\_vis_k(i)} \tau_{ij}^\alpha \cdot w_j^\beta}$$

where τ_{ij} is the amount of pheromone in edge $(r_{i,} r_j)$ and α and β are parameters controlling the relative importance of pheromone and requirement information. When $q > q_0$, ant k visiting a node r_i will move to r_j with higher value given by $\tau_{ij} \cdot w_j^\beta$.

Pheromone will be initially distributed, equally, only on optional edges. When crossing a particular optional edge $(r_{i,} r_j)$, ant k will update the pheromone as follows:

$$\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} + \varphi \cdot \tau_0$$

with $\varphi \in (0,1]$ representing the pheromone decay coefficient.

This encoding will allow us to generate an ACO adaptation to the Software Release Planning problem, as proposed in the next section.

4.2 The ACO Algorithm

The overall structure of the ACO Algorithm for the Software Release Planning problem is: each ant will be responsible for iteratively constructing a single release. At an iteration of the algorithm, k ants will be deployed, one at a time and in order, which will produce a complete release planning. A roulette procedure, considering the heuristic information measure of a node will be employed to select the initial node of each ant. Here, an ant can only be placed in a requirement that can fit, along with its precedent nodes, in that particular release. The first ant will travel through the generated graph, adding requirements to the first release until no more additions are allowed, due to budget constraints. Next, a second ant will be placed in an unvisited node, using the roulette procedure, and will start its traversal through the graph, constructing the second release. The process is repeated until all k ants have been delivered and produced their respective releases. At this point, a solution to the

Software Release Planning problem has been constructed. The process is repeated a number of times. At the end, the best found solution is returned.

Next, in Fig. 1, the general ACO algorithm for the Software Release Planning problem is presented.

OVERALL INITIALIZATION
$COUNT = 1$

MAIN LOOP
REPEAT

MAIN LOOP INITIALIZATION
FOR ALL optional edges $(r_i, r_j) \in E_o$, $\tau_{ij} \leftarrow \tau_0$
FOR ALL vertices $r_i \in V$, $visited_i \leftarrow False$
FOR ALL vertices $r_i \in V$, $current\_planning_i \leftarrow 0$

SINGLE RELEASE PLANNING LOOP
FOR EACH Release, k
Place, using a roulette procedure, ant k in a vertex $r_i \in V$,
where $visited_i = False$ and $overall\_cost_i \leq budgetRelease_k$
ADDS (r_i, k)
WHILE $opt\_vis_k(i) \neq \emptyset$ **DO**
Move ant k to a vertex $r_j \in opt\_vis_k(i)$ with probability p_{ij}^k or considering $max(\tau_{ij} \cdot w_j^{\beta})$.
Update pheromone in edge (r_i, r_j), with $\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} + \varphi \cdot \tau_0$
ADDS (r_j, k)
$i \leftarrow j$

MAIN LOOP FINALIZATION
$best\_planning \leftarrow current\_planning$

$COUNT++$
UNTIL $COUNT > MAX\_COUNT$

RETURN $best\_planning$

// Besides r_i, adds to release k all of its dependent requirements, and, repeatedly, their dependent requirements
ADDS (r_i, k)
ENQUEUE (Q, r_i)
WHILE $Q \neq \emptyset$ **DO**
$r_s \leftarrow$ DEQUEUE (Q)
FOR EACH $r_t \in mand\_vis_k(s)$ **DO**
ENQUEUE (Q, r_t)
$visited_s \leftarrow True$
$current\_planning_s \leftarrow k$

Fig. 1. ACO Algorithm for the Software Release Planning problem

This algorithm can, this way, be applied to solve the Software Release Planning problem. Such algorithm shows one way the ACO framework can be adapted to such a problem, thus answering our research question RQ1., as discussed earlier.

5 Experimental Evaluation

This section describes all aspects related to the design of the experiments aimed at evaluating the performance of the proposed ACO algorithm. It first presents the data employed in the experiments, followed by a description of the other metaheuristics employed in the experiments and the metrics used in the comparison.

