An Ant Colony Algorithm for the Sequential Testing Problem under Precedence Constraints

Bülent Çatay
Sabanci University
Orhanli, Tuzla
34956, Istanbul, Turkey
Email: catay@sabanciuniv.edu

Özgür Özlük
San Francisco State University
College of Business
San Francisco, CA, USA
Email: ozgur@sfsu.edu

Tonguç Ünlüyurt
Sabanci University
Orhanli, Tuzla
34956, Istanbul, Turkey
Email: tonguc@sabanciuniv.edu

***Abstract*— We consider the problem of minimum cost sequential testing of a series (parallel) system under precedence constraints that can be modeled as a nonlinear integer program. We develop and implement an ant colony algorithm for the problem. We demonstrate the performance of this algorithm for special type of instances for which the optimal solutions can be found in polynomial time. In addition, we compare the performance of the algorithm with a special branch and bound algorithm for general instances. The ant colony algorithm is shown to be particularly effective for larger instances of the problem.**

Keywords–Sequential testing; ant colony optimization; precedence constraints.

I. Introduction and Literature Review

In its most general setting, the sequential testing problem requires the identification of the state of a system consisting of a number of components with the minimum expected cost. The state of the system depends on the state of the individual components. Both the state of the system and the state of the components belong to discrete sets. For instance, the individual components and the system may be either in failing or working state. One has to apply costly tests to learn the states of the individual components. The sequential testing problem arises in different contexts such as diagnosis problems in various areas, artificial intelligence problems etc. Different application areas and results related to the general sequential testing problem can be found in [1]. In this particular study, we consider a series (or parallel) system where the system functions if and only if all (at least one of) the individual components function. Since all the results for the series can be adapted to the parallel case by a dual argument in what follows we will only consider a series system.

The series system is a special case of k-out-of-n systems where the system functions when at least k out of its n components are functioning. So a series system is an n-out-of-n system and a parallel system is a 1-out-of-n system.

For instance, the system that is tested could be a newly manufactured machine that could be in working or failing state. The machine consists of independent components that are themselves in failing or working state. The machine functions if all its independent components function. So there is no redundancy in this system. In order to demonstrate that this machine is failing, one has to know that at least one component does not function. On the other hand, to conclude that the machine is functioning properly one has to know that all the components are working. Finding out whether component i works or not has a cost of c_i and from historical data the probability that component i functions is known and given as p_i. A testing strategy for a series system then inspects the components one by one until a failing component is found or all the components have been tested. So an inspection strategy for this system is simply a permutation of all the components. We stop testing as soon as we find a component that fails in which case we decide that the system is in failing condition or all the components are tested and all are working in which case the system is in working state. If a permutation π is used to test a series system, the expected cost of this strategy is given by

$$c_{\pi(1)} + p_{\pi(1)}c_{\pi(2)} + ... + p_{\pi(1)}p_{\pi(2)}...p_{\pi(n-1)}c_{\pi(n)}. \quad (1)$$

since component $\pi(i)$ is tested only if components $\pi(1), \pi(2), ..., \pi(i-1)$ are tested and all of them are functioning. With this formulation, finding an optimal sequence is easy. One just needs to test all the independent components in increasing order of $\frac{c_i}{1-p_i}$. This is an intuitive ordering since we first test the component that has less cost and high probability of failing since the testing stops when a component that is not functioning is detected. This result has been published in the literature various times in different areas and can be proved easily with an interchange argument. (see e.g. [2])

Let us note the resemblance between our problem and the single machine scheduling problem where the objective is to minimize the weighted completion times. In the scheduling problem, in order to find the completion time of a job, we sum up all the processing times of the tasks that are scheduled before that job. In our objective function, the costs (corresponding to the weights in the scheduling problem) are multiplied by the product of the probabilities of the components tested before that component.

In the case of a physical system, it may not be possible to test the components in any sequence. Certain tests can be applied only after other certain tests are performed. This could be due to the physical location of the components or technological reasons. Essentially, these constraints correspond to precedence constraints. This is again similar to the scheduling literature. One can describe the precedence constraints by an acyclic directed graph where the nodes of the graph correspond to the components and an arc from node i to node j means test j can be applied only after test i has been applied. We will refer to this graph as the precedence graph.

