EGAGP: An Enhanced Genetic Algorithm for Producing Efficient Graph Partitions

Fahim Shahriar, Aakib Bin Nesar, Naweed Mohammad Mahbub and Swakkhar Shatabda
Department of Computer Science and Engineering, United International University
House 80, Road 8A, Satmasjid Road, Dhanmondi, Dhaka-1209, Bangladesh
Email: swakkhar@cse.uiu.ac.bd

***Abstract*—Graph partitioning is a well-known problem which has varied applications such as scientific computing, distributed computing, social network analysis, task scheduling in multi-processor systems, data mining, cloud computing and many other domains. In this paper, we propose EGAGP, an enhanced genetic algorithm for producing efficient graph partitions. Keeping real world applications in mind, our algorithm takes into account the capacity limitations of individual partitions to ensure balanced partitioning. This approach enables a large variety of applications for this algorithm, the most significant of which is in mobile cloud computing. Our algorithm can be used in efficient deployment of software components in cloud architecture as it is efficient and fast and it also ensures feasibility by only allowing partition sizes up to designated limits. We have achieved significant improvement over the previous state-of-the-art methods in terms of runtime and objective of graph partitioning cost. Our method is based on dividing the total execution time among primary and secondary populations and it resulted in an efficient algorithm. Several standard benchmark graph instances were used in our work to compare the performance of the algorithm. Our proposed method EGAGP is able to produce feasible and good quality results and outperforms the state-of-the-art methods in terms of time and quality of the solutions generated.**

I. Introduction

Graph partitioning is a well-studied problem of computer science. It is a NP-hard problem, so it is generally solved using heuristics and approximation algorithms. In graph partition, vertices of a graph are divided into sets of specific sizes. The cost of the links joining the sets should minimize a cut metric. Graph partitioning has widespread usage and appliance in many practical and theoretical applications. The applications include scientific computing, task scheduling in multi-processor systems, pattern recognition, social network analysis (community discovery), data mining (clustering), partitioning various stages of a VLSI design circuit and more.

In recent times, mobile applications are getting more and more complex. To reduce the burden on users mobile hand set, offloading parts of a software to cloud server is gaining popularity. This enables costly processing tasks to be done on the cloud, so that the application on the users handset remains lightweight and fast. But as the software size grows it becomes more and more difficult to allocate different software components to machines in the cloud while minimizing network bandwidth requirements and maximizing responsivity between the components. In cloudlet [1], [2], a special case of such deployment optimization occurs. It poses a global optimization problem that considers all the software components around the world. Moreover, a fast algorithm is essential for this optimal software deployment in the cloud.

This problem can be modelled as a graph partitioning problem. In the cloudlet setting, software component are mapped to vertices of an weighted graph and have to be partitioned into a number of software components set, so that the machines in cloud can house every set of components. The machines are restricted by their capacity or resource and communication between software components requires latency due to assignments in different machine. Due to the well known complexity of the problem and the fast response needed at the distributed setting, complete or systematic search algorithms are of no use. Local search algorithms with effective heuristics are often used to solve such problems to find good quality of solutions quickly. Many heuristics of different characteristics (combinatorial [3], evolutionist [4], [5], spectral [6], etc.) have been applied to solve the graph partitioning problem to find approximate solutions. One of the most successful existing approaches of graph partitioning such as Kernighan-Lin algorithm [7], is not much scalable and fails to solve the problems with large scale graph data. Besides, many of the existing methods partition the graph into a predefined number of parts of equal sizes. In the context of mobile cloud computing, the number of machines on which the components of the software will be deployed (i.e. the number of partitions) is not predefined. Moreover, the diverse capacity of the machines in the cloud must be taken into account while deploying the software components. The authors of [8] introduced a simulated annealing based graph partitioning algorithm based for software deployment in the cloud. However, the machines that they considered are homogeneous machines. Moreover, their proposed method tends to give infeasible solution in several cases. Lisul et. al. [9] introduced an efficient genetic algorithm FGPFA addressing the problem of producing feasible graph partitions in this context. In order to resemble real scenarios of mobile cloud computing, they propose a model that took into account the heterogeneity of machines along with different capacities of the machines in the cloud. Feasible partitions of the components of the software are ensured by discarding oversized partitions generated during the search.

In this paper, our focus was to improve the FGPGA algorithm proposed in [9]. We applied significant changes in the genetic algorithm described in [9] and also added new features. We proposed a faster method to calculate graph cut size which hugely reduced the calculation time. Our approach of dividing the total calculation time among primary and secondary populations proved to produce better quality results. These procedures are further elaborated in Section IV. We name our approach as EGAGP. The key contributions of our research are as follows:

- We introduce an enhanced genetic approach to obtain feasible graph partitions. We did this by improving FGPGA algorithn [9]. Our proposed algorithm EGAGP produces better quality solutions in an efficient way. We achieved reduction in the runtime by applying faster method to calculate graph cut size. Our approach of improving secondary generation for half of the calculation time and primary population for other half of the calculation time succeeded in producing better quality results.
- In our model, we include consider heterogeneity and capacity or resource constraint of the distributed machines to resemble real scenarios. We also guarantee feasible partition by discarding over-sized partitions during the search.
- Finally, we have performed extensive experiments on standard benchmark datasets to show the efficiency and effectiveness of our proposed modified approach.