5.1 The Data

For the experiments, synthetic instances were generated with different values, - indicated in the parentheses -, of *number of requirements* (50, 200, 500), *number of releases* (5, 20, 50), *number of clients* (5, 20), *density of the precedence table* (0%, 20%) and *overall budget available* (80%, 120%). The density of the precedence table indicates the percentage of requirements which will have precedence. In that case, the requirement will have, as precedents, between 1 and 5% of other requirements, which is established randomly. The value of the overall budget available is computed by adding the overall cost of all requirements in that particular instance and calculating the indicated percentage of this cost. With that overall budget value, each release budget is determined by dividing this value by the number of releases in that instance.

All combinations of values for all aspects were considered, generating a total of 72 instances. Table 1 below describes a small subset of the synthetically generated instances used in the experiments. Each instance is indicated with a label "I_A.B.C.D.E", where A represents the number of requirements, B represents the number of releases, C, the number of clients, D, the density of the precedence table and E represents the overall budget available.

Table 1. Sample of the 72 Synthetically Generated Evaluation Instances

Instance Name	Instance Features				
	Number of Requirements	Number of Releases	Number of Clients	Precedence Density	Overall Budget
I_50.5.5.80	50	5	5	0%	80%
I_50.5.5.120	50	5	5	0%	120%
…	…	…	…	…	…
I_200.50.20.20.80	200	50	20	20%	80%
…	…	…	…	…	…
I_500.50.20.20.120	500	50	20	20%	120%

The values of $cost_i$, w_j and $risk_i$, for each requirement, were randomly generated using scales from 10 to 20, 1 to 10 and 1 to 5, respectively. In addition, the importance that each costumer set to each requirement was randomly generated from 0 to 5.

In order to facilitate replication, all instances used in the experiments were described and made publicly available for download at the paper supporting material webpage - published at http://www.larces.uece.br/~goes/rp/aco/ -, which contains, additionally, several extra results that could not be presented here due to space constraints.

5.2 The Algorithms, Their Configurations and Comparison Metrics

In order to compare the performance of the proposed ACO approach to the Software Release Planning problem, the metaheuristics Genetic Algorithm and Simulated Annealing were considered, as described next.

A. Genetic Algorithm (GA): it is a widely applied evolutionary algorithm, inspired by Darwin´s theory of natural selection, which simulates biological processes such as inheritance, mutation, crossover, and selection [16].
B. Simulated Annealing (SA): it is a procedure for solving arbitrary optimization problems based on an analogy with the annealing process in solids [17].

The particular configurations for each metaheuristic were empirically obtained through a comprehensive experimentation process. First, 10 of the 72 instances were randomly selected to participate in the configuration process. For each algorithm, different configurations were considered, varying the values of each algorithm´s parameters, as shown in the Tables 2, 3 and 4. For ACO, different values of α (1, 3, 10) and β (1, 3, 10) were analyzed. Since all combinations were considered, a total of 9 configuration instances were created for ACO (see Table 2). For the Simulated Annealing, initial temperature (20, 50, 200) and cooling rate (0,1%, 1%, 5%) were examined. Once again, 9 instances were generated (Table 3). Finally, for the Genetic Algorithm, a total of 27 configuration instances were produced, varying the values for the population size (20, 50, 100), crossover (60%, 80%, 95%) and mutation (0,1%, 1%, 5%) rates (Table 4).

Table 2. ACO Configuration Instances

Configuration Instance	Parameters	
	α	β
C_ACO:1.1	1	1
C_ACO:1.3	1	3
…	….	…
C_ACO:1.10	10	10

Table 3. SA Configuration Instances

Configuration Instance	Parameters	
	Initial Temperature	Cooling Rate
C_SA:20.01	20	0.1%
C_SA:20.1	20	1%
…	..	…
C_SA:200.5	200	5%

With those configuration instances, the configuration process was performed as follows: each algorithm was executed over all 10 problem instances considering, one at a time, each configuration instance in order to determine which configuration was more suitable to that particular instance. Next, the number of times each configuration instance performed the best was counted.

Table 4. GA Configuration Instances

Configuration Instance	Parameters		
	Population Size	Crossover Rate	Mutation Rate
C_GA:20.60.01	20	60%	0.01%
C_GA:20.60.1	20	60%	0.1%
…	…	…	…
C_GA:100.95.5	100	95%	5%

The selected configuration, for each algorithm, was the one which generated more frequently the best results over the 10 sample instances. Table 5 shows the selected configuration instances of each algorithm, which will be used in the comparison experiments described later on.

