Task Similarity-Based Task Allocation Approach in Multi-Agent Engineering Software Systems

YIFENG ZHOU1,2, CHAO FEI1 AND WANYUAN WANG1

1School of Computer Science and Engineering
2Research Center for Learning Science
Southeast University
Nanjing, 211189 P.R. China
E-mail: yfzhouseu@hotmail.com

Current complex engineering software systems are often made up of many software components to perform complex tasks, which can be modeled as multi-agent systems. Task allocation in complex multi-agent engineering software systems can be described through software agents’ cooperation to satisfy the resource requirement of tasks. Although many task allocation approaches have been presented to deal with this multi-agent task allocation problem, the similarity among tasks has not been paid much attention. Hence in this paper, we propose an efficient task similarity-based learning approach for task allocation in multi-agent software systems, which works by employing a Q-learning mechanism to improve the task execution utilities and using the similarity between historical tasks and new arriving tasks to avoid redundant calculation, thereby accelerating the allocation process. Through experiments, we conclude that our approach can yield the utility near to the optimal approach, which is better than benchmark task allocation approaches, and can reduce the computation load significantly compared to the optimal approach, allowing our approach to scale well to larger scale applications.

Keywords: complex engineering software system, multi-agent system, task allocation, task similarity, Q-learning

1. INTRODUCTION

Current complex engineering software systems are often made up of many software components to perform complex tasks, which can be modeled as multi-agent systems [1-6]. For example, a traffic monitoring system may include many distributed nodes which are autonomous and cooperative, and each node can be modeled as or controlled by a software agent [7]. Each software agent in the multi-agent software system owns a set of resources which can be contributed to a specific task that requires these resources [8]. When a task arrives at a multi-agent software system, the software agents can cooperate with other agents and coordinate their actions to provide the necessary resources required by this task [2, 5]. Once the task is accomplished, a predetermined utility attached to the task will be gained by the system [9].

The task allocation problem in multi-agent systems has been studied widely by many researchers [2, 9-15]. For example, Jiang et al. [2] design a contextual resource negotiation-based model with the aim of reducing the task execution time in the complex software systems; Weerdt et al. [9] present a distributed greedy algorithm to maximize the system
utility; and then to improve the system scalability. Zhang et al. [16] propose a distributed task allocation approach based on the multi-agent learning. However, if allocating tasks always from the scratch may lead to a large amount of redundant calculations when allocating the same or similar tasks. For this reason, we exploit the similarity between historical tasks and arriving tasks to avoid redundant calculation and allocate the arriving tasks according to the experience of historical tasks in order to gain a high utility.

The main contribution of this paper is to propose a task similarity-based learning approach for task allocation in multi-agent engineering software systems; the allocation experiences of historical tasks are utilized in the allocation of new arriving tasks. Employing this approach, a learning algorithm based on Q-learning [17, 18] is first used to record and update the allocation information of historical tasks, and then the task similarity between historical and arriving tasks is used to guide the allocate of the arriving tasks. Through experiments, our approach yields approximate utility as the optimal approach, which is much better than the benchmark approaches, and it also reduces the computation load of task allocation compared with the optimal approach, i.e., the allocation time can be significantly reduced which indicates the good scalability of our approach.

2. RELATED WORK

Many centralized algorithms have been proposed to resolve the task allocation problem in different types of environments [2, 12]. For example, Jiang et al. [2] propose a contextual resource negotiation-based task allocation method for complex software systems by allocating the tasks to the contextually rich agents that can access the resources from their contextual environment easily. This method can reduce the communication costs among agents and the total execution time of tasks. However this method needs to traverse all the agents in the system, thus the computation costs for allocating tasks will be large with the growth of the system’s scale. Page et al. [12] present a multi-heuristics evolutionary algorithm to minimize the total execution time in the heterogeneous system. Although it can get preferable efficiency compared with other benchmark models, the computation load of this model is significantly larger than other simple scheduling models which employ immediate mode heuristic schedulers.

Because of the heavy computation load for allocating tasks when using centralized algorithms, decentralized task allocation algorithms have been also studied in order to reduce the task allocation computation load [9, 16]. Weerdt et al. [9] propose a distributed algorithm for task allocation in social networks which allow agents to contribute their resources to the local beneficial tasks autonomously. However, this work can only fit in a simple cooperation scenario, i.e., the task can only be allocated to one agent and its direct neighbors. In contrast, we consider a task allocation scenario where tasks can be transferred among all the agents in the system, which is more realistic in multi-agent software systems. Zhang et al. [16] present a distributed resource allocation algorithm in cluster networks using multi-agent learning. The computation load of this model is low, but in some cases where task load is light, the utilities it gains cannot be as well as some benchmark models. Moreover, it has not taken the similarity between tasks into account, which can effectively reduce the computation load of task allocation.

Generally, these approaches probably face a problem that there will be a large amount of redundant calculation in task allocation when the system scale and the number
of arriving tasks are large, which degrades the performance of the system. Against this background, we propose a task similarity-based learning approach for task allocation which employs a Q-learning method to improve the task execution utilities and uses the similarity between historical tasks and new arriving tasks to avoid redundant calculation.