II. Preliminaries

In this section, we formally define the graph partitioning problem addressed in this paper. We will also give a brief description of genetic algorithm which we used in our approach.

A. Problem Model

In this paper, we have considered a model similar to that proposed in [9]. Similar to that paper, we also considered heterogeneous machines with varied communication cost between them. In a distributed system setting, we are given a machine graph denoted by G_M. The machine graph G_M represents the backbone infrastructure of the distributed system. Here, $G_M = (M, L)$ is tuple of M which is a set of heterogeneous machines in the cloud and denoted as the set of vertices for this graph and L represents the arcs or edges denoting the connections between each pair of machines. A vertex $m_i \in M$ has an edge with another vertex $m_j \in M$ if and only if there is a communication link between the machines that they represent. Machine $m_i \in M$ has a maximum capacity C_i in terms of resources. Communication link cost are given in matrix B. For a link l_{ij} between machine m_i and m_j corresponds to a communication cost, $b_{ij} \in B$.

We are also given an application graph $G_A = (V, E)$. The application graph represents the software units of the system. An edge connecting two vertices of the graph corresponds to the communication cost between the two software components represented by the vertices. Both vertices and edges of this graph are weighted. The weight of a vertex represents the amount of resource needed for the software component it represents. The weight of vertex $v_i \in V$ is r_i. The matrix W for G_A contains the weights of the edges. For each $w_{ij} \in W$ represents the communication cost between software component v_i and v_j.

Now, the graph partitioning problem reduces to find a mapping $\phi : V \rightarrow M$ of the vertices in V of G_A to the vertices in M of G_M. The mapping ϕ assigns each of the software component $v \in V$ to a machine $m \in M$. The objective of this mapping or partition is to minimize the graph cut size (GCS) defined as following:

$$GCS = \sum_{ij} w_{ij} \times b_{\phi(v_i)\phi(v_j)} \times h_{ij} \tag{1}$$

Here, h_{ij} is defined as follows:

$$h_{ij} = \begin{cases} 0, & \text{if } \phi(v_i) = \phi(v_j) \\ 1, & \text{if } \phi(v_i) \neq \phi(v_j) \end{cases} \tag{2}$$

Here, h_{ij} determines if v_i and v_j are in the same machine or not. If not, $w_{ij} \times b_{(\phi(v_i)\phi(v_j))}$ will be added to GCS.

Since the machines in the infrastructure have maximum capacity limits, another constraint is needed that defines the feasibility of the partitions. This constraint is defined in the following equation:

$$\sum_i r_i \times \Phi(i,k) \leq C_k, \forall k \tag{3}$$

Here $\Phi(i,k)$ is true only if software unit, i is assigned to machine m_k. Formally,

$$\Phi(i,k) = \begin{cases} 0, & \text{if } \phi(v_i) \neq m_k \\ 1, & \text{if } \phi(v_i) = m_k \end{cases} \tag{4}$$

B. Genetic Algorithms

Genetic algorithms are population based nature inspired algorithm has been widely used to solve complex optimization problems [10], [4]. Genetic algorithms maintains a set of solutions called *population* in each iterations or *generations*. The multi-point or multi-candidates in the search ensures the diversification of the algorithm. Each *individual* in the population of the genetic algorithm represents solutions to the related optimization algorithm encoded as chromosomes. Each component of the solution can be thought of individual genes in the chromosome. The initial population evolves towards generating better quality solutions by optimizing a *fitness function*. In each generation of the algorithm, the current population goes over genetic operations like *crossover* and *mutation* and thus yields new individuals. The children produced by crossover adds a degree of diversification generally absent in regular population based algorithms like iterated beam search. In each generation only a set of fit solutions are kept for the next generation as running population based of a selection criteria. This selection criteria often brings intensification necessary to achieve good quality solutions. A schematic diagram for genetic algorithm is given in Fig. 1.

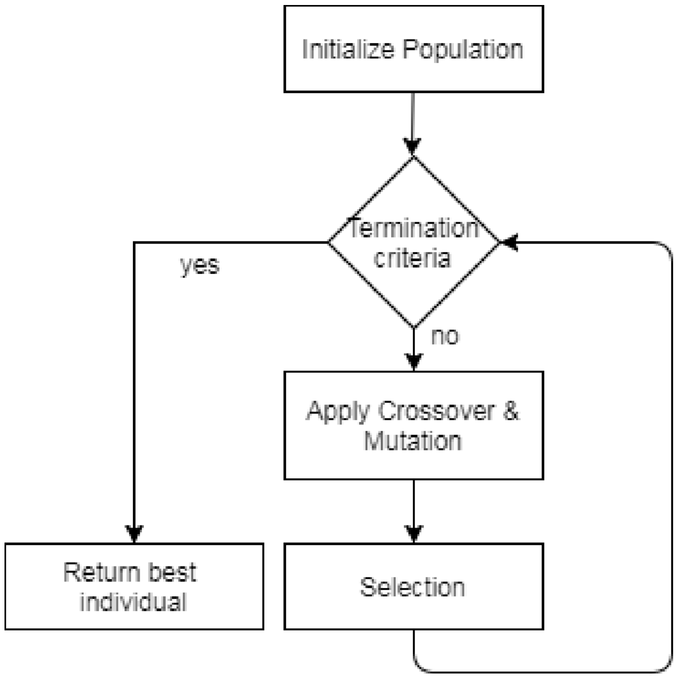


Fig. 1. A schematic diagram of genetic algorithm.

