Optimization of Task Assignment to Collaborating Agents

Irfan Younas, Farzad Kamrani, Christian Schulte, Rassul Ayani
School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden
Email: {irfany, kamrani, cschulte, ayani}@kth.se

***Abstract*—The classic task assignment problem (AP) assigns m agents to n tasks, where each task is assigned to exactly one agent. This problem and many of its variations, including the case where a task is assigned to a group of agents working independently, have been discussed extensively in the literature. We consider a specific class of task assignment problems where each task is assigned to a group of collaborating agents that work as a team. Thus, changing one of the group members may have a vital impact on the output of the group. We assume that each agent has a set of capabilities and each task has certain requirements. The objective is to assign agents to teams such that the gain is maximized.**

We suggest a Genetic Algorithm (GA) for finding a near optimal solution to this class of task assignment problems. To the best of our knowledge, this class of APs has not been considered in the literature, probably due to the difficulty of evaluating the performance of a team of agents. Recently, we have developed a formal method for measuring performance of a team which is used in this paper to formulate the objective function of our GA. We analyze the quality of the obtained solution by comparing the result of our GA with (a) the exact solution of some smaller problems, and (b) with the results of the exact solution of specific cases that can be obtained by the Hungarian algorithm. We provide experimental results on efficiency, stability, robustness and scalability of the solution obtained by our GA.

Key words: Task Assignment Problem, Genetic Algorithms, Evolutionary Algorithms, Optimization

I. Introduction

The problem of optimally matching (assigning) the elements of two sets (usually called tasks and agents), where each matching may have a different weight (cost) is known as the *Assignment Problem* and hereafter will be referred to as the *AP*. The naive approach to solve the AP by comparing all possible assignments of tasks to agents is computationally infeasible due to combinatorial explosion. To illustrate the size of combinations that arises from a seemingly small problem, consider the number of combinations for assigning 4 tasks to teams of 3, 4, 6 and 7 agents respectively from a pool of m agents. (We have used this problem, where $m \in \{100, 200, \ldots, 800\}$ as a scalability test for the proposed solution, in section V-C). For $m = 100$, the number of combinations will be 2.495×10^{30} and for $m = 800$ this value will be 1.737×10^{49}. If we assume that calculating each new combination and comparing it with earlier combinations requires one FLOP (floating-point operation), it will take more than 26 million years to solve the problem for $m = 100$ using the most powerful (known) supercomputer of the day (2010). For $m = 800$, it would take 184 million billion years if we used a supercomputer that is one billion times more powerful than the most powerful supercomputer of the day.

The first solution to the AP, which has polynomial time complexity in the number of tasks was introduced by *Harold Kuhn* in [1]. The publication had a fundamental influence on combinatorial optimization [2], and the proposed method known as *Kuhn-Munkres algorithm* or *Hungarian method* has been widely adopted in the field.

In the original AP, each task is assigned to a different agent and each agent performs exactly one task, which implies the equality of the number of tasks and agents. However, in a slightly modified version of the problem, one can assume that the number of tasks and agents differ. This problem is readily converted to the basic AP by adding “dummy” tasks (or agents). The costs of “dummy” task-agent assignments are set to a value larger than the maximum cost of assignments (c_{max}) when the objective is to minimize the cost. A maximization problem can easily be converted to the basic AP by substituting each c_{ij} by $c_{max} - c_{ij}$.

Apart from these minor modifications of the AP, there are many other variations to the problem, which sometimes require completely different approaches. A golden anniversary survey on AP [3] provides a complete and comprehensive survey on the variations of the assignment problem. In this survey three main categories of AP are recognized and in each category several variations of AP are discussed.

1) Models with at most one task per agent.
2) Models with multiple tasks per agent.
3) Multi-dimensional assignment problems, which study the matching of members of three (or more) sets, e.g., the problem of matching jobs with workers and machines, or assigning students and teachers to classes and time slots.

Despite similarities, these problems have varying degrees of complexity. While some of them like the classic AP have a polynomial time solution, some others like Generalized Assignment Problem (GAP) are NP-hard combinatorial optimization problem [4] and require heuristic approaches. In this paper, we discuss the problem of assigning tasks to agents, with the objective of maximizing the total value added by agents to tasks. Generally, tasks may have different sizes and require different numbers of agents, while each agent is assigned to at most one task. Furthermore, it is assumed that agents performing a task constitute a team and the performance of the team is a possibly non-linear function of its members.

The problem as defined in this paper is distinguished from all types of discussed AP, in that there is an interaction between agents, which work in a team environment. However, from the classification point of view as described in [3], it belongs to the first category since each agent is involved in only one task although each task is assigned to several agents. Nevertheless, the defined problem is much more complex.

The outline of the rest of this paper is as follows. In Section II, a mathematical formulation of the problem is presented. In section III and IV, the GA heuristic and implementation details are discussed. Section V presents the experimental results and discusses accuracy, stability, robustness and scalability of our algorithm. Finally we conclude this paper in section VI.