3. PROBLEM DESCRIPTION

3.1 Multi-Agent Engineering Software System

A complex engineering software system can be modeled as a multi-agent system, which is defined as follows. Assume that there are $M$ agents and $N$ types of resource in the system; let $a_i$ indicate the agent $i$, $ar_i$ indicate the resource vector of agent $a_i$, and $ar_{ij}$ express the amount of type-$j$ resources owned by agent $a_i$. Then, the multi-agent system is defined as follows according to [9]:

Definition 1 (Multi-agent system (MAS) [9]): A multi-agent system $MAS=(AG, E)$ can be abstracted into an undirected graph, where $AG$ is the set of agents and $E$ is the set of edges. Each edge $(a_i, a_j) \in E$ indicates that there is a physical connection between $a_i$ and $a_j$, and $a_i$ and $a_j$ can communicate through this edge. $e_{ij}$ is the distance between $a_i$ and $a_j$; if they are not connected directly, this value will be $\infty$; the distance between an agent and itself is 0, i.e., $e_{ii}=0$.

Fig. 1 presents an example of a multi-agent system. In this MAS, the set of agents $AG=\{a_1, a_2, a_3, a_4, a_5\}$, the set of edges $E=\{(a_1, a_2), (a_2, a_5), (a_1, a_3), (a_3, a_4), (a_4, a_5)\}$; thus $a_1$ can just communicate with $a_2$ and $a_3$; the distance between $a_1$ and $a_2$ ($e_{12}$), is 1 and the distance between $a_1$, $a_4$ ($e_{14}$) is $\infty$; and there are four types of resources owned by the agents in this MAS.

3.2 The Problem of Task Allocation

The task $t_i$ arriving at the system can be represented by a tuple $<util_i, req_i, loc_i>$ where $util_i$ is the utility to be obtained when this task is completed, $req_i$ is a vector of resources that is required by this task and $loc_i$ is the agent where the task is initially located, $loc_i \in AG$ [9]. Note that $req_i$ can be described as $<req_{i1}, req_{i2}, ..., req_{ik}, ..., req_{IN}>$ where $req_{ik}$ is the amount of type-$k$ resources required by the task $t_i$. 
When a task arrives at the system initially, it is called an original task. If the located agent can complete the arriving task independently, the task allocation process terminates; otherwise, this agent will finish a subtask of the original task according to the resources it owns and forward the remaining task to a selected neighbor. Then the neighbor agent allocates the arriving task in the same way. No matter a task is an original task or a subtask, the utility is proportional to the amount of its required resources. (The utility of an original task is the summation of its subtasks’ utilities.) Therefore, a valid task allocation can be formulated as Definition 2.

**Definition 2 (Valid task allocation):** A valid task allocation of an original task $t_i$ must satisfy the following conditions:

- Due to the limited resources of each agent, the task might be completed by a set of agents, and the involved agents must have enough amount of resources for this task. That is, if this task is executed by $a_1, a_2, ..., a_k$, we must ensure that $\sum_{i=1}^{k} r_{w_i} \geq req_{w}$ for each type-$w$ resources ($w\in\{1, 2, ..., N\}$). The task that agent $a_i, i\in\{1, 2, ..., k\}$ executes is a subtask of $t_i$.
- The MAS is distributed and cooperative, thus each agent will contribute all the resources it owns to the arriving tasks. Hence the task should be transferred among the agents and allocated hop by hop. If the original task is allocated to the agents $a_1, a_2, ..., a_k$, then it demands that there exists a path in MAS which connects these agents to make this task can be transferred, i.e., $(a_i, a_{i+1})\in E (i\in\{1, 2, ..., k-1\})$. This path is called the execution path of the original task and is indicated as $\{a_1, a_2, ..., a_k\}$.

Fig. 2 shows an example of a valid task allocation. In this case, the task $t_1$ can be executed along the execution path $\{a_1, a_2, a_5\}$, where the amount of resources provided by the three agents can satisfy the requirement of the task.

When a task is transferred from an agent $a_i$ to one of its neighbors such as $a_j$, it needs to take some communication cost to complete the transition. Then the net utility $u_{ip}$ when the original task $t_i$ is finished is $util_i$ excluding the total communication cost of the current execution path, which can be formulated as:

$$u_{ip} = util_i - \phi(p),$$  \hspace{1cm} (1)
\[
\varphi(p) = \sum_{i=1}^{k-1} c_{p_i p_{i+1}},
\]

where \( p = \{p_1, p_2, \ldots, p_k\} \) is the execution path of task \( t_i \), \( c_{p_ip_{i+1}} \) indicates the communication cost between \( p_i \) and \( p_{i+1} \), \( \varphi(p) \) is the total communication cost of \( p \).

In fact, a task may have several valid execution paths in a MAS. Among these valid execution paths, we expect to find out the one that has the highest utility in order to improve the utility of the system.

Then the multi-agent task allocation problem is defined in the following.

**Definition 3 (Multi-agent task allocation problem):** Given a \( \text{MAS} = (AG, E) \) and an original task \( t_i: \langle \text{util}_i, \text{req}_i, \text{loc}_i \rangle \) which arrives at agent \( a_i \), we expect to find out a valid execution path for this task which maximizes the net utility of \( \text{MAS} \), i.e.,

\[
\max \{u_{ip}\} = \max \{\text{util}_i - \varphi(p)\} = \max \{\text{util}_i - \sum_{i=1}^{k-1} c_{p_ip_{i+1}}\}.
\]

Subject to:

\[
p = \{p_1, p_2, \ldots, p_k\} \quad (1 \leq k \leq M),
\]

\[
p_1 = a_i,
\]

\[
p_i \in AG(1 \leq i \leq k),
\]

\[
(p_i, p_{i+1}) \in E(1 \leq i \leq k - 1),
\]

\[
\sum_{i=1}^{k} ar_{p_i w} \geq \text{req}_{iw} \quad (w = 1, 2, \ldots, N),
\]

In the case shown in Fig. 2, besides the execution path \( \{a_1, a_2, a_5\} \) introduced above, there is another execution path \( \{a_1, a_3, a_4, a_5\} \) which can also provide a valid task allocation but will result in a higher communication cost. Hence the execution path \( \{a_1, a_2, a_5\} \) is preferred, since it is able to maximize the utility of executing task \( t_i \).