III. Related Work

There are multiple applications of graph partitioning in divergent areas of research, e.g. parallel computing, circuit layout and various serial algorithm design. The process of subdividing the vertices of a graph into certain number of disjoint sets which are approximately similar in size in order to minimize cut metric value is referred by the term graph partitioning. Graph Partitioning is a NP-hard [11] problem. In addition, for general graphs there is no approximation algorithm with constant ratio factor [11]. Many heuristic algorithms for graph partitioning have been developed due to this theoretical limitation, in the past few decades. These algorithms generated high quality partitions within minimum time. A heuristic problem was developed which was inspired by the idea of partitioning electronic circuits onto boards [7]. This algorithm is known as Kernighan-Lin (KL) algorithm. Kernighan-Lin algorithm commonly called KL algorithm was eventually improved to acquire a better result in terms of running time by Fiduccia and Mattheyses [3]. Another well-known method which is known by the name spectral bisection method is based on the concept of spectrum of the graphs Laplacian Matrix. [5], [12]. But this method may get stuck in local optima. Hence to avoid local optima these methods could be combined with various stochastic methods such as simulated annealing [13], particle swarm optimization [14], or ant colony optimization [15]. Moreover, all these methods do not extend to large scale graph data.

The use of multilevel techniques [16] is one of the recent approach which has ensured the steady acceleration in this prospect. In multilevel technique, a graph is approximated by analyzing the sequence of progressively smaller graphs. Then using a spectral method this graph is partitioned [6]. After that this partition is propagated back through the hierarchy of graphs. In order to improve partition one of the variants of Kernighan-Lin algorithm is applied periodically. Some of the well-known software packages which are based on this concept are: Jostle [17] and Metis [17]. Nowadays, new heuristics are being developed by combining graph partitioning algorithms with various known techniques. A PROBE (Population Reinforced Optimization Based Exploration) heuristic is used by Chardaire [18]. In this method, the authors combined greedy algorithms and genetic algorithms with KL refinement. Also, a greedy graph growing heuristic was introduced by Loureiro and Amaral which deploys a local refinement algorithm [19].

All the above-mentioned graph partition methods have three limitations. Firstly they consider a fixed and predefined size of partitions of the graphs. Due to these criteria, the usages of traditional methods directly for cloud computing is being restricted. Verblen et al. [8] proposed a simulated annealing based algorithm for software deployment in the mobile cloud computing setting. In their work, the authors considered the different capacities of machines along with the variation in the number of the machines on which deployment of components of software would take place. Nevertheless, this approach resulted in infeasible solutions for a couple of cases. The other two limitation of the mode is the underlying assumption of homogenity of the machines and the time required to generate good quality solutions by the simulated annealing algorithm.

Heuristic based local search algorithms like genetic algorithms [20], [21], [22], large neighborhood search [23] and ant colony optimization [24], [25] have been widely used to solve complex optimization problems, specially to produce good quality solutions quickly. Lisul et al. [9] proposed FGPGA algorithm that uses genetic algorithm to solve the problem of graph partitioning. The algorithm produces good quality results due to the twin-removal and greedy mutation applied in that work. However, our investigation showed that it is possible to improve over the FGPGA algorithm in terms of runtime and solution quality. In this paper, we are propose EGAGP, an enhanced genetic algorithm that produces feasible graph partitions and improves significantly over the FGPGA algorithm.

Among other related problem to the cloudlet deployment problem or graph partitioning problem are the Google machine reassignment problem [26], community detection [27], protein interaction network partitioning [28], etc.

IV. Our Method

In this section, we describe our proposed method EGAGP which is an enhanced version of the FGPGA algorithm proposed in [9]. This section describes the components of FGPGA algorithm for the sake of completeness and also the features introduced in EGAGP.

A. Encoding

In this paper, we encoded a solution of the graph partitioning problem in a similar way to the encodings of [9]. In short, our algorithm encoded every individual by $|V|$ number of genes. Each gene of an individual is an integer number that can contain value in the range $[1, M]$. Here, M represents the total number of heterogenous machines in the cloud architecture. So, an individual X has an ordered list of $|V|$ genes and each of the genes represent the mapping of a software component to a machine. The value of the gene X_i denotes the machine the software component i is in. e.g.

$$X = X_1, X_2, \cdots, X_{|V|}$$

Here, component i is assigned to machine X_i.

B. Initialization

Our proposed algorithm EGAGP uses a random initialization procedure similar to that of FGPGA [9]. Pseudo-code of the initialization procedure is given in Algorithm 1.

```
Algorithm 1: Initialize(Population $P$)
1  foreach individual $X \in P$ do
2      $m \leftarrow |M|$
3      foreach gene $X_i \in X$ do
4          repeat
5              $m_i \leftarrow RANDU[1, m]$
6          until $free(m_i) \geq r_i$
7          $X_i \leftarrow m_i$
8      end
9  end
10 return $P$
```

Here r_i is the weight of component i. The function $free(m_i)$ returns the available resource in a machine m_i. This initialization procedure guarantees to generate feasible solutions.

C. Fitness Evaluation

Our algorithm will be always bound within the feasible search region as we have ensured feasibility in the initialization, crossover and mutation methods. We used Eqn. 1 to calculate the fitness of an individual. Algorithm 2 shows the pseudo-code of the algorithm for calculating fitness of an individual.