Table 5. Algorithms´ Configuration Instances

ACO	SA	GA
C_ACO:3.10	C_SA:50.01	C_GA:20.80.1

The paper supporting material webpage presents, with details, all results of this configuration process, showing, for each algorithm, which configuration instance performed best for each sample problem instance.

Additionally, for the Genetic Algorithm, a simple heuristic procedure was implemented to generate valid solutions to the initial population, and a single-point crossover and mutation operators were implemented, also producing only valid solutions. Here, binary tournament was used as selection method. Similarly, for the Simulated Annealing, a valid neighborhood operator was employed. For ACO, the normalization constant μ was set to 1, τ_0 to 10 and φ to 0.01.

In the experiments, this paper considers the following metrics to allow the comparison of the results generated by the different approaches: A) Quality: it relates to the quality of each generated solution, measured by the value of the objective function; B) Execution Time: it measures the required execution time of each strategy.

5.3 Experimental Results and Analyses

This section describes and discusses the results of the experiments carried out to compare the performance of the algorithms.

ACO (1k) x GA (1k) x SA (1k) (General Results)

The first set of experiments relates to the comparison of the algorithms using as stopping criterion the number of evaluated solutions. For that initial comparison, all algorithms were allowed to perform 1000 evaluations.

Table 6 below presents the results of a selected number of instances - 10 of 72 -, including the averages and standard deviations, over 10 executions, for the proposed ACO Algorithm - ACO (1k) -, Genetic Algorithm (GA) and Simulated Annealing (SA), regarding the quality of the generated solutions.

Table 6. Sample of the results, regarding the quality of the generated solutions and execution time (in milliseconds), for all algorithms executing 1000 evaluations, showing averages and standard deviations over 10 executions.

Instance		ACO (1k)	GA (1k)	SA (1k)
I_50.5.5.20.120	*quality*	11146.1±39.67	10869.8±182.51	9907.5±182.22
	exec. Time (ms)	1936.1±45.34	79.1±0.87	34.4±6.39
I_50.5.20.0.120		46161.4±253.83	44110.8±864.41	43777.9±751.19
		2194.1±14.68	123.2±0.78	46.8±0.42
I_50.20.20.20.80		93124±956.68	91606.7±5911.03	77506.7±3325.29
		1210±7.07	107.3±1.88	104.6±7.56
I_200.5.5.20.80		41953.8±101.28	37366.6±3455.83	29527.5±1139.68
		14723.5±277.3	623.1±57.44	447.7±7.39
I_200.5.5.20.120		43868.2±64.35	33205.7±2857.03	29718.1±765.27
		13579±181.42	626.9±28.17	400.9±10.49
I_200.5.20.20.120		147873.5±247.95	121931.6±4782.33	103197.6±4923.95
		13046.6±517.64	695.5±53.92	451±8.81
I_200.20.5.0.120		209290.1±396.24	157792.4±23691.97	158245.3±3364.83
		23665.6±301.9	919.2±38.9	460.2±13.33
I_200.20.20.0.120		501178.8±1682.89	442525±65014.99	428164.1±2801.25
		23012±33.43	1063.3±70.73	515.1±16.33
I_500.5.5.0.80		101377.6±317.28	95966±6044.53	83741.7±979.48
		136875±497.07	5376.9±274.98	2364.7±49.02
I_500.5.20.0.120		387081.6±157.18	342013.3±37472.1	331882.1±4308.78
		141801.3±1078.02	5551±314.3	2051.3±13.4

The complete description of all results, over all 72 instances, can be found in the paper supporting material webpage.

Overall, regarding the quality of the generated solutions, ACO performed better than GA and SA in all cases. Percentagewise, ACO generated solutions, in average, 78.27% better than those produced by GA and 96.77% than SA. Additionally, the expressively low standard deviations produced by the proposed approach demonstrates its stability, especially when compared to the results of the other algorithms.

However, in terms of execution time, ACO operated substantially slower than the other two metaheuristics when evaluation the same number of solutions. This may be attributed to the more complex constructive process performed by ACO in the building of each candidate solution. In average, for this metric, ACO required almost 60 times more than GA and more than 90 times more than SA.