It is known that the single machine scheduling problem to minimize the weighted completion time respecting precedence constraints is an NP-complete problem. (see e.g. [3]) To the best knowledge of the authors, there is no formal proof for the NP-completeness of the testing problem for the series system under precedence constraints. On the other hand, the sequential testing problem seems more difficult due to the structure of the objective function.

In the existence of precedence constraints there are a few analytical results for the sequential testing problem. In [4] an optimal algorithm is provided when the precedence constraints satisfy certain conditions. These conditions require that the precedence graph is a forest and each tree in the forest is either an out-tree or in-tree. In [5] an algorithm is proposed that is optimal for the series system when the precedence graph is a special forest where the outdegree of each node is 0 or 1. So each tree in the forest looks like a directed path. This result is a special case of the result in [4] for the series system. Both in [5] and [4], no computational results are reported. The algorithm proposed in [5] can also be used for the more general k-out-of-n systems and it is shown that the proposed algorithm is optimal for certain k-out-of-n systems. Let us note that for k-out-of-n systems a testing strategy can be described by a binary tree where the nodes correspond to the components and the two outgoing arcs outgoing from a node correspond to the failing and working condition of the components. It is not always possible to represent an optimal policy by a permutation of the components in this case. This is true even when there are no precedence constraints.

II. Problem Definition

We consider a series system consisting of n components. The cost of testing component i is c_i. The tests are perfect in the sense that at a cost of c_i we learn the correct state of component i. The components can be in one of the two states; 1 corresponds to the functioning state and 0 corresponds to the failing state of component i. The probability that component i functions is given as p_i. We assume that the functionality of the components are independent of each other. For such a system a feasible testing sequence is a permutation π of $\{1, 2, ..., n\}$. The total expected cost of a solution π is given by $\sum_{i=1}^{n} c_{\pi(i)}(\prod_{j=1}^{i-1} p_{\pi(j)})$. In our case, not all permutations are feasible due to the precedence constraints among the components. The precedence constraints can naturally be represented by a directed acyclic graph. If arc (i, j) exists in the precedence graph, that means component j can be tested only if component i is already tested. Let P be the set of all feasible sequences satisfying the precedence constraints. Then the problem can be formulated as follows:

$$\min_{\pi \in P} \sum_{i=1}^{n} c_{\pi(i)} \prod_{j=1}^{i-1} p_{\pi(j)} \tag{2}$$

Alternatively, the problem can be formulated as a nonlinear integer programming problem where the objective function is a nonlinear function, the constraints are linear functions and the decision variables are {0,1} variables in the following way where A is the set of arcs in the precedence graph. The decision variables are defined as follows:

$$x_{ij} = \begin{cases} 1, & \text{if component } i \text{ is tested in order } j \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$\min \sum_{k=1}^{n} \sum_{i=1}^{n} ((x_{ik} c_i) \prod_{j=1}^{k-1} \sum_{i=1}^{n} (x_{ij} p_j)) \tag{4}$$

$$\text{subject to} \tag{5}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \ \forall j \in \{1, 2..., n\} \tag{6}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \ \forall i \in \{1, 2, ..., n\} \tag{7}$$

$$x_{ik} \leq x_{jl}, \ \forall (i, j) \in A \text{ and } l \leq k - 1 \tag{8}$$

$$x_{ij} \in \{0, 1\}, \ \forall i, j \in \{1, 2, ..., n\} \tag{9}$$

The objective function (4) represents the total expected cost of a feasible solution. The constraints (6) ensure that every component is assigned an order and the constraints (7) ensure that every order is assigned a component. Constraints (8) ensure that precedence constraints are satisfied.

III. Heuristic Algorithms

In order to solve the problem, an ant colony algorithm is developed. In order to test the effectiveness of this algorithm a branch and bound algorithm with preprocessing is implemented. We also report the results obtained by a greedy algorithm that is based on the optimal algorithm for the case where there are no precedence constraints. In this section, we describe these algorithms.