Here, for each pair of genes of X, we are adding $(w_{ij} \times b_{\phi(v_i)\phi(v_j)})$ to the fitness of X, if the values of the genes of the pair are not same. w_{ij} and $b_{\phi(v_i)\phi(v_j)}$ are defined in problem model.

D. Partial Fitness

Partial fitness of a gene of an individual X, is the amount the gene contributes in the fitness value of X. Partial Fitness of a gene i is defined as the following:

$$PartialFitness(i) = \sum_{j \neq i} w_{ij} \times b_{\phi(v_i)\phi(v_j)} \times h_{ij} \quad (5)$$

```
Algorithm 2: Fitness(individual $X$)
1  $X.fitness \leftarrow 0$
2  for $i = 1$ to number of vertices do
3      for $j = i + 1$ to number of vertices do
4          if $X_i \neq X_j$ then
5              $X.fitness \leftarrow$
                   $X.fitness + (w_{ij} \times b_{\phi(v_i)\phi(v_j)})$
6          end
7      end
8  end
```

The pseudo-code for calculating partial fitness of an individual gene is delineated in Algorithm 3.

```
Algorithm 3: PartialFitness(individual $X$,Gene $gene$,
Value $value$)
1  $partialFitness \leftarrow 0$
2  for $i = 1$ to number of vertices do
3      if $i \neq gene$ and $X_i \neq value$ then
4          $partialFitness =$
               $partialFitness + w_{ij} \times b_{(value)(v_j)}$
5      end
6  end
7  return $particlFitness$
```

Here, we are calculating the partial fitness of the gene $gene$ in individual X. We are passing the value of X_{gene} as the parameter $value$. This enables us to calculate partial fitness of any gene containing desired value.

E. Faster Fitness Calculation

Calculating the fitness of an individual is the most repeating task in genetic algorithm. Every time we change the value of a gene, we have to calculate the fitness of the individual to measure the effect of the change. In the FGPGA algorithm [9] the fitness was calculated using Algorithm 2. We found that we can calculate fitness a lot faster.

We observed that, the fitness of an individual X only changes when we change the value of a gene of the individual. If we know the fitness of X prior changing the gene value, the fitness of X after changing the gene value can be measured in linear time. The logic behind it is, the new fitness of X does not contain the partial fitness of the gene for its previous value. So, we subtract that from the fitness of X. And the partial fitness of the gene for its new value is added to the fitness. So, we add that value to fitness of X. The new fitness of individual X is defined in the following equation:

$$\begin{aligned} X.fitness = X.fitness \\ - PartialFitness(X, gene, prev\_value) \\ + PartialFitness(X, gene, new\_value) \end{aligned} \quad (6)$$

If the fitness of X is not calculated, we can calculate it using Algorithm 2. Otherwise, when we change the value of a gene of X, we can update its fitness by using Eqn. 6. The algorithm for changing the value of gene i of X and updating the fitness is giving in Algorithm 4.

```
Algorithm 4: SetGene(individual $X$,Gene $gene$, Value $value$)
1  if $X_{gene} = value$ then
2      return
3  end
4  $valueAdd \leftarrow 0, valueSub \leftarrow 0$
5  $pValue = gene$
6  if $X.isFitnessCalculated = FALSE$ then
7      Fitness($X$)
8      $X.isFitnessCalculated = TRUE$
9  end
10 else
11     $valueAdd = PartialFitness(X, gene, value)$
12     $valueSub = PartialFitness(X, gene, pValue)$
13     $X.fitness = X.fitness - valueSub + valueAdd$
14 end
```

Here we are changing the value of X_{gene} from $pValue$ to $value$ and updating $X.fitness$ at the same time. The algorithm follows the description given above.

F. Greedy Mutation

Greedy mutation procedure described in [9] is used in our algorithm. In greedy mutation, a randomly selected individual is mutated by altering one of its gene. r different values are applied on that gene, and the value that makes the individual fittest, is set as the value of that gene.

G. One Point Crossover with Best Gene Passing

In Greedy Mutation, a lot of work was needed to update a gene. We call the gene that gets updated in greedy mutation the *Best Gene* for that particular individual. We want to pass the best gene in each individual to its offspring.

The One Point Crossover function creates a new individual from two parent individuals. First, a random point K in the range $[1, V]$ is selected Then, the first K genes of the new individual is set from individual 1 and the rest is set from individual 2. After the crossover, we pass the value of the best gene of both parent individuals to their offspring. Pseudo-code for one-point cross over is given in Algorithm 5.

```
Algorithm 5: OnePointCrossover (individual $A$, individual $B$)
1  $C \leftarrow$ new individual
2  randomly choose crossover point $k$
3  set $C[1 \cdots k]$ to the values of $A[1 \cdots k]$
4  set $C[k+1 \cdots V]$ to the values of $A[k+1 \cdots V]$
5  $C[A.bestGene] \leftarrow A[A.bestGene]$
6  $C[B.bestGene] \leftarrow B[B.bestGene]$
7  return $C$
```

Here, a new individual C is created from the crossover of two parent individual A and B. Then a random point K is selected in the range $[1 \cdots V]$. The gene values of C from 1 to k is set to the gene values of A. Then the gene values of C from $k+1$ to V is set to the gene values of B. Then, the value of the best gene of A is set to the corresponding gene of C. Same way, the value of the best gene of B is set to the corresponding gene of C.