ACO (1k) x GA (1k) x SA (1k) (Statistical Analyses)

In order to properly select the most appropriate statistical test, a normality test was performed over 20 samples - randomly chosen - generated throughout the whole experimentation process presented in this paper. The Shapiro-Wilk Normality Test was used for that purpose. The results suggest that normality should be accepted on 17 samples and rejected in the other 3 (the complete presentation of these tests can be

found in the paper supporting material webpage). Therefore, since normality can be assumed to all generated samples, the nonparametric Wilcoxon Rank Sum Test was used to evaluate statistical significance.

Thus, the statistical significance of the differences between the results generated by each pair of algorithms was calculated using the Wilcoxon Rank Sum Test, considering the significance levels of 90%, 95% and 99%. Under these conditions, for all instances, the results generated by ACO were significantly better than those generated by GA and SA, in all three significance levels, except for the instances shown in Tables 7.

Table 7. Instances in which statistical confidence cannot be assured when comparing the quality of the solutions generated by ACO with GA and SA, with all algorithms executing 1000 evaluations, calculated with the Wilcoxon Ranked Sum Test.

	90% confidence level	95% confidence level	99% confidence level
GA	I_50.5.5.20.80, I_50.5.20.0.80, I_50.20.20.0.120, I_50.20.20.20.80, I_200.5.20.0.80,I_200.5.20.0.120,, I_500.5.5.0.80,I_500.5.20.0.80	I_50.5.5.20.80, I_50.5.20.0.80 I_50.20.20.0.120,I_50.20.20.20.80, I_200.5.20.0.80,I_200.5.20.0.120, I_500.5.5.0.80,I_500.5.20.0.80	I_50.5.5.20.80, I_50.5.20.0.80, I_50.20.20.0.120,I_50.20.20.20.80, I_200.5.20.0.80,I_200.5.20.0.120, I_200.50.20.0.80,I_500.5.5.0.80, I_500.5.20.0.80,I_500.5.20.0.120, I_500.20.20.0.80
SA	-	-	-

As can be seen, only for 8 instances - out of 72 -, statistical significance could be assured under the 95% confidence level when comparing ACO with GA. For SA, even within the 99% level, ACO performed significantly better in all cases.

The complete set of results, with all measures generated by the Wilcoxon Rank Sum Test can be found in the paper supporting material webpage, for this, and all other statistical significance evaluations.

ACO (1k) x GA (10k) x SA (10k) (General Results)

To evaluate whether the better performance of the proposed ACO algorithm can be attributed to the relatively small number of evaluations performed by GA and SA, a new set of experiments where performed, this time by allowing both GA and SA to evaluate 10000 solution. A sample of the results for these experiments can be found in Table 7.

On these experiments, the ACO algorithm did better than GA in 69 out of the 72 instances. Only for instances I_50.5.5.20.80, I_50.5.20.0.80 and I_200.5.20.0.80, the Genetic Algorithm produced solutions with higher fitness value in average. The exact same behavior occurred with SA, which outperformed ACO over the same 3 instances.

Regarding execution time, ACO was still substantially slower than GA and SA. This time, however, ACO performed around 9 times slower than both GA and SA.

ACO (1k) x GA (10k) x SA (10k) (Statistical Analyses)

Considering initially only the 3 instances where both GA and SA performed better than ACO, the Wilcoxon Rank Sum Test produced the following p-values when

comparing ACO with GA: 0.1230000 for instance I_50.5.5.20.80, 0.0232300 for I_50.5.20.0.80 and 0.0015050 for I_200.5.20.0.80, and with SA: 0.0020890 for I_50.5.5.20.80, 0.0000217 for I_50.5.20.0.80 and 0.0000108 for I_200.5.20.0.80.

Table 8. Sample of the results, regarding the quality of the generated solutions and execution time (in milliseconds), with ACO executing 1000 evaluations, and GA and SA executing 10000 evaluations, showing averages and standard deviations over 10 executions.