Moreover, besides the objective of maximizing the net utility, the objective of reducing the computation load which indicates the allocation time in task allocation process is also considered, i.e., reducing the time for forwarding the remaining tasks. The proposed approach to satisfy the two objectives is introduced in Section 4.

## 4. TASK ALLOCATION MODEL

### 4.1 Task Allocation Mechanism

In our model, a learning algorithm based on Q-learning [17, 18] is adopted to manage the task allocation process: the required resource vector of a task (an original task or a task forwarded by others) is mapped to a state \( S \); an execution of a subtask by an agent can lead to a state transition \( F \); forwarding the remaining task can be called an action \( A \); and taking an action in a specific state can produce a corresponding Q value \( U \). Therefore, the task allocation process is described as a tuple \((S, A, F, U)\), where

- \( S \) is the state of the arriving task. When the task (an original task or a task forwarded by
others) arrives at agent $a_i$ at time $t$, state $s_t$ is the current resource vector of this task.

- $A=\{ac_1, ac_2, \ldots, ac_M\}$ is the set of all actions, and $ac_i$ is the set of $a_i$'s possible actions when it receives a task. As is mentioned in Section 3, when a task arrives at agent $a_i$, its selection is either to execute this task independently only if its resources can satisfy the resource requirement of this task (we use $ac_i = i$ to express this situation) or to forward this task to one of its available neighbors ($ac_i \neq i$). Available neighbors of agent $a_i$ for task $t$, are those who satisfy the following two conditions: 1) they are the neighbor of $a_i$; and 2) they are not included in the current execution path of $t$, i.e., an agent that has been allocated a task cannot receive its remaining task any longer. Therefore, the action set of agent $a_i$ is the set of agent $a_i$'s available neighbors in MAS.

- $F$ is the state transition function, which can be formulated as $s_{t+1}=F(s_t)=s_t- ar_i$ under the condition that at time $t$ the task is arriving at agent $a_i$. It means that the next state of the current task is the resource vector of the remaining task after agent $a_i$ accomplishes a subtask of the current task. When $s_0=\emptyset$, it indicates that this task is allocated successfully and there is no need to transfer this task.

- $U=\{U_1, U_2, \ldots, U_M\}$ is a function of $S$ and $A$, and $U_i$ is the function at agent $a_i$, which is the Q value (expected utility) of historical task allocation in Q-learning when the task is finished and action $A$ is implemented. This value is learned according to the history of task allocation and updated dynamically in the task allocation process. As to each agent such as $a_i$ and any state whose resource vector is equal to $req_i$, the initial value of $a_i$'s function $U_i$ is defined as below.

$$U_i(req_i, ac_i) = \begin{cases} 
0, & \text{if } ac_i \neq i \\
0, & \text{if } ac_i = i \text{ and } req_i > ar_i \\
\text{util}_i, & \text{if } ac_i = i \text{ and } req_i \leq ar_i 
\end{cases} \quad (9)$$

The $U$ value of any action that is not $i$ is 0 initially and updated dynamically with the proceeding of task allocation ($ac_i \neq i$). When $ac_i = i$ and $req_i > ar_i$, which means that the current agent $a_i$ can neither accomplish the current task nor forward the current task, then the $U$ value in this case must be 0. When $ac_i = i$ and $req_i \leq ar_i$ which means that the current agent $a_i$ can accomplish the current task independently and cannot forward the current task, then the $U$ value in this case is the utility of the current state, which is $\text{util}_i$.

Fig. 3 shows an example of a task allocation process in MAS. Each agent in the system maintains a value table which has 3 columns, $S$, $A$, and $U$. Each item which can be indicated by a vector $<S, A, U>$ means that the state set, the action set and the expected utility (Q value) under the condition of state $S$ and action $A$. The vector beside each agent indicates the agent’s resource vector and the value table that agent $a_1$ possesses. For example, the first item means that if a task whose required resource vector is $<4, 2, 0, 4>$ arrives at $a_1$ and action $a_2$ (represented by 2 in Q value table) is implemented then agent $a_3$ will finish a subtask whose required resource vector is $<3, 2, 0, 4>$, forward the remaining task $<1, 0, 0, 0>$ to $a_2$ and the expected utility (Q value) of this action is 9 (here for simplification, we assume that the utility for each required resource is 1 and the communication cost for each edge is 1).
Then the task similarity-based learning approach for task allocation can be described briefly as below. When an agent receives a task, it needs to execute and forward the task. If this agent can be capable of executing the task independently, there is no need to forward the task and a value table update process along the execution path of this task is required to record or update the information of this successful allocation; otherwise, it should execute a subtask, look up the value table and then select the action based on the predicted utility (see Definition 5), i.e., the larger an action’s predicted utility is, the higher possibility this action will be adopted. The detailed task allocation procedure is presented in Algorithm 1. The learning process of the value table and the task allocation process proceed synchronously. If a task arrives at the last agent in its execution path, there will be a value table update process inversely along the execution path, which is the role of function updateValueTable (introduced in Section 4.2) that can improve the net utilities; otherwise the current agent will look up the value table and select an action according to the task similarity, which is the role of function selectAndForward (introduced in Section 4.3) that can reduce the computation load.