H. Twin Removal and Random Restart

To ensure diversification in the generations of genetic algorithm Lisul et al. [9] used twin removal and random restart procedures FGPGA. These are effective methods in escaping stagnation in local search [29], [30].

I. Genetic Algorithm

The genetic algorithm we used in our approach is similar to the genetic algorithm used in [9]. The algorithm takes a population and produces given amount of generations. Each generation produces new offspring by selecting individuals using tournament selection and recombine them using crossover. The algorithm passes the best genes of the parent individuals to the new offspring. Each new offspring then goes through a mutation process which improves their quality. The gene of the individual that the mutation process updates is the best gene for that individual. To ensure elitism our algorithm keeps the individual with the best fitness in the next generation. To preserve diversification and to escape from stagnation, twin removal and random restart procedures are executed periodically. The pseudo-code for the genetic algorithm is given in Algorithm 6.

Here, the algorithm produces generations up to GEN_LIMIT. RST_INTERVAL is the random restart interval.

J. EGAGP Algorithm

While inspecting the FGPGA algorithm [9], we found that in some cases the algorithm terminates even if the last generation produces an improved result than its predecessor generation. This gave us the impression that if the algorithm was not terminated by the generation limit, it could have produced further improvements in the future generations. We confirmed it by increasing the generation limit. We used this knowledge in EGAGP algorithm.

Now the question we faced was, what should be the generation limit. Firstly, we removed the generation limit and let the algorithm run on a population for as long as it produced improvement. This approach improved the result significantly. However, this required massive amount of time. Our second attempt was to set the generation limit to a large value and apply genetic algorithm on a population. This produced better result in acceptable time. However, often the algorithm got stuck in local minima. To recover from this, we selected a high generation limit, GL and a primary population, PP. Then we produced multiple secondary populations and ran the genetic algorithm on each of the

Algorithm 6: GeneticAlgorithm (population P,GEN_LIMIT,$RST\_INTERVAL$)

```
1  $genCount \leftarrow 0$
2  $nonImprovingSteps \leftarrow 0$
3  $nonDiverseSteps \leftarrow 0$
4  while $genCount < GEN\_LIMIT$ do
5      $P\_New \leftarrow$empty population
6      $P\_New.elite \leftarrow$ individual $E \in P$ with best fitness
7      for each individual $X \in P$ do
8          repeat
9              $(X1, X2) \leftarrow$ TournamentSelection(P)
10             $X\_New \leftarrow$ OnePointCrossover($X1, X2$)
11         until until $X\_New$ is valid
12         add $X\_New$ to $P\_New$
13     end
14     for each individual $X \in P\_New$ do
15         GreedyMutate($X$)
16     end
17     find the individual $X\_Best \in P\_New$ with best fitness
18     if Fitness(globalBest) $< X\_Best.fitness$ then
19         $globalBest \leftarrow X\_Best$
20         $nonImprovingSteps \leftarrow 0$
21     end
22     else
23         $nonImprovingSteps++$
24     end
25     if $nonDiverseSteps \geq twinRemovalInterval$ then
26         $TwinRemoval(\P_New)$
27         $nonDiverseSteps \leftarrow 0$
28     end
29     else
30         $nonDiverseSteps++$
31     end
32     if $nonImprovingSteps \geq RST\_INTERVAL$ then
33         RandomRestart($P\_New$)
34         $nonImporvingSteps \leftarrow 0$
35     end
36     $P \leftarrow P\_New$
37     $genCount++$
38 end
39 return $P$
```

secondary populations for 3000 generations, similar to [9]. The number of secondary populations was set as such, so that their collective produced generations would be approximately equal to GL/2. For example, if GL was equal to 90000, then 15 populations were generated, as collective produced generations of 15 populations is 15 * 3000 = 45000, which is equal to GL/2. After applying genetic algorithm on every secondary population, we populated PP with the best individual of every secondary population. Remaining individuals of PP was filled with randomly generated individuals. Then we applied genetic algorithm on PP for GL/2 generations. The individual with best fitness in PP is the solution.

This approach has following benefits:

- GL controls the running time of the algorithm. If time is not an issue, the value of GL can be set to a very high.
- Multiple secondary generations make sure that the result does not get stuck in a local minima.
- Genetic algorithm runs on the primary population for a greater amount of generations. This addresses the issue we found in [9].

The pseudo-code of the EGAGP algorithm is given in Algorithm 7.

Algorithm 7: EGAGP (GL)

```
1  $primaryPopulation \leftarrow$ new population
2  $secondaryPopulationLimit \leftarrow \lceil (GL/2)/3000 \rceil$
3  $primaryRandomRestartInterval \leftarrow ((GL/2)*0.05)$
4  $ite \leftarrow 1$
5  while $ite \geq secondaryPopulationLimit$ do
6      $secondaryPopulation \leftarrow$ new population
7      $secondaryPopulation \leftarrow$ Genetic Algorithm ( $secondaryPopulation$, $secondaryGenerationLimit$, $secondaryRandomRestartInterval$)
8      save the best individual of $secondaryPopulation$ in $primaryPopulation$
9      $ite++$
10 end
11 fill rest of the individuals of $primaryPopulation$ with new random initialized individuals
12 $primaryPopulation \leftarrow$ Genetic Algorithm ($primaryPopulation$, $GL/2$,$primaryRandomRestartInterval$)
13 return best individual of $primaryPopulation$
```

Here, $secondaryPopulationLimit$ is calculated as described above. $primaryRandomRestartInterval$ is set to 5% of the generation limit for primary population.