Instance		ACO (1k)	GA (10k)	SA (10k)
I_50.5.5.20.120	*Quality*	11146.1±39.67	11038.7±197.86	10560.6±177.9
	exec. Time (ms)	1936.1±45.34	603.5±89.74	348.4±17.01
I_50.5.20.0.120		46161.4±253.83	45451.2±750.84	45605.4±352.43
		2194.1±14.68	1192±16.85	460.4±7.93
I_50.20.20.20.80		93124±956.68	91888.8±8920.19	83924.6±1604.32
		1210±7.07	1134.3±10.36	1028.4±71.77
I_200.5.5.20.80		41953.8±101.28	38398.7±3607.22	33473.9±895.87
		14723.5±277.3	7664.5±357.54	4978.5±88.11
I_200.5.5.20.120		43868.2±64.35	33471.4±1390.99	33951.4±2514.54
		13579±181.42	5514.5±225.92	4210±73.25
I_200.5.20.20.120		147873.5±247.95	134656.7±5846.67	124129.8±5285.18
		13046.6±517.64	7887.7±537.4	4841±131.76
I_200.20.5.0.120		209290.1±396.24	174431.4±8076.1	169316.3±2779.69
		23665.6±301.9	9760.1±65.68	6137.3±132.75
I_200.20.20.0.120		501178.8±1682.89	445685.5±15383.84	447483.6±6166.36
		23012±33.43	11255±89.76	7402.7±138.89
I_500.5.5.0.80		101377.6±317.28	100026.3±1221.57	95707±819.69
		136875±497.07	57316.1±284.37	31427.3±848.28
I_500.5.20.0.120		387081.6±157.18	367935±4991.23	375352.8±1660.25
		141801.3±1078.02	61106.9±1122.9	27498.4±315.2

Therefore, considering a confidence level of 95%, GA and SA had, respectively, two and three cases where they were able to produce significantly better solutions. In all other cases, ACO did significantly better, except for the instances presented below in Table 9.

Table 9. Instances in which statistical confidence cannot be assured when comparing the quality of the solutions generated by ACO with GA and SA, with ACO executing 1000 evaluations, and GA and SA executing 10000 evaluations, when ACO performed better, calculated with the Wilcoxon Ranked Sum Test.

	90% confidence level	95% confidence level	99% confidence level
GA	I_50.20.5.0.120,I_50.20.20.0.80, I_50.20.20.20.80,I_50.50.20.0.120, I_50.50.20.20.120,I_200.5.20.0.120 ,I_200.5.20.20.80,I_200.50.20.0.80, I_500.5.5.0.120, I_500.5.20.0.80	I_50.5.5.20.120,I_50.20.5.0.120, I_50.20.20.0.80,I_50.20.20.20.80, I_50.50.20.0.120,I_50.50.20.20.120 ,I_200.5.20.0.120,I_200.5.20.20.80, I_200.50.20.0.80,I_500.5.5.0.120, I_500.5.20.0.80	I_50.5.5.20.80,I_50.5.5.20.120, I_50.5.20.20.80,I_50.20.5.0.120, I_50.20.5.20.120,I_50.20.20.0.80, I_50.20.20.20.80,I_50.50.5.20.120, I_50.50.20.0.120,I_50.50.20.20.120, I_200.5.20.0.120,I_200.5.20.20.80, I_200.50.20.0.80,I_500.5.5.0.80, I_500.5.5.0.120,I_500.5.20.0.80
SA	I_50.5.20.20.120, I_500.5.20.0.80	I_50.5.20.20.120, I_500.5.20.0.80	I_50.5.20.20.80,I_50.5.20.20.120

As can be seen, under the 95% confidence level, ACO did significantly better than GA and SA in all 69 cases, expect for 11 and 1 instances, respectively.

ACO (Restricted Time) x GA (1k) x SA (1k) (General Results)

Even with the increased number of evaluation, both GA and SA performed much more time efficiently then ACO. In order to evaluate whether the increased performance of the proposed ACO approach can be attributed to the considerably high computations cost, a new set of experiments were designed. Here, the amount of time available to ACO was limited, to meet exactly the times required by GA and SA when evaluating 1000 solutions. The performance of ACO under this time restriction is sampled in Table 10.

Table 10. Sample of the results, regarding the quality of the generated solutions, with ACO with a time restriction, and GA and SA executing 1000 evaluations, showing averages and standard deviations over 10 executions.

