Ant Colony Optimization Algorithm

Nada M. A. Al Salami

dr_nada71@yahoo.com

ABSTRACT

Hybrid algorithm is proposed to solve combinatorial optimization problem by using Ant Colony and Genetic programming algorithms. Evolutionary process of Ant Colony Optimization algorithm adapts genetic operations to enhance ant movement towards solution state. The algorithm converges to the optimal final solution, by accumulating the most effective sub-solutions.
Keywords:, ACO, Genetic algorithm, System theory.

1. Background and Related Work

Genetic Algorithms (GA) have been used to evolve computer programs for specific tasks, and to design other computational structures. The recent resurgence of interest in AP with GA has been spurred by the work on Genetic Programming (GP). GP paradigm provides a way to do program induction by searching the space of possible computer programs for an individual computer program that is highly fit in solving or approximately solving the problem at hand[1][2]. The genetic programming paradigm permits the evolution of computer programs which can perform alternative computations conditioned on the outcome of intermediate calculations, which can perform computations on variables of many different types, which can perform iterations and recursions to achieve the desired result, which can define and subsequently use computed values and sub-programs, and whose size, shape, and complexity is not specified in advance. GP use relatively low-level primitives, which are defined separately rather than combined a priori into high-level primitives, since such mechanism generate hierarchical structures that would facilitate the creation of new high-level primitives from built-in low-level primitives [3] [4] [5].

Unfortunately, since every real life problem are dynamic problem, thus their behaviors are much complex, GP suffers from serious weaknesses. . random systems. Chaos is important, in part, because it helps us to cope with unstable system by improving our ability to describe, to understand, perhaps even to forecast them. Ant Colony Optimization (ACO) is the result of research on computational intelligence approaches to combinatorial optimization originally conducted by Dr. Marco Dorigo, in collaboration with Alberto Colorni and Vittorio Maniezzo [6]. The fundamental approach underlying ACO is an iterative process in which a population of simple agents repeatedly construct candidate solutions; this construction process is probabilistically guided by heuristic information on the given problem instance as well as by a shared memory containing experience gathered by the ants in previous iteration. ACO has been applied to a broad range of hard combinatorial problems. Problems are defined in terms of components and states, which are sequences of components. Ant Colony Optimization incrementally generates solutions paths in the space of such components, adding new components to a state. Memory is kept of all the observed transitions between pairs of solution components and a degree of desirability is associated to each transition depending on the quality of the solutions in which it occurred so far. While a new solution is generated, a component y is included in a state, with a probability that is proportional to the desirability of the transition between the last component included in the state, and y itself [7]. The main idea is to use the self-organizing principles to coordinate populations of artificial agents that collaborate to solve computational problems. Self-organization is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions among its lower-level components. The rules specifying the interactions among the system's constituent units are executed on the basis of purely local information, without reference to the global pattern, which is an emergent property of the system rather than a property imposed upon the system by an external ordering influence. For example, the emerging structures in the case of foraging in ants include spatiotemporally organized networks of pheromone trails [8][9][10]. The aim of this work is to enhance the ability of ACO by using GP technique, as we describe in the next secttion.

2. Genetic Programming

Some specific advantages of genetic programming are that no analytical knowledge is needed and still could get accurate results. GP approach does scale

with the problem size. GP does impose restrictions on how the structure of solutions should be formulated. There are Several variants of GP, some of them are: Linear Genetic Programming (LGP), Gene Expression Programming (GEP), Multi Expression Programming (MEP), Cartesian Genetic Programming (CGP), Traceless Genetic Programming (TGP) and Genetic Algorithm for Deriving Software (GADS). In the next section we shall concentrate on CGP, since it is the most near to our proposed method [4][5].Cartesian Genetic Programming was originally developed by Miller and Thomson [11][12] for the purpose of evolving digital circuits and represents a program as a directed graph. One of the benefits of this type of representation is the implicit re-use of nodes in the directed graph. Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. In CGP, the genotype is a fixed-length representation and consists of a list of integers which encode the function and connections of each node in the directed graph. The genotype is then mapped to an indexed graph that can be executed as a program. In CGP there are very large numbers of genotypes that map to identical genotypes due to the presence of a large amount of redundancy. Firstly there is node redundancy that is caused by genes associated with nodes that are not part of the connected graph representing the program. Another form of redundancy in CGP, also present in all other forms of GP is, functional redundancy. Simon Harding, and Ltd introduce computational development using a form of Cartesian Genetic Programming that includes self-modification operations..The interesting characteristic of CGP are :

1. More powerful program encoding using graphs, than using conventional GP tree-like representations, the population of strings are of fixed length, whereas their corresponding graphs are of variable length depending on the number of genes in use.

2, Efficient evaluation derived from the intrinsic feature of subgraph-reuse exhibited by graphs.