A. Greedy Algorithm

The greedy algorithm uses the idea of the optimal algorithm when there are no precedence constraints and chooses the next component to test greedily. Recall that when there are no precedence constraints the optimal strategy is to inspect the component with the smallest $\frac{c_i}{1-p_i}$ ratio. When there are

precedence constraints, we greedily choose the component with the smallest $\frac{c_i}{1-p_i}$ ratio among those components whose all predecessors are already tested. We will refer to this algorithm as the greedy algorithm.

Let us denote by $Prec(i)$ the set of nodes k such that $(k, i) \in A$. In other words, $Prec(i)$ is the set of immediate predecessors of node i.

0. Let $Active$ be the set of nodes i for which $Prec(i] = \emptyset$.
1. Next component to test is the component $j \in Active$ with the minimum ratio. $(c_i/(1 - p_i))$
2. Delete node j from $Active$ and update the precedence graph by deleting node j and all arcs outgoing from node j.
3. Goto step 1 as long as there are more components to test.

The greedy algorithm is very easy to implement and provides feasible solutions almost in no time.

B. Ant Colony Algorithm

Ant Colony Optimization (ACO) is metaheuristic approach designed for solving hard combinatorial optimization problems. It is based on the observation of the behavior of real ant colonies searching for food sources. Real ants deposit an aromatic chemical substance, called pheromone, on the path they walk. If other ants searching for food sense the pheromone on a path, they are likely to follow it rather than traveling at random, thus reinforcing the path. As more and more ants follow a path the level of pheromone on that path will enhance, which in turn will increase its selection probability by other ants. On the other hand, the pheromone evaporates over time, reducing the chance of other ants to follow the path. The longer the path between the nest and the food source the more the pheromone evaporates. Thus, the pheromone levels remain higher on the shorter paths. As a consequence, the level of pheromone laid is basically based on the path length and the quality of the food source. ACO simulates the above behavior of real ants to solve combinatorial optimization problems by using artificial ants. To apply ACO, the optimization problem is transformed into the problem of finding the best path on a weighted graph. The artificial ants incrementally build solutions by moving on the graph using a stochastic construction process guided by artificial pheromone and a greedy heuristic known as visibility [6]. The amount of pheromone deposited on arcs is proportional to the quality of the solution generated and increases at run-time during the computation.

The Ant System (AS) is the first ACO algorithm which was applied for solving the Traveling Salesman Problem ([7], [8]). Some other early applications include the elitist strategy for Ant System (EAS) proposed by [8] , rank-based version of Ant System (AS_{rank}) by [9], Max-Min Ant System (MMAS) by [10], and Ant Colony System (ACS) by [11]. Since its first application many implementations of ACO have been proposed for a variety of combinatorial optimization problems such as quadratic assignment problem, scheduling problem, sequential ordering problem, vehicle routing problem and its variants, etc. We skip further discussion of the ACO and refer the interested reader to [12] for a complete review and details. Next, we describe our ACO implementation for our testing problem.

We define the pheromone trail τ_{ij} as the amount of pheromone deposited on arc (i, j) where i denotes the component and j denotes its position in the sequence. We also define η_i as the visibility value which shows the desirability of selecting component i early in the sequence.

Initialization: An initial amount of pheromone τ_0 is deposited on each arc. We have observed that $\tau_0 = 1/\rho C_0$, where ρ $(0 < \rho \leq 1)$ is the pheromone evaporation parameter and C_0 is the cost corresponding to an initial feasible solution. We obtain C_0 using the greedy heuristic algorithm described before.

