

# Ant Colony Component-based System for Traveling Salesman Problem

Andreea Vescan and Camelia-M. Pinte

Department of Computer Science, Babeş-Bolyai University  
Cluj-Napoca, România  
{avescan,cmpinte}@cs.ubbcluj.ro

## Abstract

A component-based approach to model *Ant Colony System (ACS)* for the *Traveling Salesman Problem (TSP)* is shown in this paper. The used components to solve the *TSP* with *ACS* technique and the collaboration rules between components are also described. The internal reasoning about the ant colony component-based system for the *TSP* gives a better perception of solving this problem using ant-based techniques.

**Mathematics Subject Classification:** 05C85, 90C35, 68N19

**Keywords:** ant system, component-based development

## 1 Introduction

The *Ant Colony System (ACS)* algorithm has been introduced by Dorigo and Gambardella [3] to improve the performance of Ant System [2], that allowed to find good solutions within a reasonable time for great problems.

One of the most studied discrete optimization problem is the *Traveling Salesman Problem (TSP)*. Easy to formulate and with a large number of applications, *TSP* is one of the most popular difficult to solve problem. It is known that the *TSP* is  $\mathcal{NP}$ -complete.

For many combinatorial optimization problem, metaheuristics including: tabu search, simulated annealing genetic algorithms, ant systems provide the best tools for producing good quality approximate solutions.

Building software applications using components [7] significantly reduces development and maintenance costs. Components can often be reused to build new applications. Reuse components also serves to increase productivity. The

time and the maintenance costs are reduced because it is possible to make changes to a component's implementation without affecting all the systems that rely on it.

An Ant Colony component based system for *Traveling Salesman Problem* is introduced. Several components from the classical approach on modeling *TSP* [6] are reused. New specific components for *Ant Colony System* are introduced to solve the *TSP*. The new components could be reused in other ant-based techniques. Some components could be reused to solve using ant-based techniques other  $\mathcal{NP}$ -complete problems as: *Quadratic Selective Traveling Salesman Problem*, *Generalized Traveling Salesman Problem*, *Vehicle Routing Problem*, *Quadratic Assignment Problem*.

## 2 Ant Colony System for Traveling Salesman Problem

### 2.1 Ant Colony System

*Ant Colony System (ACS)* metaheuristics is a particular class of ant algorithms. The *ACS* is based on three modifications of *Ant System*: a different node transition rule, a different pheromone trail updating rule and the use of local and global pheromone updating rule, to favor exploration. An pseudocode for *ACS* algorithm it follows.

```

procedure ACS algorithm
begin
  Set parameters, initialize pheromone trails
  Loop
    Each ant is positioned on a starting node
    Loop
      Each ant applies a state transition rule to incrementally
        build a solution
        and a local pheromone updating rule
    Until all ants have built a complete solution
    A global pheromone updating rule is applied
  Until End_condition
end.

```

### 2.2 Traveling Salesman Problem solved with ACS

The *Traveling Salesman Problem* can be described as follows.

*Given a complete graph with weights on the edges (arcs), find a hamiltonian cycle in graph of minimum total weight.*

The brute-force method of explicitly examining all possible *TSP* tours is impracticable because there are  $\frac{(n-1)!}{2}$  different tours in the complete undirected graph and  $(n-1)!$  different tours in a complete directed graph.

Approximate algorithms start from a tour and iteratively improve it by changing some parts of it at each iteration. The best known tour improvement are based on edge exchange, as *k-opt* algorithms, *Lin* and *Kernighan* algorithm.

One of the most successful heuristic algorithm for solving *TSP* is *Ant System*. In the search mechanism for the *Traveling Salesman Problem* there are a number of factors to consider.

Initially the ants are placed randomly in the nodes of the graph.

At iteration  $t + 1$  every ant moves to a new node and the parameters controlling the algorithm are updated. Assuming that the *TSP* is represented as a fully connected graph, each edge is labeled by a trail intensity. Let  $\tau_{ij}(t)$  represent the intensity of trail edge  $(i, j)$  at time  $t$ . When an ant decides which node is the next move it does so with a probability that is based on the distance to that node and the amount of trail intensity on the connecting edge. The inverse of distance to the next node, is known as the *visibility*,  $\eta_{ij}$ . Visibility is defined as

$$\eta_{ij} = \frac{1}{d_{ij}}$$

where,  $d_{ij}$ , is the distance between nodes  $i$  and  $j$ .

At each time unit evaporation takes place. This is to stop the intensity trails increasing unbounded. The rate evaporation is denoted by  $\rho$ , and its value is between 0 and 1. In order to stop ants visiting the same node in the same tour a tabu list is maintained. This prevents ants visiting nodes they have previously visited.