Algorithm 1: Task allocation algorithm
/* ct_i: the current task arriving at agent a_i */
For each agent \( a_i \in AG \):
   If \( ct_i \neq \emptyset \), then:
      \( rt_i = ct_i - ar_i \);
      If \( rt_i = \emptyset \), then:
         Task allocation successfully;
         updateValueTable(\( ct_i \), i);
      Else: selectAndForward(\( ct_i \), i);

4.2 Value Table Learning

In order to describe the value table learning clearly, we give some important symbol definitions in Table 1. From the symbols in Table 1, we have \( at_t = t_{ij} + at_j(a_j = sup_{a_j}) \) in an execution path of the original task \( t_k \). Algorithm 2 shows how to learn the items in the value table recursively. Our model adopts a Q-learning method [17, 18] with backward tracking; each time when an original task is forwarded to the last agent in its current ex-
ecution path, the value table learning process begins. It will first update the Q value of the last agent’s value table and then turn to the update process of its superior’s value table in the same way until the agent where the original task is initially located.

### Table 1. Definitions of symbols.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>original(t)</td>
<td>A function that returns the original task of the task t</td>
</tr>
<tr>
<td>at&lt;sub&gt;i&lt;/sub&gt;</td>
<td>The arriving task when the original task &lt;i&gt;t&lt;/i&gt; arrives at or is forwarded to</td>
</tr>
<tr>
<td>vt&lt;sub&gt;i&lt;/sub&gt;</td>
<td>The value table of agent &lt;i&gt;a&lt;/i&gt;&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>size&lt;sub&gt;i&lt;/sub&gt;</td>
<td>The current size of vt&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>&lt;i&gt;t&lt;/i&gt;&lt;sub&gt;&lt;i&gt;k&lt;/i&gt;&lt;/sub&gt;</td>
<td>The subtask that agent &lt;i&gt;a&lt;/i&gt;&lt;sub&gt;&lt;i&gt;i&lt;/i&gt;&lt;/sub&gt; executes when it receives at&lt;sub&gt;&lt;i&gt;k&lt;/i&gt;&lt;/sub&gt;</td>
</tr>
<tr>
<td>rw&lt;sub&gt;&lt;i&gt;k&lt;/i&gt;&lt;/sub&gt;</td>
<td>The util value of &lt;i&gt;t&lt;/i&gt;&lt;sub&gt;&lt;i&gt;k&lt;/i&gt;&lt;/sub&gt;</td>
</tr>
<tr>
<td>sup&lt;sub&gt;&lt;i&gt;k&lt;/i&gt;&lt;/sub&gt;</td>
<td>Agent &lt;i&gt;a&lt;/i&gt;&lt;sub&gt;&lt;i&gt;i&lt;/i&gt;&lt;/sub&gt;’s superior who forwards at&lt;sub&gt;&lt;i&gt;k&lt;/i&gt;&lt;/sub&gt; to &lt;i&gt;a&lt;/i&gt;&lt;sub&gt;&lt;i&gt;i&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
<tr>
<td>c&lt;sub&gt;&lt;i&gt;ij&lt;/i&gt;&lt;/sub&gt;</td>
<td>The communication cost between agent &lt;i&gt;a&lt;/i&gt;&lt;sub&gt;&lt;i&gt;i&lt;/i&gt;&lt;/sub&gt; and &lt;i&gt;a&lt;/i&gt;&lt;sub&gt;&lt;i&gt;j&lt;/i&gt;&lt;/sub&gt;</td>
</tr>
<tr>
<td>( \delta )</td>
<td>The size threshold of the value table</td>
</tr>
</tbody>
</table>

### Algorithm 2: Value table learning algorithm: updateValueTable(<i>t</i>, <i>i</i>)

/* <i>t</i>: the arriving task at agent <i>a</i><sub><i>i</i></sub>, i.e., at<sub><i>k</i></sub>; \( \gamma \): the discount factor of Q-learning. */

1. \( t_k = \text{original}(t), a_j = \text{sup}_k, u = 0; \)
2. If \( a_j \neq \emptyset \), then:
   /* The original task is executed by more than one agent, which needs to update the value table. */
   1. For each item \( w: <s, a, u> \in vt_j; \)
      1. If \( t + t_k = s \) and \( i = a \), then: /* update the Q value */
         \[
         u = \max \{ u, rw_{kj} - c_{ij} + \gamma \cdot \max_{\text{maxSim}} U_i(t, \hat{c}) \}
         \]
        Break;
      1. If \( u = 0 \), then:
         \[
         u = rw_{kj} - c_{ij} + \gamma \cdot \max_{\text{maxSim}} U_i(t, \hat{c})
         \]
         insertOrUpdateItem(<i>j</i>, \( t + t_k <i>i</i>, \( i, u; \) /* at<sub><i>k</i></sub> = t + t_k */
         updateValueTable(\( t + t_k <i>i</i>, j); /* updateValueTable(at<sub><i>j</i></sub>, j) */

If there arrives a new task which is not ever inserted in the value table, then the size of the value table will increases and lead to a longer task allocation time because the traverse of all items in the value table is required when allocating tasks. Hence to avoid the unlimited increase of the value table’s size, an update strategy of the value table is designed, which is the role of function insertOrUpdateItem. When the size of value table is smaller than a given threshold, insert the new item directly; otherwise replace an existing item in the current value table which has the largest similarity (see Definition 4) with the new item and the lower expected utility (Q value), as detailed in Algorithm 3.