Visibility values: Also known as the heuristic information, the visibility value η_i of a component i shows its degree of desirability to be placed first in the sequence. If η_i is large, then the algorithms will tend to give a higher selection chance to component i. We use the following visibility function:

$$\eta_i = (1 - p_i)/c_i \tag{10}$$

Testing Sequence Construction: we use the pseudo-random proportional rule introduced in ACS for constructing the test sequences. The number of ants we utilize is equal to the number of components that has no predecessors. Each ant initially places its associated component to the first position in its sequence and then constructs its testing sequence by successively selecting a component from the feasible components set N_i^k. For each ant k that has recently selected component i, N_i^k is formed by taking not yet selected components that do not violate the precedence constraints. If N_i^k is empty then all components have been sequenced. The choice of the next component to be placed is based on its attractiveness value, which is a function of the pheromone trail and visibility value:

$$\varphi_{ij} = \tau_{ij}^{\alpha} \eta_i^{\beta}. \tag{11}$$

where α and β are parameters to control the relative weight of trail intensity τ_{ij} and visibility η_i. An ant k may either select the most favorable (attractive) component or randomly select a component using the following selection rule:

$$j^k = \begin{cases} argmax_{j \in N_i^k} \varphi_{ij}, & \text{if } q \leq q_0 \\ J^k, & \text{otherwise} \end{cases} \tag{12}$$

where q is a random variable drawn from a uniform distribution $U[0, 1]$ and $q_0 (0 \leq q_0 \leq 1)$ is a parameter to control exploitation versus exploration. J_k is selected according to the

following probability distribution:

$$p_{ij}^k = \begin{cases} \frac{\varphi_{ij}}{\sum_{l \in N_i^k} \varphi_{il}}, & \text{if} j \in N_i^k \\ 0, & \text{otherwise} \end{cases} \quad (13)$$

Pheromone Update: The pheromone update includes the pheromone evaporation and pheromone reinforcement. The pheromone evaporation refers to uniformly decreasing the pheromone values on all arcs. The aim is to prevent the rapid convergence of the algorithm to a local optimal solution by reducing the probability of repeatedly selecting certain components in certain positions in the sequence. The pheromone reinforcement process, on the other hand, increases the pheromone values on the arcs belonging to the sequence of the best performing ant(s) at each iteration as well as from previous iterations. The aim is to increase the probability of selecting the arcs frequently used by the ants that construct the least costly sequences. In our pheromone update process we adopt a rank-based MMAS strategy. In this strategy, w best-ranked ants of each iteration are used to update the pheromone trails. The pheromone reinforcement of each ant is proportional to its rank. We allow the iteration-best ant to deposit additional pheromones. Our pheromone update rule is as follows:

$$\tau_{ij} := (1-\rho)\tau_{ij} + \sum_{r=1}^{w}(w-r)\Delta\tau_{ij}^r + w\Delta\tau_{ij}^* \quad (14)$$

In this formulation, ρ is the evaporation parameter as defined earlier and $\Delta\tau_{ij}^r = 1/C^r$ for all (i,j) pairs belonging to the sequence built by the r^{th} best ant where C^r is the cost of the corresponding sequence. "*" denotes the best solution obtained so far. Furthermore, if the pheromone level on any arc drops below an explicit lower limit or exceeds an explicit upper limit it is set equal to that limit. In other words, if any $\tau_{ij} < \tau_{min}$ $(\tau_{ij} > \tau_{max})$ then $\tau_{ij} = \tau_{min}$ $(\tau_{ij} = \tau_{max})$. The motivation of this strategy is to reduce the risk of a premature convergence.

C. Branch and bound algorithm

In order to test the effectiveness of the heuristic algorithms, we have implemented a branch and bound algorithm for the sequential testing algorithm after applying preprocessing procedures proposed in [4]. These preprocessing techniques are provided as reduction theorems in [4]. Loosely, the reduction theorems state that when certain conditions hold, either the number of nodes of the predecessor graph can be decreased by combining two tests into one or the precedence relations can be relaxed. These reductions are centered around terminal tests and ratios. Let us recall that the ratio of component i is defined as $c_i/(1-p_i)$.A test is said to be terminal if it has no successors in the precedence graph and nonterminal otherwise. Given this definition, we can summarize these reductions as follows:

a) Reduction A: Let j be a terminal test with an immediate predecessor i that has a better ratio. Then it is possible to update the predecessor graph by replacing the arc from i to j, with an arc from each immediate predecessor of i to j. The resulting predecessor graph would produce the same optimal solution with the original graph.

b) Reduction B: Let i be a nonterminal test with only terminal tests as its successors and j be the test with the minimum ratio among the successors of i. If j has no immediate predecessors other than i and ratio of j is not worse than i then i and j occurs one after another in an optimal solution.