To favor the selection of an edge that has a high pheromone value,  $\tau$ , and high visibility value,  $\eta$  a function  $p^k_{iu}$  is considered.  $J^k_i$  are the unvisited neighbors of node  $i$  by ant  $k$  and  $u \in J^k_i$ . According to this function may be defined as

$$p^k_{iu}(t) = \frac{[\tau_{iu}(t)][\eta_{iu}(t)]^\beta}{\sum_{o \in J^k_i} [\tau_{io}(t)][\eta_{io}(t)]^\beta}, \quad (1)$$

where  $\beta$  is a parameter used for tuning the relative importance of edge length in selecting the next node.

$p^k_{iu}$  is the probability of choosing  $j = u$  as the next node if  $q > q_0$  (the current node is  $i$ ).  $q$  is a random variable uniformly distributed over  $[0, 1]$  and  $q_0$  is a parameter similar to the temperature in simulated annealing,  $0 \leq q_0 \leq 1$ . If  $q \leq q_0$  the next node  $j$  is chosen as follows:

$$j = \operatorname{argmax}_{u \in J^k_i} \{ \tau_{iu}(t)[\eta_{iu}(t)]^\beta \} \quad (2)$$

After each transition the trail intensity is updated using the local rule:

$$\tau_{ij}(t+1) = (1 - \rho)\tau_{ij}(t) + \rho\tau_0. \quad (3)$$

Within *ACS* only the ant that generate the best tour is allowed to *globally* update the pheromone. The global update rule is applied to the edges belonging to the *best tour*. Let  $L^+$  be the length of the best tour. The correction rule is

$$\tau_{ij}(t+1) = (1 - \rho)\tau_{ij}(t) + \rho\Delta\tau_{ij}(t) \quad (4)$$

where  $\Delta\tau_{ij}(t)$  is the inverse length of the best tour.

The result of the algorithm is the shortest tour found. The algorithm computes for a given number of iteration  $t_{max}$  to find a good solution, the optimal solution if it is possible.

## 3 Modeling Traveling Salesman Problem with components

### 3.1 Component model. Specification and execution

*Component-based Software Engineering (CBSE)* is concerned with the development of systems from reusable parts, the development of components, system maintenance and improvement by means of component replacement or customization.

There are two issues [1] that need to be addressed where a software system is to be constructed from a collection of components. First there has to be a way to connect the components together. Then we have to get them to do what we want. Systems and schemes that address the issue of making pieces of software to fit together work by managing and controlling the interfaces between components. The other problem is more subtle and difficult. We need to ensure that the assembled system does what is required. One approach to solving this problem is to reason about and check the behavior of a system and the components from which it is built. Building models that are sufficiently formal to be executed and subjected to examination with model checking tools is one technique that can help. These models need to represent the particular aspects of the behavior of the system that we are concerned to check.

A general model to formalize component-based software systems was proposed in [8]. Starting from this component model a method for composing specified components is introduced in [5]. The method computes all the possibilities of combining the components, according to the specification of each

component. The model is used in [4] to develop a new specification, construction and execution of a component-based model.

In the following we present the specification of the component model introduced in [5].

**Definition 3.1** *A simple component  $C$  is specified using a 5-tuple of the form:*

$$C = ( \mathbf{name}, \{\mathbf{in}\}, \mathbf{out}, \mathbf{FC}, \mathbf{iC}), \text{ where :}$$

$\mathbf{name}$  represents the name of the component;  $\mathbf{in}$  represent the inputs for the component function;  $\mathbf{out}$  represents the output of the component function;  $\mathbf{FC}$  represent the component function  $\mathbf{FC} = (\mathbf{name}, \mathbf{evaluationCode})$  ( $\mathbf{name}$  represents the name of the function and  $\mathbf{evaluationCode}$  represents the evaluation code of the function) and  $\mathbf{iC}$  represents the component interface specified by describing the *inputAssumption* and the *outputGuarantee*.

**Definition 3.2** *Input specification. The input  $\mathbf{In}$  of a component function (or of the component) is specified using a 4-tuple of the form:*

$$\mathbf{In} = (\mathbf{name}, \mathbf{type}, \mathbf{semantic}, \mathbf{value}), \text{ where :}$$

$\mathbf{name}$  represents the name of the import;  $\mathbf{type}$  represents the type of the import;  $\mathbf{semantic}$  describes the semantic of the import and  $\mathbf{value}$  represents the value of the input at execution time.