3. Less complicated graph recombination via the crossover and mutation genetic operators.
.

3. Proposed ACO Genetic Algorithm (ACOG)

A combinatorial optimization problem is a problem defined over a set $\mathbf{C} = c_1, ..., c_n$ of basic *components*. A subset $\mathbf{S}$ of components represents a *solution* of the problem; $\mathbf{F} \subseteq 2^C$ is the subset of *feasible solutions*, thus a solution $\mathbf{S}$ is feasible if and only if $\mathbf{S} \in \mathbf{F}$. A *cost function z* is defined over the solution domain, $z : 2^C \rightarrow \mathbf{R}$, the objective being to find a minimum cost feasible solution S*, i.e., to find S*: S* $\in \mathbf{F}$ and $z(S^*) \leq z(\mathbf{S})$, $\forall \mathbf{S} \in \mathbf{F}$ [8].. They move by applying a stochastic local decision policy based

on two parameters, called *trails* and *attractiveness*. By moving, each ant incrementally constructs a solution to the problem. The ACO system contains two rules:

1. Local pheromone update rule, which applied whilst constructing solutions.
2. Global pheromone updating rule, which applied after all ants construct a solution.

Furthermore, an ACO algorithm includes two more mechanisms: trail evaporation and, optionally, daemon actions. Trail evaporation decreases all trail values over time, in order to avoid unlimited accumulation of trails over some component. Daemon actions can be used to implement centralized actions which cannot be performed by single ants, such as the invocation of a local optimization procedure, or the update of global information to be used to decide whether to bias the search process from a non-local perspective [6][10]

At each step, each ant computes a set of feasible expansions to its current state, and moves to one of these in probability. The probability distribution is specified as follows. For ant k, the probability of moving from state t to state ṇ depends on the combination of two values [9]:

- the attractiveness of the move, as computed by some heuristic indicating the priori desirability of that move;
- the trail level of the move, indicating how proficient it has been in the past to make that particular move: it represents therefore an a posteriori indication of the desirability of that move.

ACOG Algorithm

An ACOG is differ from that algorithm given in refrence [13], it use genetic programming to enhance performance. It consists of two main sections: *initialization* and a *main loop, where Gp is used in the second sections*. The main loop runs for a user-defined number of iterations. These are described below:

♦*Initialization:*
 a. Set initial parameters that are system: variable, states, function, input, output, input trajectory, output trajectory.
 b. Set initial pheromone trails value.
 c. Each ant is individually placed on initial state with empty memory.

♦*While termination conditions not meet do*
 a. Construct Ant Solution:
Each ant constructs a path by successively applying the transition function the probability of moving from state to state depend on: as the attractiveness of the move, and the trail level of the move.
b. Apply Local Search

c. Best Tour check:

If there is an improvement, update it.
d. Update Trails:
- Evaporate a fixed proportion of the pheromone on each road.
- For each ant perform the "ant-cycle" pheromone update.
- Reinforce the best tour with a set number of "elitist ants" performing the "ant-cycle"
d. Create a new population by applying the following operation, based on **pheromone trails***. The operations are applied to individual(s) selected from the population with a probability based on fitness.*

- *Darwinian Reproduction*
- *Structure-Preserving Crossover*
- *Structure-Preserving Mutation*

End While

4. The Performance of Genetic Process

Genetic generation process involves probabilistic steps, because of these probabilistic steps, non-convergence and premature convergence, i.e. convergence to a globally sub-optimal result, problems become inherent features of genetic generation process. To minimize the effect of these problems, multiple independent runs of a problem must be made. Best-of-run individual from all such multiple independent runs can then be designated as the result of the group of runs. If every run of GPG were successful in yielding a solution, the computational effort required to get the solution would depends primarily on four factors: population size, M, number of generation that are run, g, (g must be less than or equal to the maximum number of generation G) the amount of processing required for fitness measure over all fitness cases, and the amount of processing required for test phase e, we assume that the processing time to measure the fitness of an individual is its run time, P. If success occurs on the same generation of every run, then the computational effort E would be computed as follows:

$$E = M \cdot g \cdot \beta \cdot e \qquad \ldots\ldots\textbf{E q.1}$$

Since the value of e is too small with respect to other factors, we shall not consider it. However, in most cases, success occurs on a different generations in different runs, then the computational effort E would be computed as follows:

$$E = M \cdot g_{avr} \cdot \beta \qquad \ldots\ldots\textbf{Eq.2}$$

where: g_{avr} is the average number of executed generations

Since GPG is a probabilistic algorithm, not all runs are successful at yielding a solution to the problem by generation G. Thus, the computational effort is computed in this way, first determining the number of independent runs R needed to yield a success with a certain probability. Second, multiply R by the amount of processing required for each run, that is. The number of independent runs R required to satisfy the success predicate by generation i with a probability z which depends on both z and P (M, i), where z is the probability of satisfying the success predicate by generation i at least once in R runs defined by:

$$z = 1 - [1 - P (M, i)]R \qquad \ldots\ldots\textbf{Eq.3}$$

P (M, i) is the cumulative probability of success for all the generations between generation 0 and generation i. P (M, i) is computed after experimentally obtaining an estimate for the instantaneous probability Y (M, i) that a particular run with a population size M yields, for the first time, on a specified generation i, an individual is satisfying the success predicate for the problem]. This experimental measurement of Y(M, i) usually requires a substantial number of runs. After taking logarithms for equation 4, we find:

$$R = \left\lceil \frac{\log(1-z)}{\log(1-\rho(M,i))} \right\rceil$$
$$\ldots\textbf{Eq.4}$$