The EGAGP algorithm first finds out the number of secondary population to be created using GL. Then the secondary populations are created and enhanced using genetic algorithm. Best individual of each secondary population is saved in primary population. After that for every remaining uninitialized individual in primary population, a randomly initialized individual is assigned. The primary population is then enhanced using genetic algorithm. Significantly higher number of generations is produced in primary population compared to secondary population. The best individual from the enhanced primary population is then returned as the result.

V. Experimental Results

This section provides the details of the experimental results. All algorithms used in the experiments were implemented in Java Programming language using JDK 8. We performed our experiments on a Intel(R) Core(TM) i5 4570 processor machine equipped with 8GB RAM and running Windows 10 operating system.

A. Dataset Generation

The dataset we worked on is generated in the same way as described in [9]. Sparse graphs were generated using the Eppstein power law generator. Sparse graphs are more realistic in the cloudlet setting since in a distributed system direct communication links are present only in the geographically neighboring machines. The details of the dataset generation can be found in [9], [8].

B. Experimental Parameters

The necessary parameters for our algorithm are described in Table I.

TABLE I
PARAMETERS USED IN THIS PAPER.

Parameter	Value
Population Size for Secondary Population	20
Population Size for Primary Population	30
Generations Limit for Secondary Population	3000
Similarity Threshold	0.95
Secondary Random Restart Interval	100
Twin Removal	50
Tournament Size	5

C. Comparison with Existing Methods

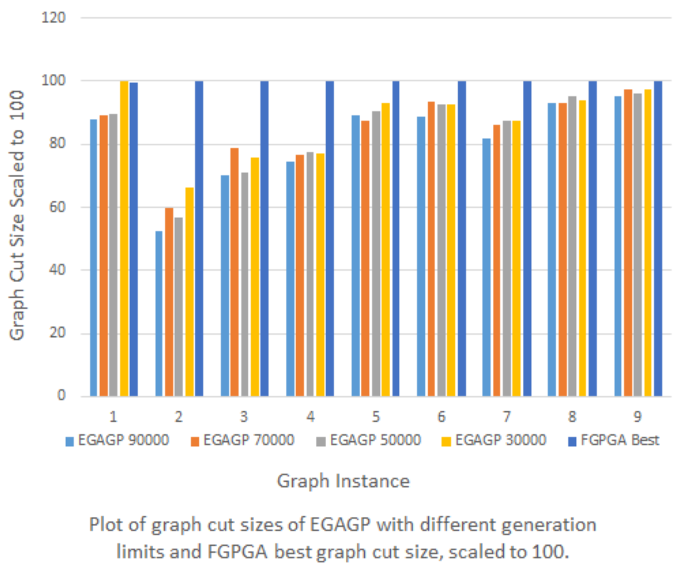
We compared our method with FGPGA method proposed in [9] as well as with Simulated Annealing (SA) algorithm proposed in [8]. We modified the SA algorithm as described in [9]. Rest of the parameters were kept similar as described in the original paper.

D. Results

We report the graph cut size found by EGAGP algorithm with different generation limits, best and average graph cut size found by SA and FGPGA in Table II. Both of the algorithms SA and FGPGA were run 10 times for each of the graph instances. Table II reports best (minimum graph partition cose) and average results from these 10 runs. FGPGA were run for 6000 generations for smaller instances ($\geq$ 500 nodes) and for 3000 generations for larger instances ($>$ 500 nodes) as described in [9]. For EGAGP, the generation limits were set to 90000, 70000, 50000 and 30000. We call EGAGP with generation limit set to 90000 as 'EGAGP 90000'. Each row of the table corresponds to the results for a particular graph instance with different sizes in the of number of vertices or nodes. The solutions with best graph partition costs for each instance is shown in bold faced font. From the results reported in Table II, we can clearly note that EGAGP is able to produce better quality partitions in all the cases compared to SA and FGPGA.

To show the significance of improvement of EGAPG algorithm over FGPGA algorithm, we plot the results for each of the graph instances in Fig. 2. SA was excluded because of low quality partitions. To observe the differences better, the highest value of each row was converted to 100 and the other values were scaled accordingly.

Fig. 2. Plot of graph cut sizes of EGAGP with different generation limits and FGPGA best graph cut size, scaled to 100.



E. Time Comparison

The main reason behind EGAGP algorithms better results is it takes much lower time to produce results. So, more generations can be created compared to [9] and in much less time. Time comparison between EGPGA 90000, EGPGA 70000, EGPGA 50000, EGPGA 30000, SA and FGPGA is shown in Fig. 3. From Fig. 3 we can see that EGAGP algorithms require much lower time compared to SA and FGPGA for all graph instances.

F. Effects of Best Gene

To show the effect of best gene passing in genetic algorithm, we prepare another version of EGAGP 90000 algorithm by turning the best gene passing method off. Fig. 4 shows the effect of best gene passing. The highest value was converted to 100 and the other value was scaled accordingly.

From Fig. 4 we can see that best gene passing improves the graph cut size by approximately 3% on average.