II. Problem formulation

Let $\mathcal{A} = \{a_1, \ldots, a_m\}$ be the set of m agents and let $\mathcal{T} = \{t_1, \ldots, t_n\}$ be the set of n tasks, where generally $m \neq n$. Assume that each task $t_j \in \mathcal{T}$ requires a fixed number d_j of agents to be performed, while each agent $a_i \in \mathcal{A}$ is performing at most one task, implying $\sum_{j=1}^{n} d_j \leq m$. We denote the group of d_j agents performing task t_j by $g_{\mathcal{S}_j}$, where the index $\mathcal{S}_j$ is a set

$$\mathcal{S}_j \subset \{1, 2, \ldots m\}, \quad |\mathcal{S}_j| = d_j, \quad a_i \in g_{\mathcal{S}_j} \Leftrightarrow i \in \mathcal{S}_j.$$

The goal is to optimally (defined later) assign all tasks to groups of agents, that is to find subsets $\mathcal{S}_j$ for all t_j. Clearly, our assumption that each agent performs at most one task implies that these subsets are disjoint.

Moreover, we assume that agents in a group collaborate in a team environment and the value produced by a team is a possibly non-linear real function of its members and the task. That is, team $g_{\mathcal{S}_j}$, which performs task t_j produces the value $f(g_{\mathcal{S}_j}, t_j)$. Optimality is defined as maximizing the sum of values produced by agents, i.e. maximizing the objective function

$$u(\mathcal{S}_1, \ldots \mathcal{S}_n) = \sum_{j=1}^{n} f(g_{\mathcal{S}_j}, t_j). \tag{1}$$

We make the assumption that the value produced by a team $g_{\mathcal{S}_j}$ may be expressed as the sum of values produced by agents while they are influenced by the team

$$f(g_{\mathcal{S}_j}, t_j) = \sum_{i \in \mathcal{S}_j} v(a_i, g_{\mathcal{S}_j}, t_j). \tag{2}$$

Substituting equation 2 into equation 1, we obtain

$$u = \sum_{j=1}^{n} \sum_{i \in \mathcal{S}_j} v(a_i, g_{\mathcal{S}_j}, t_j). \tag{3}$$

One convenient way to express the problem is introducing the *assignment matrix* $\mathcal{X} = [x_{ij}]_{m \times n}$, where $x_{ij} = 1$ if task t_j is assigned to agent a_i and 0 otherwise. Using this notation, the objective function expressed by (3) can equivalently be reformulated as the following equation, which should be maximized

$$u(\mathcal{X}) = \sum_{j=1}^{n} \sum_{i=1}^{m} v(a_i, g_{\mathcal{S}_j}, t_j) x_{ij} \tag{4}$$

subject to constraints

$$\sum_{i=1}^{m} x_{ij} = d_j, \qquad \forall t_j \in \mathcal{T} \tag{5}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad \forall a_i \in \mathcal{A} \tag{6}$$

$$\sum_{j=1}^{n} d_j \leq m \tag{7}$$

$$d_j \geq 1, \qquad \forall t_j \in \mathcal{T} \tag{8}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall a_i \in \mathcal{A}, \forall t_j \in \mathcal{T} \tag{9}$$

where $\mathcal{S}_j = \{i : i \in \{1, \ldots m\}, x_{ij} = 1\}$. Constraint (5) ensures that each task is performed by a group of agents where each group $g_{\mathcal{S}_j}$ has d_j agents for task t_j. Constraint (6) means that no agent is assigned to more than one task. Constraint (7) ensures that the number of agents, which are part of n teams should be equal or less than the total number of agents and constraint (8) ensures that each group (team) should have one or more agents.

III. THE PROPOSED GA HEURISTIC

Genetic algorithms [5], [6] which are famous meta heuristics based on evolutionary ideas of natural selection and genetics were invented by John Holland [7] in 1960s. They simulate "survival of the fittest" principle which was laid down by Charles Darwin. The GAs have frequently been used for solving many optimization problems [8]–[11].

Similar to nature, genetic algorithms work with a population of individuals. Each potential solution is represented by a set of parameters known as genes. The parameters are combined to form a chromosome and each chromosome represents one candidate solution. The basic outline of a genetic algorithm can be stated as follows. First a random population of candidate solutions is generated. The next step is to evaluate the fitness of the individuals using some defined fitness function. Then a selection criteria is applied to choose some of the individuals. Various operations are then applied to the selected candidate solutions in order to produce new candidate solutions. Most common and widely used operations are crossover and mutation. Due to crossover operator, a new candidate solution inherits partial characteristics from its parent solutions. As Holland points out in [7] mutation prevents the loss of diversity. Mutation is helpful to traverse different regions of search space and thus escaping local minima/maxima.