### Algorithm 3: Update strategy of the value table: insertOrUpdateItem(<i>j</i>, \( t, t, u' \))

/* <i>t</i>: the arriving task at agent <i>a</i><sub><i>j</i></sub>, i.e., at<sub><i>k</sub></i>; \( \text{maxSim} \): save the maximal similarity with <i>t</i>; \( \text{maxU} \): save the U value of the item which has the largest similarity with <i>t</i>; \( x \): save the item which has the largest similarity with <i>t</i>. */
If size, < δ, then: /* insert the item directly */ vt, j = vt, j ∪ < t, i, u’ >; Return;

Else:

For each item w: < s, a, u > ∈ vt, j;

If sim(s, t) > maxSim, then: /* get the item which has the maximal similarity */
maxSim = sim(s, t); maxU = u; x = w;

If u‘ ∙ maxSim > maxU, then: /* replace an existing item in the value table */
vt, j = vt, j – x;
vt, j = vt, j ∪ < t, i, u’ >;

4.3 Similarity-Based Task Forwarding

When an agent cannot finish the current task located on it independently, it should accomplish a subtask and forward the remaining task to a selected neighbor. We then propose a task forwarding mechanism which is based on task similarity as follows.

Definition 4 (Task similarity): Given two tasks t, x and t, y, whose required resources are <req, x1, req, x2, ..., req, xN> and <req, y1, req, y2, ..., req, yN> respectively, then the similarity between the two tasks can be denoted as:

\[ sim(t, x, t, y) = \frac{\min\{\|t, x\|, \|t, y\|\} \cdot |cos < t, x, t, y >|}{\max\{\|t, x\|, \|t, y\|\}} \]

where \( \|t, x\| \) and \( \|t, y\| \) are the norms of \( t, x \) and \( t, y \), respectively, and \( cos < t, x, t, y > \) is the cosine of the intersection angle between \( t, x \) and \( t, y \). Formally,

\[ \|t, x\| = \sqrt{\sum_{i=1}^{N} req, x_{i}^{2} (k \in \{x, y\})}, \]

\[ cos < t, x, t, y >= \frac{\sum_{i=1}^{N} (req, x_{i} \cdot req, y_{i})}{\|t, x\| \cdot \|t, y\|}, \]

Based on the definition above, it can be inferred that \( sim(t, x, t, y) \in [0, 1] \), and the more closely to 1 this value reaches, the more similar the two tasks will be. We give an example of the calculation of task similarity as follows. Let <4, 2, 0, 4>, <4, 1, 5, 3> be the vector of required resources of task \( t, x \) and \( t, y \). Then according to (11), the norms of the vectors are \( (4^{2} + 2^{2} + 0^{2} + 4^{2})^{0.5} = 6 \) and \( (4^{2} + 1^{2} + 5^{2} + 3^{2})^{0.5} = 7.14 \), respectively, and \( cos < t, x, t, y >= (4 \times 4 + 2 \times 2 + 0 \times 5 + 4 \times 3)/(6 \times 7.14) = 0.70 \). Then we can have \( sim(t, x, t, y) = (6/7.14) \times 0.70 = 0.59 \). The task similarity can be used to calculate the predicted utility of each action as defined in Definition 5 which is an important basis for forwarding the remaining task.

Definition 5 (Predicted utility): As to an item w: < s, a, u > in the value table of agent \( a_{i} \), if the arriving task is \( t_{i} \), then the predicted utility for w is:
pre(w, t_i) = u \cdot \text{sim}(s - ar_i, t_i - ar_i).

Algorithm 4 shows how to select an available neighbor to forward the remaining task. We calculate the predicted utilities of all possible actions according to the task similarity and select the action stochastically based on the predicted utilities, i.e., the larger an action’s predicted utility is, the higher possibility this action will be implemented. This is mainly because if an agent always takes the action whose predicted utility is the largest, the forwarding strategy will fall into the local convergence, i.e., there probably exists another action whose real predicted utility is larger than it but its expected utility is not learned sufficiently.

Algorithm 4: Task Forwarding Mechanism: selectAndForward(t, i)
/* maxPre_a: the maximal predicted utility when taking action a; p_a: the possibility of taking action a; k: a learning parameter. */
AN: all available neighbors of the current agent a_i;
For each item w: <s, a, u> \in vt_i and a \in AN:
    If pre(w, t_i) > maxPre_a, then:
        /*calculate the maximal predicted utility of each available action */
        maxPre_a = pre(w, t_i);
    For each action a \in AN: /*calculate the possibility of taking each action */
        p_a = \frac{k_{\text{maxPre}_a}}{\sum_{\bar{a} \in AN} k_{\text{maxPre}_{\bar{a}}}}
Forward the remaining task t-ar_i to the agent which the action a indicates with the probability p_a

In Fig. 3, the vector <5, 6, 3, 8> is the required resource vector of the arriving task at a_1, a_1 cannot execute it independently; thus it finishes a subtask <3, 2, 0, 4> and then selects a neighbor to forward the remaining task <2, 4, 3, 4>. The probability of taking action a_2 is higher than a_1 for a_1 because the predicted utility of action a_2 is larger than that of a_3. The arrows indicate the execution path with higher probability for the task.

The time complexity of our algorithm when allocating one task is $O(\delta \cdot \rho)$, where $\delta$ is the threshold for the size of the value table and $\rho$ represents the average number of agents needed by a task. Therefore, the allocation time per task cannot be influenced by the number of agents in MAS, allowing our approach to scale to large scale applications.