G. Discussion

If we observe Table II, Fig. 2 and Fig. 3, we can see that to produce better results or better graph cut size, more time is needed. But the priority between better graph cut size and time varies in different applications. Some application demands fastest possible result while sacrificing quality. Other application demands quality over time. With EGAGP we tried to meet both of the demands. EGAGP 30000 is the fastest and it produces acceptable results while EGAGP 90000 is almost 3 times slower than EGAGP 30000, but it produces the

TABLE II
Graph Cut size produced by EGAGP with different generation limits, best and average cut size produced by simulated annealing [8] and FGPGA [9] for different size graph instances.

Graph	vertices	EGAGP				SA [8]		FGPGA [9]	
Instance	$\|V\|$	9000	7000	5000	3000	Best	Average	Best	Average
1	100	**585292.87**	595210.14	598868.34	667706.67	1.31E+07	1.62E+07	665323.58	3114607.81
2	200	**3.59E+07**	4.07E+07	3.88E+07	4.52E+07	1.93E+08	2.13E+08	6.83E+07	7.84E+07
3	300	**1.96E+07**	2.20E+07	1.98E+07	2.12E+07	1.13E+08	1.22E+08	2.79E+07	3.17E+07
4	400	**2.46E+08**	2.54E+08	2.56E+08	2.55E+08	6.57E+08	6.78E+08	3.31E+08	3.69E+08
5	500	2.99E+08	**2.93E+08**	3.03E+08	3.11E+08	5.48E+08	5.55E+08	3.35E+08	3.50E+08
6	600	**1.79E+09**	1.89E+09	1.87E+09	1.87E+09	2.73E+09	2.80E+09	2.02E+09	2.05E+09
7	700	**2.04E+09**	2.14E+09	2.18E+09	2.18E+09	3.90E+09	3.92E+09	2.49E+09	2.57E+09
8	800	3.72E+09	**3.71E+09**	3.80E+09	3.75E+09	6.05E+09	6.09E+09	3.99E+09	4.07E+09
9	900	**3.06E+09**	3.14E+09	3.10E+09	3.14E+09	4.67E+09	4.68E+09	3.22E+09	3.39E+09

Fig. 3. Plot of running time of EGAGP with different generation limits, FGPGA and Simulated Annealing.

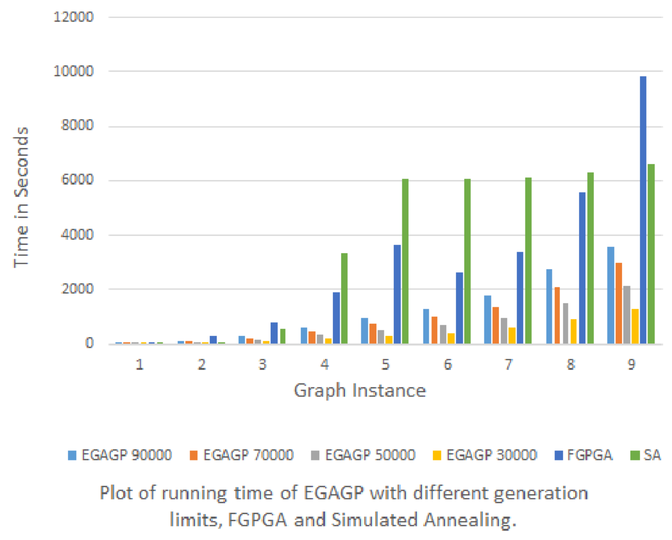
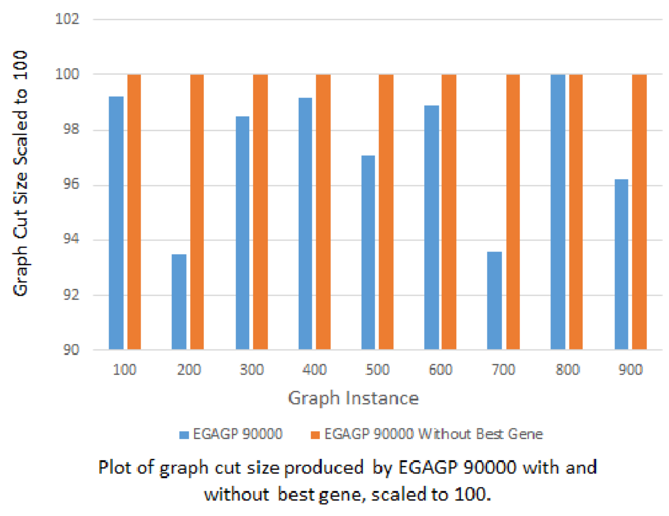


Fig. 4. Plot of graph cut size produced by EGAGP 90000 with and without best gene, scaled to 100.



best results. To get the best result out of EGAGP algorithm concerning time and quality, it is necessary to figure out the correct value of generation limit. We performed Wilcoxon signed-rank test to ensure the performance improvement of our algorithm over other methods.