The first step in designing a genetic algorithm for a particular problem is to devise a suitable representation. Our

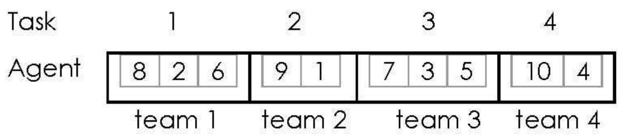

Fig. 1. Chromosome representation of a candidate solution

algorithm uses an appropriate representation scheme in which we have a n-dimensional vector of disjoint subsets to represent a task set (called chromosome in GAs literature), as shown in Fig. 1. In our case each task is performed by a team of agents and here the n-dimensional vector means that there are $\{1, 2, \dots n\}$ tasks which need to be performed by n teams of agents, where team j requires d_j agents. Thus the total number of agents required to perform the tasks will be $\sum_{j=1}^{n} d_j$. For instance, consider a case where we have four tasks requiring 3, 2, 3 and 2 agents respectively. Assume that there are 20 agents available. This means, $n = 4$, $d_1 = 3$, $d_2 = 2$, $d_3 = 3$, $d_4 = 2$ and $m = 20$. Figure 1 represents a possible assignment of agents to teams. The designed representation scheme ensures that all the constraints in (5), (6), (7) and (8) are automatically satisfied. The steps involved in our algorithm, almost similar to GA steps described by Chu et al. [8], are as follows:

1) Construct N candidate solutions to form an initial population. The initial population constitutes base solutions for successive generations. Each candidate solution (chromosome) is generated by randomly assigning agents to the tasks. Each agent is assigned to only one task and our initial population does not have duplicate candidate solutions.
2) Calculate the fitness value according to a given fitness function. In our problem the objective is to maximize the gain which is the fitness value for a candidate solution. It means the fitness value is the objective function value as given by equation (4). For each candidate solution in the population the fitness value is calculated and the list of chromosomes is sorted based on the fitness value. The candidate solutions (chromosomes) with higher fitness value are said to be fitter.
3) For reproduction, two candidate solutions are selected and these solutions are called parents. There are different schemes for selection but here we have used only binary tournament selection which was also used in [8]. The idea of binary tournament selection is to randomly choose two candidate solutions from the population. The fitness value for both selected solutions is compared and the individual with higher fitness value is selected for reproduction. For reproduction we need two parents and these parents are selected in the same manner using two binary tournaments.
4) A crossover operator is applied to the selected parents in order to generate a child solution. The proposed algorithm uses one-point crossover in which the crossover point, which is an integer $p \in \{1, 2, ..., \sum_{j=1}^{n} d_j\}$, is selected randomly. The generated child solution will take p genes from one parent and the remaining genes from the other parent. But this operation may make our candidate solution infeasible by violating constraint (6) which states that no agent can be assigned to more than one task. In order to make the solution feasible, the duplicate assignments are replaced by some other agent numbers which are not part of the chromosome. This operation changes the invalid child chromosome to a valid chromosome.
5) Due to the crossover operator the population tends to have similar characteristics after some generations because the child solutions tend to inherit almost all of their attributes from their parents. This means that individuals in the population can belong to some special part of the search space and we may miss some better solutions. In order to overcome this situation, the crossover and repair procedure is followed by a mutation procedure with a smaller probability value (20% for our experiments). For our problem we have used a simple procedure in which we randomly choose two genes and exchange their values. In this way we can traverse almost the entire search space and escape local maxima.
6) After crossover and mutation, the child solution need to be part of the population if it is not already found in the population. First of all the fitness value is calculated for the child chromosome and then the chromosome with the smallest fitness value in the population is replaced by the child. The population is then sorted based on fitness value. The replacement scheme helps to introduce new solutions in the population and eliminate those solutions which are weaker (have less fitness value).
7) The operations like selection, crossover, mutation and an individual replacement are performed repeatedly until $C_{non-duplicate}$ children have been generated without improving the best solution found so far.

IV. IMPLEMENTATION OVERVIEW

This section presents the key design decisions of our Java-based implementation of the GA. In particular, this section presents the representations of the data structures and outlines the algorithms and their asymptotic complexity.

a) Data structures: Each chromosome is implemented by a class storing the identity of its agents as a list and the fitness value (that is, the fitness value is computed only once for efficiency). The entire population is maintained as an array of chromosomes. In the following we refer by M to the size of a chromosome and by N to the size of the population.

Each field of the array of chromosomes is initialized by random shuffling of a list of the $\{1, \dots, m\}$ agent numbers (using the Java collections shuffle method) where duplicate chromosomes are avoided. The check for duplicate chromosomes checks all already generated chromosomes. We have decided to use a naive $O(MN^2)$ algorithm as it is only run during initialization.

b) Maintaining the population: The selection of parents for the crossover operation is by randomly choosing two chromosomes from the array (using a Java Random object) and choosing the fitter one (taking $O(1)$ time). The implementation of the crossover and repair operations is exactly as discussed in the previous section III and takes $O(M)$ time.

Finally, the best and worst chromosome (with respect to fitness) are maintained by a priority queue each (using simple binary heaps). Instead of entering chromosomes into the priority queues directly, we enter the chromosomes' positions (that is, integer values) into the priority queues. By that we can use the array of chromosomes for random parent selection as well as use the two priority queues for selecting and updating the best and worst chromosome.

Each iteration of the GA takes $O(\log N)$ for maintaining the best and worst chromosome in the population and $O(M)$ for checking that no duplicate chromosome is entered into the population. Hence, the time spent for each iteration is $O(\log N + M)$.