**Definition 3.3** *Output specification. The output  $\mathbf{Out}$  of a component function (or of the component) is specified using a 4-tuple of the form:*

$$\mathbf{Out} = (\mathbf{name}, \mathbf{type}, \mathbf{semantic}, \mathbf{value}), \text{ where :}$$

$\mathbf{name}$  represents the name of the outport;  $\mathbf{type}$  represents the type of the outport;  $\mathbf{semantic}$  describes the semantic of the outport and  $\mathbf{value}$  represents the value of the outport at execution time.

In the following the execution elements [5] of the component-based model are shown. The execution of the component-based system is composed of sequences as

$$(Op_0, C_0), (Op_1, C_1), (Op_2, C_2) \dots$$

where for each  $i \geq 0$ ,  $Op_i$  is a subset of possible operations and  $C_i$  is a subset of components ready for execution.

The possible operations are: *-propagation* - this rule moves values that have been generated by a component along connections from the component's out port to other components; *-evaluation* - the component function is evaluated

and the result is passed to the output of the component. In a moment of execution time, the state is presented as follows:

$$State = (operation, componentForEval),$$

where:

-*operation* =  $\{C \rightarrow, C \equiv\}$ ;  $C \rightarrow$  is the propagation operation from component  $C$  and  $C \equiv$  is the evaluation operation of component  $C$ .

-*componentForEval* - a component ready for evaluation.

If both types of operation can be performed, the propagation operation is chosen.

### 3.2 Component-based classical approach for TSP

A *TSP* component-based system for an exact algorithm, *Backtracking* is introduced in [6]. Figure 1 is showing the minimalist structure of the model. The intern structure of the model shows the components involved in the computation of *TSP*.

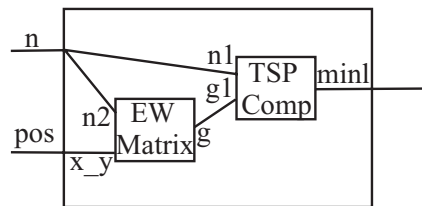


Figure 1: The minimalist structure of the Component-based Model for the Traveling Salesman Problem

The *Input data* are the number of nodes and the Euclidean coordinates for each node. The *Output data* is the minimal total length of a tour visiting each node exactly once.

The *EW\_Matrix* component used in the model computes the matrix of edge-weights using the Euclidean distances. Despite that Euclidean *TSP* is  $\mathcal{NP}$ -hard, was confirmed that Euclidean *TSP* is somewhat simpler than the general *STSP*. We will reuse some of these components to solve *TSP* using *Ant Colony System*.

## 4 Ant Colony Component-based System for TSP

### 4.1 Collaborative Components in ACS model for TSP

The component-based system for solving *TSP* using *ACS* has three components, as in Figure 2, involved in the computation: *EW\_Matrix* (reused from the classical approach), the *ANT\_TSP* component and the *Write* component. The *TSP\_Comp* component from the classical approach is changed with the *ANT\_TSP* component that performs the same task but using a different method. The *Write* component prints into a file the nodes of the tour and the minimum length found.

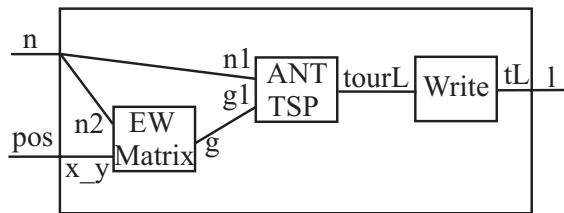


Figure 2: The structure for the Ant Colony Component-based System for TSP

The steps of the computation of the ant colony component model for *TSP* are described successively.

- $state_0 = (\{start \rightarrow\}, \{\})$ .  
 -*n* receives the initial value for the number of nodes;  
 -*pos* receives the values of the nodes coordinates;  
 -data is propagated through the connections to the components:  
 $n \rightarrow n1$ ;  $n \rightarrow n2$  and  $pos \rightarrow x\_y$ .
- $state_1 = (\{EW\_Matrix \equiv\}, \{Read\})$ .  
 -*EW\_Matrix*  $\equiv$  : the variable *g* will have the matrix of edge-weights computed using the Euclidean Distance.
- $state_2 = (\{EW\_Matrix \rightarrow\}, \{\})$ .  
 -*EW\_Matrix*  $\rightarrow$  : data is propagated through the connections to the *ANT\_TSP* component:  $g \rightarrow g1$ .
- $state_3 = (\{ANT\_TSP \equiv\}, \{ANT\_TSP\})$ .  
 -*ANT\_TSP*  $\equiv$  : computation of the minimal total length of a tour visiting each node of the given graph exactly once. The variable *tourL* receives the result of the computation.
- $state_4 = (\{ANT\_TSP \rightarrow\}, \{\})$ .  
 -*ANT\_TSP*  $\rightarrow$  data is propagated through the connection to the input of the *Write* component:  $tourL \rightarrow tL$ .
- $state_5 = (\{Write \equiv\}, \{\})$ .