4.4 Detailed Example of Task Similarity-Based Task Allocation

In this section, we give an example in the following to demonstrate the detailed process of task similarity-based task allocation, which includes the task forwarding and the Q value table learning. The initial state of multi-agent software systems is shown as Fig. 4 (a). Each node represents a software agent and the vector beside each agent indicates the resource vector this agent owns. The number close to each edge indicates the communication cost through that edge. The table below each agent is the Q value table which is empty initially. In this example, a small task set \{t_1, t_2, t_3\} will be allocated. The threshold of Q value table’s size $\delta$ is set to 3, the learning parameter $k$ is set to 1.2 and the discount factor of Q learning $\gamma$ is set to 1.
Fig. 4. An detailed example of task similarity-based task allocation process for the task set \{t_1, t_2, t_3\}. Firstly, when task \( t_1 \) whose resource vector is \(<4, 2, 0, 4>\) arrives at agent \( a_1 \), \( a_1 \) cannot execute it independently and it will execute a subtask \(<3, 2, 0, 4>\), and then forward the remaining task \(<1, 0, 0, 0>\) to a selected neighbor. The available action is agent \( a_2 \) or \( a_3 \) (represented by 2 or 3 in Q value table) and the corresponding Q values are both 0 according to Eq. (9); hence, the maximal predicted utilities of action \( a_2 \) and \( a_3 \) are also both 0 according to Eq. (13). Therefore the possibility of taking each action is \( \frac{0.2^9}{(0.2^9+0.2^0)}=0.5 \). Assume action \( a_2 \) is implemented, then the remaining task \(<1, 0, 0, 0>\) arrives at agent \( a_2 \) and \( a_2 \) can execute it independently (see Fig. 4 (a)). Then the Q value table learning process begins along the execution path of \( t_1 \) reversely. Agent \( a_1 \) executes a subtask \(<3, 2, 0, 4>\) whose utility is 9 (Here we assume that the utility for each required resource is 1.) and adopts action \( a_2 \) whose communication cost is 3. The Q value of state \(<3, 2, 0, 4>\) and action 2 is \(9-3+1\cdot1=7\) according to Algorithm 2. Then a new item \(<<4, 2, 0, 4>, 2, 7>\) is inserted in the Q value table of \( a_1 \) (see Fig. 4 (b)).

Subsequently, when task \( t_2 \) whose resource vector is also \(<4, 2, 0, 4>\) arrives at agent \( a_1 \), we query the Q table of \( a_1 \) and find that the action \( a_2 \)'s maximal predicted utility is \(7\cdot \text{sim}(<4, 2, 0, 4>, <4, 2, 0, 4>)=7\) and the action \( a_3 \)'s maximal predicted utility is 0 according to Eq. (13). Therefore the possibility of taking action \( a_2 \) is \( \frac{1.2^7/(1.2^7+1.2^0)=0.54}{} \) and the possibility of taking action \( a_3 \) is 0.36. Here we assume the action \( a_3 \) is implemented, then the remaining task \(<1, 0, 0, 0>\) arrives at agent \( a_3 \), and this agent can execute it independently (see Fig. 4 (c)). Then a new item \(<<4, 2, 0, 4>, 3, 9-2+1\cdot1=8>\) is inserted in the Q value table of \( a_1 \) (see Fig. 4 (d)).
Finally, task $t_3$ whose resource vector is $<5, 6, 3, 8>$ arrives at agent $a_1$, $a_1$ will execute a subtask $<3, 2, 0, 4>$, and forward the remaining task $<2, 4, 3, 4>$ to a selected neighbor according to Algorithm 1. We query the Q value table of $a_1$ and find that the action $a_2$’ maximal predicted utility is $7 \cdot \text{sim}(<5, 6, 3, 8>, <4, 2, 0, 4>)=3.34$ and the action $a_3$’ maximal predicted utility is $8 \cdot \text{sim}(<5, 6, 3, 8>, <4, 2, 0, 4>)=3.82$ according to Eq. (13). Therefore the possibility of taking action $a_2$ is $1.2 \cdot 3.34 / (1.2 \cdot 3.34 + 1.2 \cdot 3.82) = 0.48$ and the possibility of taking action $a_3$ is 0.52. We assume the action $a_3$ is implemented, then the remaining task $<2, 4, 3, 4>$ arrives at agent $a_3$ who can only execute a subtask, $<2, 1, 3, 2>$, and $a_3$ will forward the remaining task, $<0, 3, 0, 2>$ to the sole neighbor $a_4$. Then $a_4$ will execute the task: $<0, 3, 0, 2>$ independently (see Fig. 4(e)). Once this task is allocated successfully, the update process of Q value table begins along the current execution path reversely. In the Q value table of $a_3$, a new item: $<<2, 4, 3, 4>, 5, 8-4+1\cdot 5=9>$ is inserted because the current Q value table is empty; and in the Q value table of $a_1$, a new item: $<<5, 6, 3, 8>, 3, 9-2+1\cdot 9=16>$ is then inserted (see Fig. 4(f)).

This example of task allocation also indicates the following facts: 1) employing the proposed task allocation approach, the arriving task can be allocated through an execution path which has a higher expected utility by considering the task allocation history, and also can have the probability to be allocated through a new execution path in order to seek a higher utility; 2) employing the proposed approach only need to traverse the Q value table when forwarding the remaining task, which can save much allocation time than exploring the existed execution paths in the MAS.

5. EXPERIMENTAL RESULTS

In this section, we conduct comprehensive experiments to investigate the performance of the proposed task allocation approach in multi-agent engineering software systems. The simulation platform of MAS is developed in Java using Eclipse IDE. The initial network of the MAS is constructed by random network model [19]; and there are four types of resources owned by agents and required by tasks. Without loss of generality, each task arrives at an agent randomly selected from the system.

In the experiments, four aspects of task allocation performance are considered, which are the effectiveness, convergence, scalability and complexity. The effectiveness test is to verify whether our model is effective for task allocation in multi-agent engineering software systems; the convergence test is to test whether the proposed learning algorithm can converge to a stable state which means that the average utility per task will become higher persistently as the task allocation process proceeds and become stable in the end; the scalability test is to test whether our model can fit well in systems with different scalability; and the complexity test is to verify the upper bound of the allocation time by employing our approach.