V. Experimental Results

In this section, we present different sets of experiments and their results. The first step towards any experiment is to employ a model to calculate the value produced by agents in a team in performing a task. We use the model introduced in [12] and [13], but modify it and take into the account the effect of team working. In this model each agent a_i has a set of p real-valued attributes, called *capabilities*, $c_i = \{c_{i1}, c_{i2}, \ldots, c_{ip}\}$ that affect the value produced by the agent. Each task t_j has a set of p real-valued attributes that we from now on refer to as *weights*, $w_j = \{w_{1j}, w_{2j}, \ldots, w_{pj}\}$, which give the importance of the capabilities of agents in performing the task. If a task is performed by a single agent then the performance or the value produced by the agent is defined by the weighted sum of the agents capabilities, that is

$$v(a_i, t_j) = \sum_{k=1}^{p} c_{ik} w_{kj}. \tag{10}$$

However, if a task is assigned to more than one agent, the produced value is more than the sum of values produced by individual agents. We assume that for each capability type, the capabilities of agents are influenced by the maximum capability of that type ($c_k^{\max}$) in the team ($g_{\mathcal{S}_j}$) and the new capabilities (c'_{ik}) are calculated by

$$c'_{ik} = c_{ik} + c_{ik}(c_k^{\max} - c_{ik})/c_k^{\max}, \tag{11}$$

where $c_k^{\max} = \max_{i \in \mathcal{S}_j} \{c_{ik}\}, \quad \forall k.$

Equation (11) implies the following.

1) In a team of agents only those having a smaller capability than the maximum capability benefit from the team working.
2) The capability of the agent who has the maximum capability is not affected by the cooperation.
3) The capability 0 of an agent is not affected by the team working.
4) Agents that have a capability equal to $c_k^{\max}/2$ receive the highest benefit from the team working.

Equations (10) and (11) together yield the value produced by an agent in a team when collaborating with other agents

$$v(a_i, g_{\mathcal{S}_j}, t_j) = \sum_{k=1}^{p} (c_{ik} + c_{ik}(c_k^{\max} - c_{ik})/c_k^{\max}) w_{kj}. \tag{12}$$

Substituting equation (12) into the objective function defined by equation 4, we obtain

$$u(\mathcal{X}) = \sum_{j=1}^{n} \sum_{i=1}^{m} \sum_{k=1}^{p} (c_{ik} + c_{ik}(c_k^{\max} - c_{ik})/c_k^{\max}) w_{kj} x_{ij}, \tag{13}$$

where $c_k^{\max}(\mathcal{X}) = \max_{1 \le i \le m} \{c_{ik} x_{ij}\}$
subject to the constraints (5) to (9).

To test our algorithm by using the above model, a series of experiments with different problem sizes are conducted. In this scenario, each agent a_i has 4 capabilities $\{c_{i1}, c_{i2}, c_{i3}, c_{i4}\}$ and each task t_j weights capability c_{ik} by $w_{kj} \in \{w_{1j}, w_{2j}, w_{3j}, w_{4j}\}$. All values c_{ik} and w_{kj} are randomly chosen from a uniform distribution between 0 and 4 (multiples of 0.5). For conducting different experiments, we have formulated different problems as shown in Table I. We can relate these problems to real world applications. Consider a human resource allocation problem where a human resource manager of company X has to recruit human resource teams to be assigned to given number of tasks. The objective is to assign human resources to different teams in a manner such that the performance is maximized. Problem # 9 in Table I represents such a human resource allocation problem, where there are total 50 agents (human resources) and the objective is to assign 20 agents to 4 teams such that the gain is maximized. The number of agents in the teams are 3, 4, 6 and 7 respectively.

The algorithm presented was coded in Java and was run on a PC with a 3.16 GHz CPU and 3.49 GB of RAM. In our experiments, 5 trials of the GA heuristic were performed for each problem. The population size is set to 800 and each trial is terminated when $C_{non-duplicate} = 20000$ children have been generated without improving the best solution found. The selection of these input parameters (population size and number of iterations) depends on the type of the underlying application and the size of the problem. There is a tradeoff between quality and execution time. For example, in case of real time applications, the execution time may be more important than the quality of the solution. In this case the number of iterations for termination should be smaller. Different problem sizes and domain complexities can assume different optimal settings for these parameters. In this paper our focus is not to find the optimal parameter settings. As we are not dealing with real time applications and our focus is to obtain high-quality solutions within reasonable execution time, we assume larger number of iterations and population size as given above.

Below, we analyze the experimental results and discuss the accuracy, stability, scalability and robustness of our proposed solution.