VI. Conclusion

The current outburst of mobile cloud computing has led us to a new set of challenges such as optimal implementation of software applications on the venturesome infrastructure in the cloud. These problems can be modeled as graph partitioning problems, where a weighted graph of software components need to be partitioned into different available machines in the cloud. Therefore, we have introduced some modifications in this paper such as faster graph cut cost, best gene passing using one-point crossover to the approach of FGPFA [9], which was initially used to tackle the problem mentioned above by the use of genetic algorithm. Just like [9], our modifications also take into consideration the diversity and capacity constraints of the partitions that imitate real scenario of mobile cloud computing. Along with the usefulness of the partitions, superiority of experimental results has been exhibited in this paper with not only standard benchmark dataset but also with [9] and [8]. Our modification is faster, more feasible and light weight for which it is applicable to use in cloud architecture. As a future work, we shall apply EGAGP to real architecture in order to investigate the performance of our proposed modified method for real world problem.

Acknowledgment

The authors would like to thank Md. Lisul Islam for providing his implementations of FGPGA algorithm [9].

REFERENCES

[1] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.

[2] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 29–36.

[3] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Papers on Twenty-five years of electronic design automation*. ACM, 1988, pp. 241–247.

[4] T. N. Bui and B. R. Moon, "Genetic algorithm and graph partitioning," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 841–855, 1996.

[5] J. G. Martin, "Spectral techniques for graph bisection in genetic algorithms," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, pp. 1249–1256.

[6] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM journal on matrix analysis and applications*, vol. 11, no. 3, pp. 430–452, 1990.

[7] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[8] T. Verbelen, T. Stevens, F. De Turck, and B. Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 451–459, 2013.

[9] M. L. Islam, N. Nurain, S. Shatabda, and M. S. Rahman, "Fgpga: An efficient genetic approach for producing feasible graph partitions," in *Networking Systems and Security (NSysS), 2015 International Conference on*. IEEE, 2015, pp. 1–8.

[10] J. H. Holland, "Genetic algorithms and the optimal allocation of trials," *SIAM Journal on Computing*, vol. 2, no. 2, pp. 88–105, 1973.

[11] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is np-hard," *Information Processing Letters*, vol. 42, no. 3, pp. 153–159, 1992.

[12] S. T. Barnard, "Pmrsb: Parallel multilevel recursive spectral bisection," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. ACM, 1995, p. 27.

[13] L. Sun and M. Leng, "An effective multi-level algorithm based on simulated annealing for bisecting graph," in *Energy Minimization Methods in Computer Vision and Pattern Recognition*. Springer, 2007, pp. 1–12.

[14] L. Sun, M. Leng, and S. Yu, "A new multi-level algorithm based on particle swarm optimization for bisecting graph," *Lecture notes in computer science*, vol. 4632, pp. 69–80, 2007.

[15] M. Leng and S. Yu, "An effective multi-level algorithm based on ant colony optimization for bisecting graph," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2007, pp. 138–149.

[16] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Review*, vol. 41, no. 2, pp. 278–300, 1999.

[17] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

[18] P. Chardaire, M. Barake, and G. P. McKeown, "A probe-based heuristic for graph partitioning," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1707–1720, 2007.

[19] R. Zamprogno and A. R. Amaral, "An efficient approach for large scale graph partitioning," *Journal of combinatorial optimization*, vol. 13, no. 4, pp. 289–320, 2007.

[20] S. Khatun, H. U. Alam, and S. Shatabda, "An efficient genetic algorithm for discovering diverse-frequent patterns," in *Electrical Engineering and Information Communication Technology (ICEEICT), 2015 International Conference on*. IEEE, 2015, pp. 1–7.

[21] S. Khatun, H. U. Alam, M. A. Rasid, and S. Shatabda, "Gene transfer: A novel genetic operator for discovering diverse-frequent patterns," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2015, pp. 309–316.

[22] S. Shatabda, M. H. Newton, M. A. Rashid, and A. Sattar, "An efficient encoding for simplified protein structure prediction using genetic algorithms," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*. IEEE, 2013, pp. 1217–1224.

[23] S. Shatabda, M. H. Newton, and A. Sattar, "Constraint-based evolutionary local search for protein structures with secondary motifs," in *Pacific Rim International Conference on Artificial Intelligence*. Springer, 2014, pp. 333–344.

[24] S. Baral, S. Shatabda, and M. A. Rashid, "Cycloant: sequencing cyclic peptides using hybrid ants," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 4–11.

[25] M. Hasanat, M. Hasan, I. Ahmed, M. I. Chowdhury, J. Ferdous, and S. Shatabda, "An ant colony optimization algorithm for load shedding minimization in smart grids," in *Informatics, Electronics and Vision (ICIEV), 2016 5th International Conference on*. IEEE, 2016, pp. 176–181.

[26] H. Gavranović, M. Buljubašić, and E. Demirović, "Variable neighborhood search for google machine reassignment problem," *Electronic Notes in Discrete Mathematics*, vol. 39, pp. 209–216, 2012.

[27] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

[28] J. Vlasblom and S. J. Wodak, "Markov clustering versus affinity propagation for the partitioning of protein interaction graphs," *BMC bioinformatics*, vol. 10, no. 1, p. 99, 2009.

[29] M. A. Rashid, S. Shatabda, M. Newton, M. T. Hoque, D. N. Pham, and A. Sattar, "Random-walk: a stagnation recovery technique for simplified protein structure prediction," in *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*. ACM, 2012, pp. 620–622.

[30] M. L. Islam, S. Shatabda, and M. S. Rahman, "Gremutrrr: A novel genetic algorithm to solve distance geometry problem for protein structures," in *8th International Conference on Electrical and Computer Engineering*.