Let us note that when the conditions for Reduction A hold, the precedence requirements are relaxed (the relaxation is most apparent if test i has no predecessors). So it is possible that the resulting precedence graph has no arcs in which case one could run the simple optimal algorithm for the case where there are no precedence constraints. On the other hand, if the conditions for Reduction B holds, it is possible to replace tests i and j with a single component with the testing cost of $c_i + p_i c_j$ and the probability of functioning is $p_i p_j$. Consequently, the number of nodes in the updated precedence graph is one less than the original graph.

These preprocessing rules are implemented before the branch and bound in order for the algorithm to conduct a more efficient search. Currently, we first attempt to apply Reduction A the predecessor graph to possibly reduce the number of precedence relations and then we apply Reduction B to reduce the number of nodes of the precedence graph.

After the preprocessing, a branch and bound algorithm is implemented in order to find the optimal solution. In the branch and bound algorithm each node corresponds to a partial feasible sequence of the tests. Let's suppose that the root has depth 0 and the depth of any other node is one more than the depth of its parent. Then the number of tests in the partial sequence is the depth of the node in the branch and bound tree. The lower bound of a node is calculated as if there are no precedence constraints for the tests that are not in the partial feasible sequence of the associated node. For each node, we also compute an upper bound by using the greedy algorithm mentioned above for the tests that are not in the partial feasible sequence of that node. In the branch and bound implementation, depth first search is used in order to find a feasible solution quickly.

IV. Computational Results

In order to test the effectiveness of the ant colony algorithm we have generated two classes of random instances of the problem. For the first class of problems, forest type precedence constraints are generated. Each tree in the forest is an out-tree meaning that the direction of the arcs are away from the root. We will refer to this class of problems as forest type problems. An example for such a precedence graph can be seen in figure 1.It is possible to find the optimal solutions for this class of problems with the algorithm of [4]. For this class, the number of components is taken as 20, 50, 100 and 200.

In the precedence graph the maximum outdegree is taken as 2 or 5. The cost of testing components is determined uniformly between 1 and 10. The probability vector is generated in three different ways: Uniform between 0 and 1, uniform between 0.5 and 1 and finally uniform between 0.75 and 1. For each set of parameters 10 random instances are generated. For forest type problems, in total we have 240 instances.

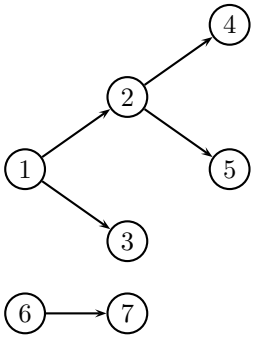

Fig. 1. An example forest type precedence graph for 7 components

For the general precedence case, we consider 12, 20, 50, 100 and 200 components. As in the previous case, the cost values are taken as uniform between 1 and 10. The probability vector is also generated the same way as before. In the general case, we have a parameter called intensity. For a precedence matrix for n components, the maximum number of non zero entries is $(n^2 + n)/2$. Intensity refers to the percentage of non zero entries out of this maximum number. We use 25%, 50% and 75% for the intensity parameter. Again for each set of parameters we generate 10 random instances. In total, we have 450 instances for the general case. We will refer to this class of problems as general type problems.

The ant colony algorithm is coded using C++. 10 runs of 100 iterations each were performed for each instance and best solution out of the 10 runs is recorded. The parameters were set according to initial experimental runs as: $q_0 = 0.5$, $\alpha = 1$, $\beta = 1$, $\rho = 0.05$, $\tau_{max} = 1/\rho C^*$, and $\tau_{min} = \tau_{max}/10$. The ant colony algorithm is run for forest type problems and general type problems. The running time of the ant colony algorithm is in the order of a few seconds.