TABLE I
SPECIFICATION OF THE ASSIGNMENT PROBLEMS USED IN THIS PAPER

Prob#	Total Agents	Required Agents	Number of Tasks	Teams (# of agents assigned to the tasks)
1	10	10	4	[2, 3, 2, 3]
2	20	20	4	[3, 4, 6, 7]
3	20	20	8	[3, 2, 4, 3, 2, 2, 2, 2]
4	30	30	8	[4, 5, 2, 3, 4, 6, 3, 3]
5	30	30	12	[2, 2, 3, 3, 1, 4, 5, 2, 3, 2, 2, 1]
6	60	60	12	[6, 4, 8, 2, 7, 3, 5, 5, 3, 7, 1, 9]
7	80	80	20	[5, 4, 8, 2, 3, 3, 4, 3, 4, 7, 1, 6, 3, 6, 2, 6, 3, 1, 5, 4]
8	96	96	30	[5, 4, 8, 2, 3, 2, 2, 3, 1, 4, 7, 1, 5, 3, 2, 6, 2, 6, 3, 1, 5, 4, 2, 2, 4, 2, 1, 2, 3, 1]
9	50	20	4	[3, 4, 6, 7]
10	100	20	4	[3, 4, 6, 7]
11	200	20	4	[3, 4, 6, 7]
12	300	20	4	[3, 4, 6, 7]
13	400	20	4	[3, 4, 6, 7]
14	500	20	4	[3, 4, 6, 7]
15	600	20	4	[3, 4, 6, 7]
16	700	20	4	[3, 4, 6, 7]
17	800	20	4	[3, 4, 6, 7]

TABLE II
COMPUTATIONAL RESULTS FOR AGENTS WORKING IN TEAMS WITHOUT INTERACTION (ACCURACY TEST)

Prob #	Solution in each of the 5 trials					Avg. Execution Time (s)	Best Overall soln.	Hungarian algo results	$avgDev$ (%)	σ (%)
1	102.06	102.06	102.06	102.06	102.06	0.26	102.06	102.06	0.0	0.0
2	216.62	216.62	216.62	216.62	216.62	0.24	216.62	216.62	0.0	0.0
3	223.88	223.88	223.88	223.88	223.88	0.39	223.88	223.88	0.0	0.0
4	317.5	317.5	317.5	317.5	317.5	0.64	317.5	317.5	0.0	0.0
5	319.31	319.12	319.31	319.31	319.31	0.75	319.31	319.31	0.01	0.02
6	629.19	627.62	629.06	630.0	630.19	2.49	630.19	630.25	0.16	0.14
7	866.06	868.56	868.38	867.12	868.25	4.92	868.56	869.06	0.16	0.11
8	1043.88	1045.56	1045.5	1046.81	1043.94	8.33	1046.81	1047.56	0.23	0.11

$avgDev$ (%) : Average percentage deviation from the optimal solution (optimal obtained by Hungarian algorithm)

A. Accuracy

For testing the accuracy of the algorithm and to verify the quality of the results, we consider the scenario where agents work independently in teams and have no interaction. We compare our results with the optimal solutions obtained by the well-known Hungarian algorithm. The results for this scenario have been shown in Table II. Comparison of the results shows that the best solutions obtained by our algorithm are almost consistent with the base line results of Hungarian algorithm. Moreover, the average percentage deviation is almost 0.0 for smaller and medium problems.

In this paper we assume that agents in a group collaborate in a team environment and the value produced by a team is a non-linear real function of its members and the task. Agents participating in the same team affect the capabilities of each other. The agents having higher capability values help to boost objective gain by cooperating with other team members with lower capability values. In this scenario, Hungarian algorithm cannot be applied. To test the accuracy of our algorithm we have considered a simple data set for agents and tasks as given in Table III and IV respectively. The data set given in these tables is designed in such way that we can easily find the optimal solution and verify the correctness of our algorithm. We have considered problem number 1 where we have 4 tasks and 4 teams of agents. The number of agents in the teams are 2, 3, 2 and 3 respectively. Observing the data given for agents capabilities and tasks attributes, we can see that for each task exactly one attribute has the highest weight 4.0, and similarly one capability for each agent has the highest value 4.0.

For example, for task number 1, weight for the first attribute is 4.0 while the remaining weights are 1.0. We can easily conclude that the gain will be maximum if this task is assigned to those agents, which have highest values for capability 1. From Table III, we find 2 such agents namely agent number

TABLE III
DATA SET FOR AGENTS ATTRIBUTES (FOR ACCURACY TEST)

Agent#	Capabilities values			
	c_1	c_2	c_3	c_4
1	4.0	1.0	1.0	1.0
2	1.0	4.0	1.0	1.0
3	4.0	1.0	1.0	1.0
4	1.0	1.0	4.0	1.0
5	1.0	4.0	1.0	1.0
6	1.0	1.0	1.0	4.0
7	1.0	4.0	1.0	1.0
8	1.0	1.0	4.0	1.0
9	1.0	1.0	1.0	4.0
10	1.0	1.0	1.0	4.0

TABLE V
COMPARING GA RESULTS WITH THE OPTIMAL ONE (ACCURACY TEST)

Prob #	Solution in each of the 5 trials (using GA Heuristic)					Avg. Execution Time(s)	Agents Assignment (obtained by GA)	Optimal Solution value (calculated manually)	Optimal Assignment (calculated manually)
1	190.0	190.0	190.0	190.0	190.0	0.28	$g_{S_1} = \{3,1\}$ $g_{S_2} = \{7,5,2\}$ $g_{S_3} = \{8,4\}$ $g_{S_4} = \{6,10,9\}$	190.0	$g_{S_1} = \{1,3\}$ $g_{S_2} = \{2,5,7\}$ $g_{S_3} = \{4,8\}$ $g_{S_4} = \{6,9,10\}$

g_{S_j} : group of d_j agents performing task t_j

TABLE VI
COMPUTATIONAL RESULTS FOR AGENTS WORKING IN TEAMS COOPERATING WITH EACH OTHER (STABILITY TEST)