-*Write*  $\equiv$  writes into a file the tour nodes and the length of the tour.

- $state_6 = (\{Write \rightarrow\}, \{\})$ .

-*Write*  $\rightarrow$  data is propagated through the connection to the output of the black box component:  $tL \rightarrow l$ .

- $state_7 = (\{\}, \{\})$ . -There are no more possibilities of applying either propagation or evaluation. The execution of the components involved in the system is finished and we have the result into the output variable of the component.

## 4.2 Internal reasoning of ACS model for TSP

In this section the internal reasoning about the Ant Colony component-based System for TSP is described. Two views are described: the iteration with all the ants and the usage of the global update rule and the complete tour construction for an ant and the usage of the update local rule.

Figure 3 describes the computation at a current time after each ant computes a completed tour. The parameters are not included in the figure, the scope being understanding the steps of computation. The condition  $cK$  is for checking if all the ants have completed a tour. While there is an ant  $k$  that has not performed a complete tour, the *Ant<sub>k</sub>* component computes the tour for that ant.

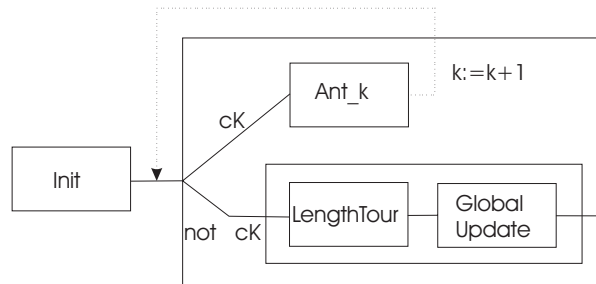


Figure 3: An iteration with all the ants and the usage of the global update rule

The *LengthTour* component computes the length for the tour and the *GlobalUpdate* component updates the pheromone. The global update rule is applied to the edges belonging to the best tour, as described in section 2.

The execution states for the representation on Figure 3 are described in the following. The execution of the components starts from  $state_3$  from the general computation from section 4.1.

- $state_{3a} = (\{Init \equiv\}, \{Init\})$ .

-Initialize the variables need for computation.

-The evaluation of the condition  $c_k : k \leq m$ , where  $m$  represents the number of ants.

- $state_{3a'} = (\{Init \rightarrow\}, \{\})$ .

- $Init \rightarrow$  the evaluation of the condition  $c_k$  is propagated: if the condition is true then the next component to be executed is  $Ant_k$ , otherwise follows the  $LengthTour$  component.

- $state_{3b1} = (\{Ant\_k \equiv\}, \{Ant\_k\})$ .

-A complete tour for the  $k$  ant is computed and the index for the next ant is increased:  $k := k + 1$ .

- $state_{3b1'} = (\{Ant\_k \rightarrow\}, \{\})$ .

-The execution goes to the  $state_{3a}$  where the evaluation of the  $c_k$  component is performed again.

- $state_{3b2} = (\{LengthTour \equiv\}, \{LengthTour\})$ .

-The length of the tour is computed.

- $state_{3b2'} = (\{LengthTour \rightarrow\}, \{\})$ .

-All the necessary data is propagated to the input of the  $GlobalUpdate$  component.

- $state_{3b3} = (\{GlobalUpdate \equiv\}, \{GlobalUpdate\})$ .

-The global update of the pheromone is performed.

- $state_{3b3'} = (\{GlobalUpdate \rightarrow\}, \{\})$ .

-A complete tour for all the ants is computed for the current time iteration.

-The next iteration will be performed, starting from the state  $Init$ .

Figure 4 describes the computation of a complete tour for an ant  $k$ . While not all the nodes were included in the tour (the condition  $c_{iN}$  means that  $c_{iN} < n$ , with  $n$  number of nodes ) a next node for the tour is selected computed using the rules from section 2.

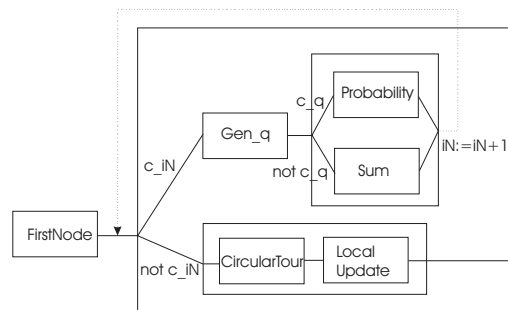


Figure 4: A complete tour construction for an ant and the usage of the update local rule

The  $Gen\_q$  component generate a random number  $q$ . The condition  $c_q$  is  $q \geq q_0$ . Two components for each condition evaluation are presented in

the model: *Probability* component and *Sum* component for the computations described in section 2. A local pheromone updating rule is applied using the *LocalUpdate* component.