Instance	ACO w/ Time GA (1k)	GA (1k)	ACO w/ Time SA (1k)	SA (1k)
I_50.5.5.20.120	11038.8±27.62	10869.8±182.51	11007.8±17.14	9907.5±182.22
I_50.5.20.0.120	45633.9±140.17	44110.8±864.41	45637±188.55	43777.9±751.19
I_50.20.20.20.80	91271.7±736.73	91606.7±5911.03	91469±951.19	77506.7±3325.29
I_200.5.5.20.80	41604.4±109.05	37366.6±3455.83	41525.4±93.41	29527.5±1139.68
I_200.5.5.20.120	43715.7±36.64	33205.7±2857.03	43681.8±34.35	29718.1±765.27
I_200.5.20.20.120	147478±351.54	121931.6±4782.33	147404.3±350.34	103197.6±4923.95
I_200.20.5.0.120	208845.6±290.21	157792.4±23691.97	209149.9±486.96	158245.3±3364.83
I_200.20.20.0.120	498434.1±1061.48	442525±65014.99	497747.4±1092.87	428164.1±2801.25
I_500.5.5.0.80	100880.4±216.46	95966±6044.53	100747±100.92	83741.7±979.48
I_500.5.20.0.120	386919.9±205.62	342013.3±37472.1	386785.2±135.83	331882.1±4308.78

Even with the time restriction, ACO continues to outperform both GA and SA, respectively, in 70 and 72 out of 72 cases. These results confirm the ability of the proposed ACO algorithm to find good solutions in little computational time.

ACO (Restricted Time) x GA (1k) x SA (1k) (Statistical Analyses)

When performing better, GA, over instances I_50.5.20.0.80 and I_50.20.20.20.80, could not obtain solutions significantly better than ACO (95% confidence level). However, over all other cases, under the 95% level, ACO did significantly better 63 times. Considering SA, ACO significantly outperformed this algorithm all but one case.

Since the ACO seemed to degrade very little with the time restrictions, a statistical analysis was performed to evaluate, with precision, this level of degradation. For that purposed, three correlation metrics (Pearson, Kendall and Spearman) where employed to estimate the correlation between the results generated by ACO before and after the time restriction. The results of these tests are shown in Table 11.

Table 11. Correlation metrics (Pearson, Kendall and Spearman) over the results generated by ACO before and after the time restriction

	Pearson Correlation	Kendall Correlation	Spearman Correlation
ACO (1k) vs ACO - Time GA (1k)	0.9999907	0.9929577	0.9993890
ACO (1k) vs ACO - Time SA (1k)	0.9999879	0.9866980	0.9987137

Therefore, these measurements demonstrate that those samples are highly correlated, indicating that ACO lost very little of its capacity when subjected to such time constraints. Additionally, using the fitness averages generated by the two versions (with and without time restriction) of ACO, the Wilcoxon Rank Sum Test was applied to measure the significance in the difference of these samples. The p-value 0.8183 was obtained when comparing ACO with its versions restricted by the time required by GA, and 0.7966, using the time used by SA. Those measurements confirm that there are no significant differences in the results produced by ACO before and after the time restrictions were employed.

Final Remarks

Therefore, all experimental results presented and discussed above indicate the ability of the proposed ACO approach for the Software Release Planning problem to generate precise solutions with very little computational effort relative to the results produced by Genetic Algorithm and Simulated Annealing. Thus, our research question RQ2., as described earlier, can be answered, pointing out to the competitive performance of the proposed approach, both in terms of accuracy as well as for time performance.

The main treat to the validity of the reported results is the fact that only synthetically generated data was considered, since no real-world data was available. Therefore, there is the possibility that some peculiarities of such real-world scenarios are not being considered.

6 Conclusion and Future Works

Very little has been done regarding the employment of the Ant Colony Optimization (ACO) framework to tackle software engineering problems modeled as optimization problems, which seems surprising given the large applicability and the significant results obtained by such approach.

This paper proposed a novel ACO-based approach for the Software Release Planning problem with the presence of dependent requirement. Experiments were designed to evaluate the effectiveness of the proposed approach compared to both the Genetic Algorithm and Simulated Annealing. All experimental results pointed out to the ability of the proposed ACO approach to generate precise solutions with very little computational effort.

As future works, the experiments could be extended to cover real-world instances, which should evoke other interesting insights as to the applicability of the proposed ACO algorithm to the Software Release Planning problem. Additionally, other types of dependencies among requirements (value-related, cost-related, and, or) could be considered.