Prob #	Solution in each of the 5 trials					Avg. Execution Time (s)	Best Overall soln.	$avgDev$ (%)	σ (%)
1	118.11	118.11	118.11	118.11	118.11	0.34	118.11	0.0	0.0
2	260.1	260.1	260.1	260.1	260.1	0.3	260.1	0.0	0.0
3	253.55	253.82	253.82	253.82	253.82	0.47	253.82	0.02	0.04
4	369.96	370.22	370.9	370.9	370.9	0.86	370.9	0.09	0.11
5	364.39	364.11	364.12	363.67	363.34	0.96	364.39	0.13	0.1
6	753.4	752.99	751.69	751.89	751.94	3.21	753.4	0.14	0.09
7	1023.01	1025.19	1023.24	1027.47	1024.89	7.16	1027.47	0.26	0.16
8	1212.74	1215.27	1214.44	1217.29	1215.19	11.06	1217.29	0.19	0.12
9	302.41	302.17	302.35	302.41	302.29	0.7	302.41	0.03	0.03
10	308.22	308.68	307.94	308.68	308.76	0.72	308.76	0.1	0.1
11	312.56	312.14	312.14	312.32	312.35	0.82	312.56	0.08	0.05
12	313.7	314.02	313.73	314.21	314.21	0.8	314.21	0.08	0.07
13	314.94	315.09	314.94	315.09	315.09	1.02	315.09	0.02	0.02
14	316.0	315.51	315.89	316.04	315.59	0.94	316.04	0.07	0.07
15	316.09	316.04	316.11	316.02	315.94	1.02	316.11	0.02	0.02
16	316.16	316.02	315.99	316.06	316.16	1.14	316.16	0.03	0.02
17	316.33	316.29	316.29	316.31	316.22	1.26	316.33	0.01	0.01

$avgDev$ (%) : Average percentage deviation from overall best solution in all trials

1 and 3. Similarly, we can find optimal assignments for other tasks. We can verify the correctness of our algorithm by considering the results given in Table V. The results obtained by executing our proposed GA are consistent with the optimal values determined manually.

B. Stability

The results for some scenarios where agents work in teams collaborating with each other have been shown in Table VI. For each problem, the solution value for each of the 5 trials, along the average execution time (in seconds) are given in the table.

To check the stability of our algorithm, the quality of a solution in all 5 executions is evaluated as an average percentage deviation named $avgDev$ with respect to the best solution S_b of all trials, by using standard deviation σ. The percentage deviation $avgDev$ is defined as $avgDev = 1/5 \sum_{i=1}^{5} dev_i$, where $dev_i = ((S_b - S_i)/S_b) \times 100$ and S_i is the solution value of the i−th trial. The standard deviation σ of dev_i is calculated by formula $\sigma = \sqrt{1/5 \sum_{i=1}^{5} (dev_i - avgDev)^2}$.

The average deviations and the variance indicate that the result of different trials are very close to each other. This is a measure of stability of the obtained solutions.

TABLE IV
DATA SET FOR TASKS ATTRIBUTES (FOR ACCURACY TEST)

Task#	Weights for attributes			
	w_1	w_2	w_3	w_4
1	4.0	1.0	1.0	1.0
2	1.0	4.0	1.0	1.0
3	1.0	1.0	4.0	1.0
4	1.0	1.0	1.0	4.0

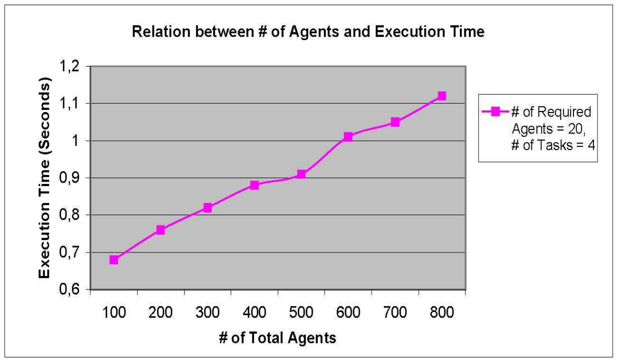

Fig. 2. How problem size affects the execution time (Scalability)

TABLE VII

COMPUTATIONAL RESULTS FOR AGENTS WORKING IN TEAMS COOPERATING WITH EACH OTHER (ROBUSTNESS TEST)