The execution states for Figure 4 are described in the following. The execution of the components starts from  $state_{3b1}$  from the above description steps.

- $state_{fN} = (\{FirstNode \equiv\}, \{FirstNode\})$ .

-Random positioning of the  $k$  ant in the first node of the tour.

- $state_{fN'} = (\{FirstNode \rightarrow\}, \{\})$ .

-The evaluation of the condition  $c_{iN} : iN \leq n$ , where  $n$  represents the number of nodes.

-The evaluation of the condition  $c_{iN}$  is propagated: if the condition is true then the next component to be executed is *Gen<sub>q</sub>*, otherwise the *CircularTour* component.

- $state_{gQ} = (\{Gen\_q \equiv\}, \{Gen\_q\})$ .

-Generation of the  $q$  number.

-The evaluation of the condition  $c_q : q_0 \leq q$ , where  $q_0$  is a constant.

- $state_{gQ'} = (\{Gen\_q \rightarrow\}, \{\})$ .

-The evaluation of the condition  $c_q$  is propagated: if the condition is true then the next component to be executed is *Probability*, otherwise the *Sum* component.

- $state_{nP} = (\{Probability \equiv\}, \{Probability\})$ .

-The next node in the tour is computing using the probability described in section 2.

- $state_{nP'} = (\{Probability \rightarrow\}, \{\})$ .

-The node computed using the probability is propagated and the number of nodes in the current tour is increased:  $iN := iN + 1$ .

-The execution goes to the  $state_{fN}$  where the evaluation of the  $c_{iN}$  component is performed again.

- $state_{nS} = (\{Sum \equiv\}, \{Sum\})$ .

-The next node in the tour is chosen using the formula from section 2.

- $state_{nS'} = (\{Sum \rightarrow\}, \{\})$ .

-The chosen node is propagated and the number of nodes in the current tour is increased:  $iN := iN + 1$ .

-The execution goes to the  $state_{fN}$  where the evaluation of the  $c_{iN}$  component is performed again.

- $state_{cT} = (\{CircularTour \equiv\}, \{CircularTour\})$ .

-The tour becomes circular by adding on the last position the first node form the current tour.

- $state_{cT'} = (\{CircularTour \rightarrow\}, \{\})$ .

-All the necessary data are propagated to the *LocalUpdate* component.

- $state_{lU} = (\{LocalUpdate \equiv\}, \{LocalUpdate\})$ .

-The correction rule from section 2 is applied.

- $state_{U'} = (\{LocalUpdate \rightarrow\}, \{\})$ .

-The current tour for the  $k$  ant is completed and propagated outside the component.

## 5 Conclusions

Ant algorithms are based on the real world phenomena that ants are able to find their way to a food source and back to their nest, using the shortest route. The component based model for solving the *Traveling Salesman Problem (TSP)* with *Ant Colony System* is introduced. The *TSP* components, the internal reasoning about the ant colony component-based system and the rules of collaboration are described.

## References

- [1] I. Crnkovic, M. Larsson, Building Reliable Component-Based Software Systems, *Artech House publisher*, 2002.
- [2] M.Dorigo, Optimization, Learning and Natural Algorithms (in Italian), Ph.D thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, (1992), 1-140.
- [3] M. Dorigo, L. M. Gambardella, Ant Colony System: A cooperative learning approach to the Traveling Salesman Problem, *IEEE Trans. Evol. Comp.*, **1**, (1997), 53-66.
- [4] A. Fanea, Specification, Construction and Execution of a Component-Based Model, *Proceeding of the Symposium Cluj-Napoca Academical Days*, (2005), 87-92.
- [5] A. Fanea, S. Motogna, A Formal Model for Component Composition, *Proceedings of the Symposium Cluj-Napoca Academical Days*, (2004), 160-167.
- [6] A. Fanea, C-M. Pintea, Component model for a NP-hard problem, *An.Univ.Oradea Fasc.Mat., Univ.Oradea*, (2005) vol.XII, 91-100.
- [7] K. McInnis, Component-based Development, The Concepts, Technology and Methodology, *Castek Software Factory Inc.*, 2000.
- [8] Baoming Song, A General Model for Component-based Software, *Master Thesis, Dalhousie University DalTech*, 2000.

**Received: December 14, 2006**