References

1. Harman, M.: The Current State and Future of Search Based Software Engineering. In: Proc. of International Conference on Software Engineering / Future of Software Engineering 2007 (ICSE/FOSE 2007), pp. 342–357. IEEE Computer Society, Minneapolis (2007)
2. Dorigo, M., Stutzle, T.: The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances. In: Glover, F., Kochenberger, G. (eds.) Handbook of Metaheuristics, Norwell, MA (2002)
3. Dorigo, M., Maniezzo, V., Colorni, A.: The Ant System: Optimization by a Colony of Cooperating Agents. IEEE Trans. Systems, Man Cybernetics, Part B 26(1), 29–41 (1996)
4. Dorigo, M., Gambardella, L.M.: Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. IEEE Trans. Evolutionary Computation 1(1), 53–66 (1997)
5. Bianchi, L., Birattari, M., Chiarandini, M., Manfrin, M., Mastrolilli, M., Paquete, L., Rossi-Doria, O., Schiavinotto, T.: Metaheuristics for the vehicle routing problem with stochastic demands. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 450–460. Springer, Heidelberg (2004)
6. Gambardella, L.M., Dorigo, M.: Ant Colony System hybridized with a new local search for the sequential ordering problem. Informs. J. Comput. 12(3), 237 (2000)
7. Socha, K., Sampels, M., Manfrin, M.: Ant algorithms for the university course timetabling problem with regard to the state-of-the-art. In: Raidl, G.R., Cagnoni, S., Cardalda, J.J.R., Corne, D.W., Gottlieb, J., Guillot, A., Hart, E., Johnson, C.G., Marchiori, E., Meyer, J.-A., Middendorf, M. (eds.) EvoIASP 2003, EvoWorkshops 2003, EvoSTIM 2003, EvoROB/EvoRobot 2003, EvoCOP 2003, EvoBIO 2003, and EvoMUSART 2003. LNCS, vol. 2611, pp. 334–345. Springer, Heidelberg (2003)
8. Mahanti, P.K., Banerjee, S.: Automated Testing in Software Engineering: using Ant Colony and Self-Regulated Swarms. In: Proc. of the 17th IASTED International Conference on Modelling and Simulation (MS 2006), pp. 443–448. ACTA Press, Montreal (2006)
9. Chicano, F., Alba, E.: Ant Colony Optimization with Partial Order Reduction for Discovering Safety Property Violations in Concurrent Models. Information Processing Letters 106(6), 221–231 (2007)
10. del Sagrado, J., del Águila, I.M.: Ant Colony Optimization for requirement selection in incremental software development. In: Proc. of 1st International Symposioum on Search Based Software Engineering (SSBSE 2009), Cumberland Lodge, UK (2009), `http://www.ssbse.org/2009/fa/ssbse2009_submission_30.pdf` (fast abstracts)
11. del Sagrado, J., del Águila, I.M., Orellana, F.J.: Ant Colony Optimization for the Next Release Problem: A Comparative Study. In: Proc. of the 2nd International Symposium on Search Based Software Engineering (SSBSE 2010), Benevento, IT, pp. 67–76 (2010)
12. Karlsson, J., Olsson, S., Ryan, K.: Improved practical support for large-scale requirements prioritising. Requirements Engineering 2(1), 51–60 (1997)
13. Bagnall, A., Rayward-Smith, V., Whittley, I.: The next release problem. Information and Software Technology 43(8), 883–890 (2001)
14. Zhang, Y., Harman, M., Mansouri, S.A.: The multiobjective next release problem. In: Proc. of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1129–1137. ACM Press, New York (2007)

15. Greer, D., Ruhe, G.: Software release planning: an evolutionary and iterative approach. Information & Technology 46(4), 243–253 (2004)
16. Holland, J.: Adaptation in natural and artificial systems. Univ. of Michigan Press (1975)
17. Kirkpatrick, S., Gelatt, Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
18. Alaya, I., Solnon, G., Ghedira, K.: Ant algorithm for the multidimensional knapsack problem. In: Proc. of the International Conference on Bio-inspired Optimization Methods and their Applications (BIOMA 2004), pp. 63–72 (2004)
19. Leguizamon, G., Michalewicz, Z.: A new version of Ant System for Subset Problem. In: Congress on Evolutionary Computation, pp. 1459–1464 (1999)
20. Fidanova, S.: Evolutionary Algorithm for Multidimensional Knapsack Problem. In: PPSNVII- Workshop (2002)