The branch and bound algorithm is run only for the general type instances since it is possible to find the optimal solutions for the tree type problems by the algorithm in [4]. We run the branch and bound algorithm for at most 10 minutes for each instance and either the best solution found in 10 minutes or the optimal solution is reported. The number of optimal solutions out of the 450 general type instances that can be obtained by the branch and bound algorithm with respect to the number of tests and different intensity values is given in table I. Let us recall that intensity refers to the ratio of the number of arcs in the precedence graph to the maximum possible number of arcs. For each value of N, there are 90 instances in total. We observe that all instances with 12 and 20 tests are solved to optimality whereas no instance with 200 tests can be solved to optimality in 10 minutes. The instances that can be solved optimally for 50 or 100 tests are those with relatively more precedence constraints.

In addition, we run the greedy algorithm for all instances. Let us note that the solution provided by the greedy algorithm is the first solution found by the branch and bound algorithm.

For forest type problems, we compare the ant colony algorithm and greedy algorithm with the optimal solutions that are obtained by the polynomial time algorithm proposed by [4]. The average optimality gaps are better for the ant colony algorithm for all problem sizes in comparison with the greedy algorithm. The optimality gaps are from 4.90% to 9.49% for the greedy algorithm whereas the gaps are from 3.39% to 4.47% for the ant colony based algorithm. We observe that both the ant colony algorithm and the greedy algorithm tend to perform better for larger instances. This is probably due to the fact that missing a component that comes early in the optimal sequence cannot be compensated later when there are not many components. The optimality gaps are plotted against problem size in figure 2. Let us recall that the probabilities are generated randomly from a uniform distribution, from (0,1), (0.5,1) and (0.75,1). We observe that the optimality gaps of the algorithms get worse as randomly derived probabilities are closer to 1 from a narrower range. When the probabilities are derived from uniform(0,1), it is easier for the heuristic algorithms to distinguish the components in terms of the ratios. When the probabilities are closer to each other, the ratios are closer to each other and it is more likely to pick a wring component.

For general type problems, we plot the gap of the heuristic algorithms from the best solution found by the branch and bound in 10 minutes in figure 3. Again the ant colony algorithm outperforms the greedy algorithm on average for all problem sizes. The average gaps of the greedy algorithm from the best solutions found by the branch and bound algorithm range from 0% to 3.75 % whereas the same gaps are from -0.89% to 1.37%. The ant colony algorithm also outperforms the best solutions found by the branch and bound algorithm on average when the problem size is 50, 100 and 200. The ant colony algorithm performs best when $N = 200$. Since the greedy algorithm gives the first solution found by the branch and bound algorithm, the greedy algorithm can never beat the branch and bound algorithm. On the other hand when $N = 200$, the branch and bound algorithm cannot improve the first solution that it finds for all the 90 instances in 10 minutes. For this class of problems, the ant colony algorithm performs the best. For $N = 20$ where all problems are solved to optimality, it is interesting to observe that the optimality gaps for the general case are smaller than the optimality gaps of the tree type instances with 20 tests.

N	Solved to optimality	25%	50%	75%
12	90	30	30	30
20	90	30	30	30
50	66	7	29	30
100	28	0	8	20
200	0	0	0	0

TABLE I

NUMBER OF INSTANCES SOLVED TO OPTIMALITY WITH RESPECT TO NUMBER OF TESTS AND INTENSITY VALUES

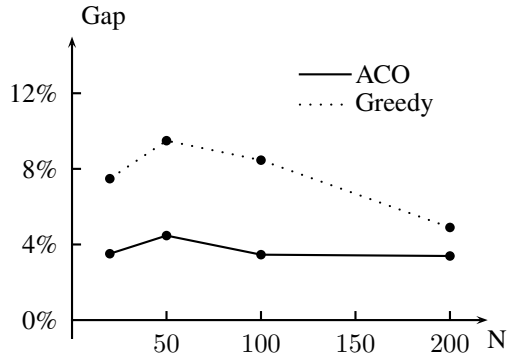

Fig. 2. Optimality gap of ACO and greedy with respect to N for forest type problems, for $N = 20, 50, 100, 200$

We observe that the greedy algorithm performs better as the intensity increases. This could be due to the fact that when there are fewer feasible instances the greedy algorithm cannot pick a completely wrong component since most of them are not even feasible. In addition, the way that the greedy algorithm resembles the the way the optimal algorithm for the case where there are no precedence constraints. For general type problems, we do not observe a structure in the results in terms of the way the probabilities are determined.