The computational effort E, is the minimal value of the total number of individuals that must be processed to yield a solution for the problem with z probability (ex: z = 99%):

$$E = M \cdot (\dot{g} + 1) \cdot \beta \cdot R \qquad \ldots\ldots\textbf{Eq.5}$$

Where $\dot{g}$ is the first generation at which minimum number of individual evaluation is produced, it is called best generation. $\dot{g}$ value is incremented by one since generation $\dot{g}$ must also run to reach the solution. From equation (5), computational effort depends on the particular choices of values for M, G, P (M, i), and the effort required for fitness evaluation, hence, the value of E is not necessarily the minimum computational effort possible for the problem.

5. Conclution
Since Ant colony algorithm may produce redundant states in the graph, its better to minimize such graphs to enhance the behavior of the inducted system. A colony of ants moves through system states **X**, by applying Genetic Operations. By moving, each ant incrementally constructs a solution to the problem. .

When an ant complete solution, or during the construction phase, the ant evaluates the solution and modifies the trail value on the components used in its solution. Ants deposit a certain amount of pheromone on the components; that is, either on the vertices or on the edges that they traverse. The amount of pheromone deposited may depend on the quality of the solution found. Subsequent ants use the pheromone information as a guide toward promising regions of the search space. Ants adaptively modify the way the problem is represented and perceived by other ants, but they are not adaptive themselves. The genetic programming paradigm permits the evolution of computer programs which can perform alternative computations conditioned on the outcome of intermediate calculations, which can perform computations on variables of many different types, which can perform iterations and recursions to achieve the desired result, which can define and subsequently use computed values and sub-programs, and whose size, shape, and complexity is not specified in advance.

References

[1] A. Abraham et al.: Evolutionary Computation: from Genetic Algorithms to Genetic Programming, Studies in Computational Intelligence (SCI) 13, 1–20 (2006), www.springerlink.com c _ Springer-Verlag Berlin Heidelberg 2006.

[2] C. Grosan and A. Abraham: Hybrid Evolutionary Algorithms: Methodologies, Architectures, and Reviews, Studies in Computational Intelligence (SCI) 75, 1–17 (2007), www.springerlink.com c_ Springer-Verlag Berlin Heidelberg 2007.

[3] Héctor A Montes and Jeremy L Wyatt ,” Cartesian Genetic Programming for Image Processing Tasks”,

[4] John R. Koza, Margaret Jacks Hall, “ SURVEY OF GENETIC ALGORITHMS AND GENETIC PROGRAMMING”, http://www-cs-faculty.stanford.edu/~koza/.

[5] Riccardo Poli, William B. Langdon , Nicholas F. McPhee, John R. Koza, “Genetic Programming :An Introductory Tutorial and a Survey of Techniques and pplications” , Technical Report CES-475 ISSN: 1744-8050 October 2007. essex.ac.uk/dces/research/publications/.../2007/ces475.pdf Appendex

[6] Figures[1] M. Dorigo, M. Birattari, and T. Stitzle, “Ant Colony Optimization: Arificial Ants as a Computational Intelligence Technique, IEEE computational intelligence magazine, November, 2006.

[7] M. Dorigo, G. Di Caro, and L.M. Gambardella, “Ant algorithm for discrete optimization”, Artificial Life, vol. 5, no. 2, pp. 137-172, 1999.

[8] J. Holland, “Adaptation in Natural and Artificial Systems”, Ann Arbor: University of Michigan Press, 1975.

[9] Nada M. A. AL-salami, Saad Ghaleb Yaseen, “Ant Colony Optimization”, IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.6, pp 351-357, June, 2008.

[10] M. Dorigo and G. Di Caro, “The Ant Colony Optimization meta-heuristic”, in New Ideas in Optimization, D. Corne et al., Eds., McGraw Hill, London, UK, pp. 11-32, 1999.

[11] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, Genetic Programming, Proceedings of EuroGP’2000, volume 1802 of LNCS, pages 121–132, Edinburgh, 2000. Springer-Verlag.

[12] Simon Harding, Julian F. Miller, Wolfgang Banzhaf, “Self-Modifying Cartesian Genetic Programming”, GECCO’07, July 7–11, 2007, ACM 978-1-59593-697-4/07/0007, pp: 1021-1028.

[13] Nada M.A. AL-Salami, “System Evolving using Ant Colony Optimization Algorithm “, Journal of Computer Science 5 (5): 380-387, 2009, ISSN 1549-3636.