	Uniform Distribution (for agents capabilities): Uniformly distributed on $[0, 4]$			Non-Uniform Distribution (for agents capabilities): 2% agents have value 4.0; 2% have value 3.5; 96% Uniformly distributed on $[0, 3]$		
Prob #	$avgDev$ (%)	σ (%)	Avg. Execution Time (s)	$avgDev$ (%)	σ (%)	Avg. Execution Time (s)
1	0.0	0.0	0.34	0.0	0.0	0.29
2	0.0	0.0	0.3	0.0	0.0	0.33
3	0.02	0.04	0.47	0.0	0.0	0.58
4	0.09	0.11	0.86	0.27	0.22	0.82
5	0.13	0.1	0.96	0.11	0.11	1.0
6	0.14	0.09	3.21	0.11	0.18	2.98
7	0.26	0.16	7.16	0.17	0.16	7.13
8	0.19	0.12	11.06	0.14	0.13	9.48
9	0.03	0.03	0.7	0.05	0.08	0.62
10	0.1	0.1	0.72	0.13	0.19	0.68
11	0.08	0.05	0.82	0.09	0.12	0.76
12	0.08	0.07	0.8	0.07	0.1	0.82
13	0.02	0.02	1.02	0.09	0.06	0.88
14	0.07	0.07	0.94	0.17	0.16	0.91
15	0.02	0.02	1.02	0.07	0.15	1.01
16	0.03	0.02	1.14	0.03	0.04	1.05
17	0.01	0.01	1.26	0.03	0.05	1.12

$avgDev$ (%) : Average percentage deviation from overall best solution in 5 trials

C. Scalability

After testing the accuracy and stability of our GA, we want to observe the scalability of our proposed solution. An algorithm design is said to scale if it does not fail and performs efficiently even if applied to larger problem sizes. In this section we show that our proposed solution can solve the larger problem instances by a small increase in execution time (increase in execution time is the expected behavior). The execution time does not increase as quickly as the size of the problem. In order to show how the problem size affects the efficiency of our algorithm, we show in Figure 2 how the increase in number of total agents affects the execution time of our algorithm. We want to analyze the effect of total number of agents on execution time while the required number of agents and number of tasks are kept constant. An increase in the total number of agents results in an increase in the size of the search space. We select problems 10 to 17 as given in Table I where we need to assign 20 agents to 4 teams. If we analyze the figure, the execution time is 0.68, 0.76, 0.88 and 1.12 seconds for 100, 200, 400 and 800 agents respectively. We can see from the shape of the graph that the execution time is increased from 0.68 to 1.12 (by less than 65%) when the number of agents is increased by a factor of 8 (from 100 to 800). By looking at the behavior of the execution time, we can say that our algorithm is scalable to larger problem sizes.

D. Robustness

In this section we show that our proposed solution is able to cope with different types of input data. Input data belonging to different classes of distributions can introduce varying degree of complexities in the input search space. In order to test the effect of input data on the performance of our algorithm, we perform experiments on two different distributions of agent's capabilities as given in Table VII. The left portion of the table shows results for the input data where capabilities of agents are randomly chosen from a uniform distribution between 0 and 4. On right side of the table, we consider non-uniform distribution in which for each capability, 4% agents have values 3.5 and 4.0 (2% each) while the rest 96% are randomly chosen from a uniform distribution between 0 and 3. The average percentage deviation named $avgDev$, standard deviation σ and average execution time for all experiments are given in the table. The average percentage deviation $avgDev$ and standard deviation σ are calculated according to the formula given in subsection V-B. The results given in Table VII show small deviation from the best solution (less than 0.28% for all problems) and the average execution time behavior is also almost same in both distributions. By observing the quality of results we can say that our GA is robust to input data and it perform very well for other families of input data. Here, by robustness we mean that the performance of our algorithm does not degrade when tested on different distributions of input data.

E. Impact of Number of Iterations (Termination Criteria)

Figure 3 shows the effect of number of iterations (the termination criteria of the proposed GA) of experiments on percentage average deviation of solutions from best solution (The best solution is obtained with 20,000 iterations). We can see from the shape of the graph that increase in number of iterations significantly improves the quality of the solutions. As we increase the number of iterations of the experiment, the solution values deviate less from best solution value. Similarly, if we see the execution time of the algorithm in Figure 4, it increases more rapidly between 1000 and 3000 iterations. After this point, the increase in execution time is smaller

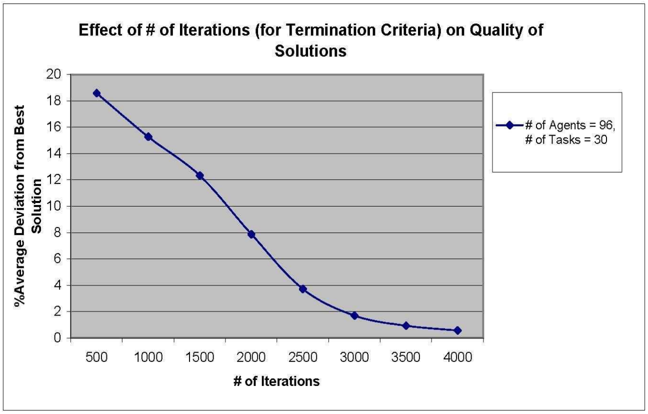

Fig. 3. The effect of number of iterations (for termination) on quality of solutions