Four benchmark models are employed to compare with our model, which is optimal task allocation model, random task allocation model, self-owned resource-based task allocation model and contextual resource-based task allocation model. In the experiments, our model is abbreviated as QL model.

- Optimal task allocation model (OPT model): all available execution paths are traversed and the one which has the maximal net utility will be selected. For example in Fig. 4, if
the task: \( <5, 6, 3, 8> \) arrives at agent \( a_1 \), then the available execution paths are \{\( a_1, a_2, a_5 \)\} and \{\( a_1, a_3, a_5 \)\}. The net utility of \{\( a_1, a_2, a_5 \)\} is 18 and the net utility of \{\( a_1, a_3, a_5 \)\} is 16; thus the OPT model will allocate the task through \{\( a_1, a_2, a_5 \)\}.

- Random task allocation model (R model): the next forwarding agent is selected randomly among the available neighbors of the current agent. For example in Fig. 4, if the task: \( <5, 6, 3, 8> \) arrives at agent \( a_1 \), \( a_1 \) will select a random neighbor, \( a_2 \) or \( a_3 \), to forward the remaining task: \( <2, 4, 3, 4> \); and then the next hop agent will select a random neighbor if it cannot execute the arriving task independently.

- Self-owned resource-based task allocation model (SR model) [2]: the next forwarding agent is selected heuristically according to the amount of overlapped resources between the neighbors and the remaining task. For example in Fig. 4, if \( a_1 \) want to forward the remaining task \( <2, 4, 3, 4> \); then the overlapped resources of \( a_2 \) is \( <2, 4, 3, 4> \cap <2, 4, 1, 3> = <2, 4, 1, 3> \) and the overlapped resources of \( a_3 \) is \( <2, 4, 3, 4> \cap <2, 1, 3, 2> = <2, 1, 3, 2> \); thus \( a_2 \) will be selected by employing SR model, because of the more overlapped resources (10>8).

- Contextual resource-based task allocation model (CR model) [2]: the next forwarding agent is selected heuristically according to the resource enrichment of available neighbors’ contextual environment. For example in Fig. 4, if the task \( <5, 6, 3, 8> \) arrives at agent \( a_1 \), \( a_1 \) will considered not only the number of overlapped resources of the direct neighbors, but also the number of overlapped resources of the multi-hop neighbors and their distances. (Note that in the experiments, the SR and CR models in [2] have been modified to fit the task allocation environment considered in this paper.)

In the experiments, the threshold for the size of Q value table \( \delta \) is set to 2000 (Note that if this value is too large, the allocation time will become too high, whereas if this value is too small, it may be difficult to find an item in the Q value table which is sufficiently similar with the arriving task.), the learning parameter \( k \) in Algorithm 4 is set to 1.8 (Note that this value should be set larger than and close to 1. if this value is set too large, e.g., 3 or 4, the possibility of taking available actions will become too high which may lead to the local convergence of Q learning) and the discount factor of Q learning \( \gamma \) is set to 0.9 (which is a common setting in the implementation of Q learning [17, 18]).

### 5.1 Effectiveness

In the effectiveness test, we focus on the performance of our model on the average allocation time and the average net utility per task. The tasks’ arriving process is divided into different rounds. Each round there will be 500 tasks arriving at the system and the tasks from different rounds are the same. The average allocation time and average net utility per task are calculated among 30 rounds. We mainly focus on two parameters’ influence on task utilities and allocation time, which are the average task size and the average degree of MAS. The average task size here indicates the average required number of agents executing one task.

From the results in Figs. 5 and 6, we observe that: (1) from Fig. 5 (a), the average net utility of our model is the largest one except the OPT model and it can reach 80% of
OPT model no matter how large the average size of arriving tasks is; (2) the average net utility of SR model is always more than R model. It is because that SR model is considering the amount of overlapped resources between available neighbors and the remaining task which will lead to a lower communication cost; (3) the utility of CR model is not well enough because it is just adaptive to the systems where each resource type is concentrated on specific sections, but in our experiments the resource types are distributed randomly; (4) from Fig. 5 (b), the average allocation time per task of our model is a little more than R, SR and CR model, but far less than OPT model. With the growth of task average size, the average allocation time of OPT model is increasing sharply; (5) from Fig. 6 we can see that when the average degree of MAS is relatively low (less than 7), the average net utility using our model is higher than SR, CR and R model significantly. But once the average degree of MAS is more than 7, this superiority becomes unobvious. The potential reason is that when the average degree is large, there will be many excellent execution paths for each original task, which makes the scope of local information larger and then the local information-based models (SR and CR model) will perform close to global information-based model (OPT model). Hence our model can fit better for those multi-agent systems whose average degree is not too large.

In conclusion, our model can obtain a satisfying utility which approaches to that of OPT model and can have much lower allocation time meanwhile.
5.2 Convergence

In the convergence test, we also divide the task allocation process into different rounds as described in Section 5.1. The average net utility in each round is calculated separately. We run this convergence test in two kinds of multi-agent systems with different number of agents (50 and 100). Due to the great allocation time of OPT model when the number of agents and average task size in multi-agent systems are large, the utility of OPT model is not depicted in figures which is unpractical.

From the result in Figs. 7 (a) and (b), we observe that: (1) in the initial phase of task allocation, the average net utility of our model is not better than CR and SR model, which is because the learning of value tables is not sufficient, but as the round grows, the average net utility of our model exceeds R, SR and CR model gradually and in the end tends to be stable; (2) the average net utilities of CR and SR model are stable with the growth of rounds because these two models are determinate models without learning, which means that as to an original task no matter how many times it arrives at multi-agent systems, the allocation results of this task will be the same; (3) the average net utility of R model is the worst because it does not consider any local or global information of the multi-agent systems.