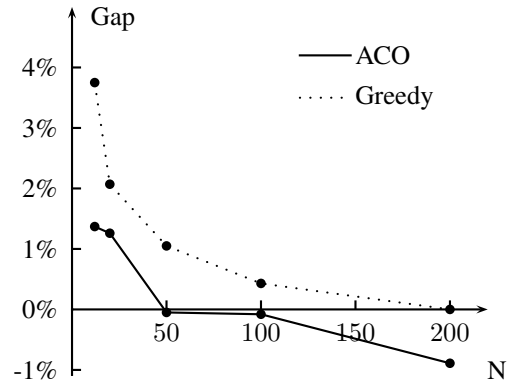

Fig. 3. Optimality gap of ACO and greedy with respect to N for general type problems, for $N = 12, 20, 50, 100, 200$

V. CONCLUSION AND FUTURE WORK

In this work, we develop an ant colony algorithm for the sequential testing of a series system under precedence constraints. We demonstrate the performance of the algorithm by comparing it with the solutions obtained by a special branch and bound algorithm. The experiments performed on random instances show that the ant colony algorithm produces satisfactory and robust results in relatively short computation times. As future work, it would be nice to have a formal proof of the NP-completeness of the problem. The performance of the ant colony algorithm may be improved by developing a more efficient visibility function and implementing a different pheromone update mechanism. In addition, a local search algorithm may be utilized to further enhance the solution quality, if needed. The branch and bound algorithm could be developed further in terms of branching strategies and by different bounding schemes. From a computational point of view, effective and efficient heuristic algorithms for k-out-of-n systems, which is a more general class of systems should be developed. To our knowledge, there are a few analytical results related to testing of k-out-of-n systems under precedence constraints yet no computational results are reported in the literature.

REFERENCES

[1] T. Ünlüyurt, "Sequential testing of complex systems: A review," *Discrete Applied Mathematics*, vol. 142,(1-3), pp. 189–205, 2004.

[2] L. Mitten, "An analytic solution to the least cost testing sequence problem," *The Journal of Industrial Engineering*, p. 17, January-February 1960.

[3] E. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints," *Annals of Discrete Mathematics*, vol. 2, pp. 75–90, 1978.

[4] M. Garey, "Optimal task sequencing with precedence constraints," *Discrete Mathematics*, vol. 4, pp. 37–56, 1973.

[5] S. Chiu, L. Cox, and X. Sun, "Optimal sequential inspections of reliability systems subject to parallel-chain precedence constraints," *Discrete Applied Mathematics*, vol. 97, pp. 327–336, 1999.

[6] M. Dorigo, "Ant colony optimization," Scholarpedia, http://www.scholarpedia.org/article/Ant_Colony_Optimization, last accessed in June 2008, 2008.

[7] ——, "Optimization, learning and natural algorithms (in italian)," Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.

[8] M. Dorigo, V. Maniezzo, and A. Colorni, "The ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man and Cybernatics-Part B*, vol. 26, pp. 29–41, 1996.

[9] B. Bullnheimer, R. Hartl, and C. Strauss, "An improved ant system algorithm for the vehicle routing problem," *Annals of Operations Reserach*, vol. 89, pp. 319–328, 1999.

[10] T. Stützle and H. Hoos, "The max-min ant system and local search for the traveling salesman problem," in *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC'97)*, T. Back, Z. Michalewicz, and X. Yao, Eds. NJ: IEEE Press, 1997, pp. 309–314.

[11] M. Dorigo and L. Gambardella, "Ant colony system:a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computing*, vol. 1, pp. 53–66, 1997.

[12] M. Dorigo and T. Stützle, *Ant colony optimization*. Cambridge, Massachussets: MIT Press, 2004.