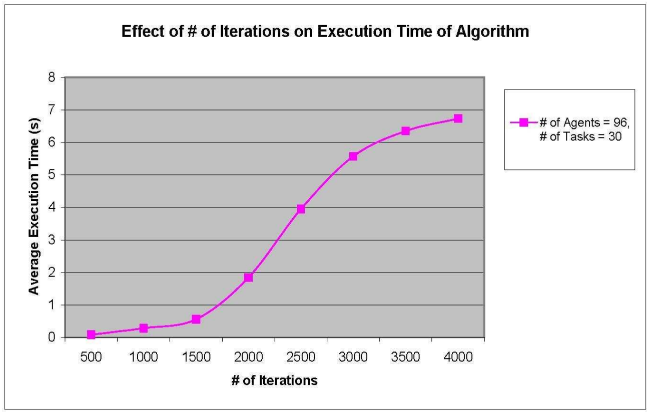

Fig. 4. The effect of number of iterations (for termination) on execution time of the algorithm

because of smaller deviation of the solutions. The shapes of these two figures show that the quality of the solutions improves at the expense of execution time. For example if we see the figures 3 and 4, after 4000 iterations the average standard deviation is 0.56% and the execution time is 6.73 seconds. If we run it for $20,000$ iterations then we can further improve the quality of the solutions. We can see from Table VI for the problem number 8 that the average percentage deviation of the 5 solutions from the best solution (Best solution value = 1217.29, obtained after $20,000$ iterations) is 0.19% and the execution time is 11.06 seconds after $20,000$ iterations. The decrease in average percentage deviation from 0.56% to 0.19% increases the execution time from 6.73 to 11.06 seconds (almost 65% increase in execution time). To conclude, we can say that there is a tradeoff between quality of solution and execution time.

VI. Summary and Conclusions

In this paper, we focus on a specific class of assignment problems where agents work in collaborating teams. We assume that each agent has a set of capabilities and each task has a set of requirements. The objective is to assign the agents to the teams such that the gain is maximized. The value produced by a team is the result of collaboration between team members and is measured according to equation (12). We propose a Genetic Algorithm (GA) for finding a near optimal solution to this class of task assignment problems. The fitness function of the GA is based on team performance and is calculated according to equation (13). We conducted a wide range of experiments to analyze accuracy, stability, robustness and scalability of the solution obtained by the proposed GA. Our analysis can be summarized as follows: (a) Accuracy: Tables II and V show that the results of the GA are very close to the optimal solution. (b) Stability: Table VI shows that the GA results for five replications deviate very little from each other. This is a measure of stability of the obtained solutions. (c) Scalability: Figure 2 shows that the GA can be used to solve large scale APs. (d) Robustness: Table VII indicates that the GA results are not sensible to changes in the input data. Our conclusion is that the proposed GA is scalable, stable and robust and produces near optimal solution to a specific class of task assignment problems.

References

[1] H. W. Kuhn, “The Hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[2] A. Frank, “On Kuhn’s Hungarian method – a tribute from Hungary,” *Naval Research Logistics,*, vol. 52, pp. 2–5, 2005.

[3] D. W. Pentico, “Assignment problems: A golden anniversary survey,” *European Journal of Operational Research*, vol. 176, no. 2, pp. 774–793, January 2007.

[4] M. L. Fisher, R. Jaikumar, and L. N. V. Wassenhove, “A multiplier adjustment method for the generalized assignment problem,” *Management Science*, vol. 32, no. 9, pp. 1095–1103, September 1986.

[5] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Massachusetts: Addison Wesley, 1989.

[6] M. Mitchell, *Introduction to genetic algorithms*. Cambridge, Massachusetts: MIT Press, 1999.

[7] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press, 1992.

[8] P. C. Chu and J. E. Beasley, “A genetic algorithm for the generalised assignment problem,” *Computers and Operations Research*, vol. 24, pp. 17–23, January 1997.

[9] B. M. Baker and M. A. Ayechew, “A genetic algorithm for the vehicle routing problem,” *Computers and Operations Research*, vol. 30, pp. 787–800, April 2003.

[10] J. F. Gonçalves, J. J. M. Mendes, and M. G. C. Resende, “A genetic algorithm for the resource constrained multi-project scheduling problem,” *European Journal of Operational Research*, vol. 189, no. 3, pp. 1171–1190, 2008.

[11] O. Etiler, B. Toklu, M. Atak, and J. Wilson, “A genetic algorithm for flow shop scheduling problems,” *Journal of the Operational Research Society*, vol. 55, no. 8, pp. 830–835, 2004.

[12] F. Kamrani, R. Ayani, F. Moradi, and G. Holm, “Estimating performance of a business process model,” in *Proceedings of the 2009 Winter Simulation Conference*, M. D. Rossetti, B. Hill, R. R.and Johansson, A. Dunkin, and R. G. Ingalls, Eds., Austin, TX, December 2009.

[13] F. Kamrani, R. Ayani, and A. Karimson, “Optimizing a business process model by using simulation,” in *Proceedings of the 2010 Workshop on Principles of Advanced and Distributed Simulation*, Atlanta, GA, May 2010, pp. 40–47.