![Fig. 7. The average net utilities on the rounds. (a) The number of agents is 50. The average task size is 8. The average degree is 10; (b) The number of agents is 100. The average task size is 14. The average degree is 10.](image)

In conclusion, the proposed learning algorithm is a converged algorithm and by combining task similarity it can improve the net utility effectively with more rounds.

5.3 Scalability

In the scalability test, we investigate whether our model is feasible when the scale of multi-agent systems becomes large. The same sets of original tasks are imported in different multi-agent systems whose number of agents ranges from 50 to 500. The average net utility and allocation time are calculated among 15000 tasks.

From the result in Fig. 8, we can conclude that (1) from Fig. 8 (a), no matter how many agents there are in multi-agent systems, our model always gets a relatively higher
net utility than R, SR and CR model; (2) from Fig. 8 (b), the allocation time of SR, CR and R model is stable with the growth of scale, which is because the allocation time of these three models is just relative to the average degree of MAS; (3) the allocation time of our model is just relative to the average size of each agent’s value table, i.e., it cannot be influenced by the scale of multi-agent systems. Therefore the average allocation time per task of our model has an upper bound because the size of value tables must be less than a given threshold, which ensures the feasibility of our model in large-scale MAS. As a result of the same task number in different scales and there will exist a value table update process per successful task, then the larger the scale of multi-agent systems is, the smaller the average size of value tables is. Hence, with the same task number the allocation time per task of our model is decreasing with the growth of the system’s scale.

![Fig. 8. The average net utilities (a) and the average allocation time (b) in MAS with different numbers of agents.](image)

In conclusion, the allocation time per task of our model is decreasing with the growth of the scale if the task number is the same, meanwhile the average net utility of our model are always larger than other benchmark models no matter how large the MAS is. Hence our model can scale well to large-scale applications.

### 5.4 Complexity

In the complexity test, we investigate whether there is an upper bound of the allocation time of our model. A task flow including tasks with similar size arrives at MAS persistently. We calculate the average allocation time per 500 tasks as a round respectively.

From Fig. 9 we observe that: (1) in the initial phase of task allocation, the allocation time is short, since the size of value table is small; (2) the allocation time will become larger with the growth of rounds and finally in the end the allocation time will reach an upper bound. The main reason is that in the task forwarding process, each agent will look up its value table and select an action, thus the size of value table is very important to the allocation time. As the task allocation proceeds, the size of value table in each agent becomes larger because of learning. Once the size of a value table reaches its threshold, it cannot be larger, which ensure the existence of the upper bound of the allocation time.

In conclusion, the average allocation time of our model has an upper bound, which cannot increase limitlessly with the proceeding of learning in the task allocation process.
Fig. 9. The average allocation time on the rounds; The number of agents is 50; The average task size is 8; The average degree is 5.

6. CONCLUSIONS

In previous research on task allocation in multi-agent engineering software systems, little attention has been paid to the concept of task similarity between tasks, which actually can be used to reduce the redundant calculation of task allocation. Based on this motivation, this paper proposes a task similarity-based learning approach for task allocation in multi-agent software systems, which incorporate a learning algorithm in order to improve the system utility for task allocation.

To increase the system utility, the learning algorithm is proposed based on Q-learning, which can learn experiences from historical tasks to guide the allocation of following tasks. Experiments have verified that the system utility of our model reaches more than 80% of that of optimal task allocation model and is much more than other benchmark models significantly. On the other hand, to reduce the allocation time, task similarity is employed which allocates the arriving tasks according to their similarity with historical tasks. Experiments have proved that the average allocation time of our model can be much less than the optimal task allocation model.

Indeed, the proposed approach focuses on the environments with cooperative agents who will contribute all of their resources to the arriving tasks, and the influence of heterogeneous loads of agents on task allocation time has not been considered. Hence, there are some directions for future work:

- Load balancing is another typical objective of task allocation. Our model can be extended to achieve the objective of load balancing by considering the current load of actions in the calculation of the predicted utility.
- There may be also some selfish or deceptive agents in the MAS [20-22], i.e., such agents’ objective is to maximize their own utilities regardless of others. Therefore, we also expect to extend our model to fit the multi-agent software systems with selfish or deceptive agents in the future.

REFERENCES

1. D. Batory and S. O’Malley, “The design and implementation of hierarchical software systems with reusable components,” ACM Transactions on Software Engi-


---

**Yifeng Zhou** (周一峰) received his B.E. degree in Electronic Engineering from Southeast University, Nanjing, China, 2007. He is currently pursuing the Ph.D. degree at the Distributed Intelligence and Social Computing Laboratory, School of Computer Science and Engineering, and Research Center for Learning Science, Southeast University. His main research interests include multiagent systems and social networks.

---

**Chao Fei** (费超) received his B.E. degree in Computer Science and Technology from Southeast University, Nanjing, China, 2012. He is currently pursuing the M.E. degree at the Distributed Intelligence and Social Computing Laboratory, School of Computer Science and Engineering, Southeast University. His main research interests include task allocation and load balancing in multiagent systems.

---

**Wanyuan Wang** (王万元) received his B.S. degree in Information and Computing Science from Nanjing University of Aeronautics and Astronautics, Nanjing, China, 2011. He is currently pursuing the Ph.D. degree at the Distributed Intelligence and Social Computing Laboratory, School of Computer Science and Engineering, Southeast University. His main research interests include social networks and multiagent systems.