A Java Framework for Ant Colony Systems

Ugo Chirico
Siemens Informatica S.p.A.
Via del Maggiolino, 151/163 00155 Rome, Italy
e-mail: ugos@ugosweb.com
home: http://www.ugochirico.com

Abstract. This paper describes an Object-Oriented framework written in Java designed to implement Ant Colony Systems. Thanks to Object-Oriented methodology the proposed framework can be reused and adapted to implement Ant Colony Systems able to solve specific problems such as the Travel Salesman Problem, the Multicasting Problem in a Network, the Steiner Problem on Networks and so on.
In the first part of this paper we introduce the framework showing how it has been designed and what are the base abstract classes the composed it. In the second part we describe how we adapted the framework to solve the Travel Salesman Problem showing how we specialized the base classes and what are the results we found. Finally in the third part we further specialize the framework to solve the Multicasting Problem in a Network, seen as an instance of the Steiner Problem on a Network. We describe how the Steiner problem can be solved by using an Ant Colony System and how the classes for the Travel Salesman Problem can be adapted to solve it, showing, at the end, the results we found.

1 Introduction

It's common opinion that Object-Oriented Programming is a good methodology to develop modular software that can be adapted and reused more times with few changes. Usually, such a modeling and programming methodology starts by designing an abstract model which cover the main and more general aspects of a problem, seen in its most general instance, and goes on with several specializations derived from the starting model, each of them solving a particular instance of the problem. In other words, if the main model and the derived ones have been designed with a high level of modularity and abstraction they can be reused, or sometimes adapted, for other similar problems with very few, (or, in the best case, without), changes dropping down the costs of the software development.
Java language, more then its competitors and predecessors such as C++, Smalltalk, etc., supplies a simply and elegant approach in designing and developing software using such a methodology and has contributed to diffuse, more than before, the use of the Object-Oriented Programming.
Following this philosophy we carried out a Java Framework for Ant Colony System in order to have a basic skeleton with an high level of abstraction by which one can implement and solves problems, in the set of problems solvable by ACS, without writing bulky lines of code, but simply specializing with few lines, the basic classes supplied by the framework.
In the next paragraph we'll show the classes which compose the framework. In the subsequent chapters we'll specialize the framework to solve the Travel Salesman Problem (TSP) [1] and, finally, we further derive from the classes for TSP to create a specialized model able to solve the Multicasting Problem [3] in an actual network seen as an instance of the Steiner Problem on a network [2].

2 Java Ant Colony System Framework

Referring to the paper wrote by M. Dorigo and L.M. Gambardella [1], Ant Colony System consists of a set of cooperating agents called *ants* that cooperate to find a good solution for optimization problems on graphs similar to the Travel Salesman Problem. Each single ant reflects a very trivial behavior: it simply goes from a node to another across an arc, but when all ants cooperate, like actual ants do in a real colony, the whole system reveals an intelligent behavior, as much as it is able to find a good solution for the TSP.
In this work we took the same assumptions made in the mentioned paper:

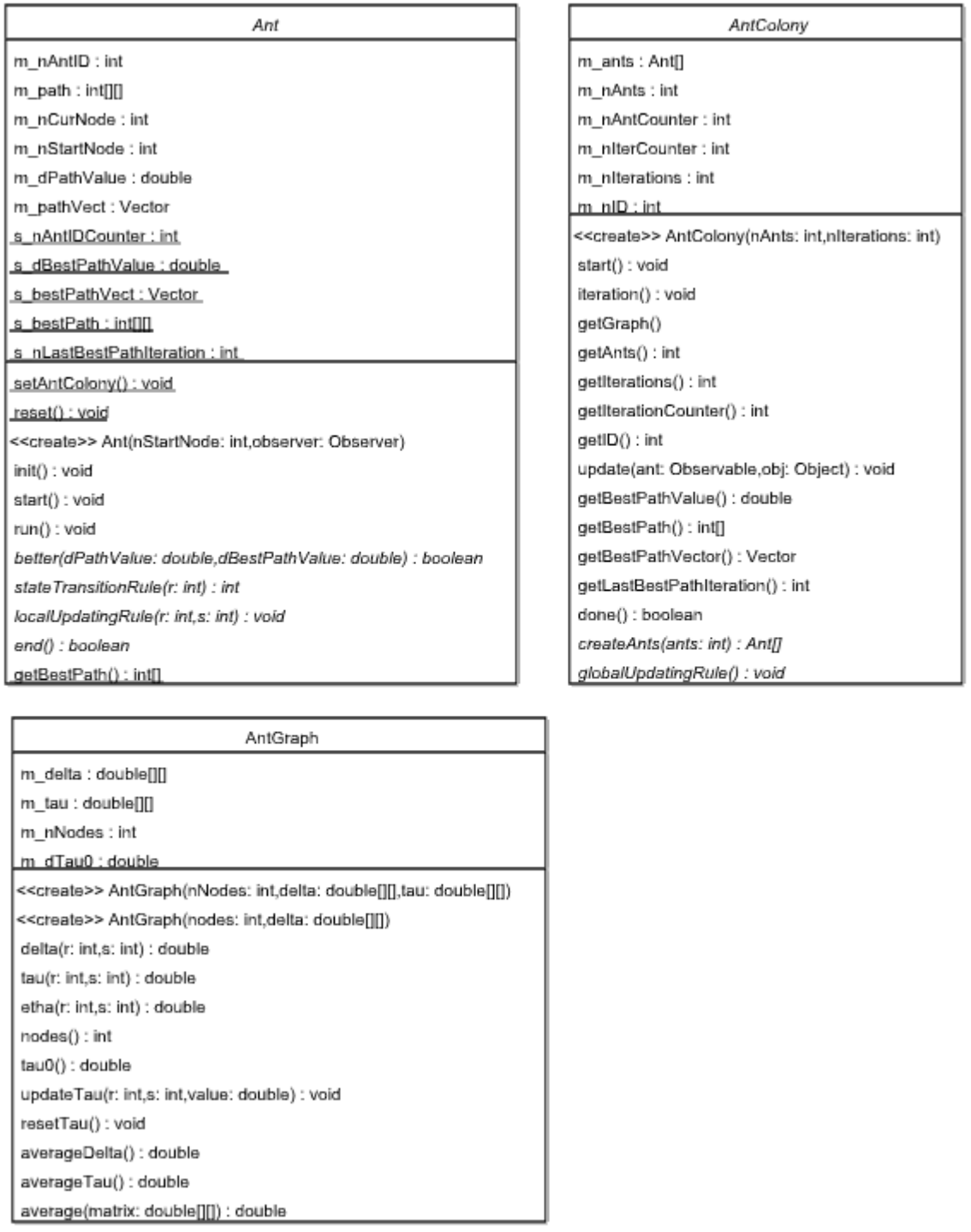
- the problem is represented by a graph $G = (N, A)$ where N is the set of nodes and A is the set of arcs which connect the nodes;
- cooperation between ants is made by using an indirect form of communication mediated by pheromones they deposit on the arcs of the graph representing the problem;
- the functions *cost measure* $\delta(r, s)$ and *desirability measure* $\tau(r, s)$, where $r, s \in N$, are defined on the graph;
- the rules defining the behavior of the ants and the whole colony are the same as those proposed in the mentioned paper:

- a *State Transition Rule* which brings the concrete ant from a node to another across an arc;
- a *Local Updating Rule* which updates the pheromones deposited by the ant on the arc it walked in;
- a *Global Updating Rule* which updates the pheromones deposited on the arcs when an ant ends its trip;

In addition we defined a *Comparison Function* which says if a given path is better then another and an *End Of Activity Rule* which specifies when an ant has completed its trip.

Following those assumptions, we started to design our framework. We considered three entities which play together in an Ant Colony System: a set of ants, an ant colony which coordinates all ants and a graph which defines the topology of the space where the ants move (or, from another point of view, where the colony lives).

Then, we mapped these entities into an object model using the Object-Oriented methodology and we obtained the object model shown in the Picture 1 using UML (version 1.3) notation.



Picture 1 Object Model of the JACSF in UML (version 1.3) notation.

The framework is composed by three classes:

- *Ant* is an abstract class which implements the general behavior of an artificial Ant. It must be specialized, in the context of the specific problem, in order to behave as a concrete Ant. As can be seen in the Picture 1, the abstract methods which must be implemented in a concrete derived class are *stateTransitionRule*, *localUpdatingRule*, *compare* and *end* which correspond to the rules mentioned above. The method *start* launches a thread in which the ant works. This means that each ant works in its own thread in parallel with all others giving to ACS the parallelism shown by actual ant colonies. The other class methods and members, such as *init*, *m_nCurNode*, etc., are used internally to perform the activity of the ant. The algorithm which drives the artificial ant during its trip is shown in Picture 2. Picture 10 in appendix A proposes a possible implementation in Java.
- *AntColony* is an abstract class implementing the general behavior of an Ant Colony System. It must be specialized, in the context of the specific problem, in order to implement a concrete Ant Colony. The abstract methods which must be implemented in a concrete derived class are *createAnts* and *globalUpdatingRule*. The first one creates the instances the class *Ant* which walk on the edges of the graph. The second one matches with the corresponding rule mentioned above. The other class methods and members, such as *start*, *iteration*, *m_ants*, etc., are used internally to perform and track the activity of the whole ant colony. The algorithm which directs the colony is shown in Picture 3. Picture 9 in appendix A illustrates a possible implementation in Java.
- *AntGraph* is a class which realizes the features of a graph specific for Ant Colony Systems. It supplies the values for the *cost measure* $\delta(r, s)$ and for the *desirability measure* $\tau(r, s)$ and gives a method to update the value $\tau(r, s)$. Because the ants work really in parallel they access asynchronously and unpredictably to the graph. To avoid that an ant tries to read from the graph while another are writing on, the accesses to *AntGraph* objects are synchronized by using the synchronization mechanism supplied by Java language.

The resulting framework, that we named *Java Ant Colony System Framework* or JACSF, forms an *Application Programming Interface* (API) which allows straightforwardly to realize artificial ant colonies in Java. In fact, by design, is necessary to create two classes, derived respectively from *Ant* and *AntColony*, implementing only the *Transition Rule*, *Local Updating Rule*, *Global Updating Rule*, *End Of Activity* and the *Comparison Function* in the related abstract methods, leaving out definitely all the aspects related to the algorithms that drives the general activities of the ants and the colony which has been implemented once and for all in the base classes of the framework.

```
Iterate
    Call State Transition Rule to find the next node
    Update the value of the trip adding the value of the arc which bring to the next node
    Call Local Updating Rule to update the pheromone level and the selected arc
Until End of Activity returns true
Call Comparison Function to compare the value of the current route with the value of the best tour.
If the current tour is better then the best tour update the best tour.
```

Picture 2 Algorithm of a single artificial ant

```
For all ants
    Create a new ant
    Set the starting node for newly created ant
    Create a new thread
    Attach newly created ant to newly created thread
End For

Loop
    Run all ants in their related threads
    Wait until all ant have completed their trip
    Call Global Updating Rule
    Increment iteration counter
Until iteration counter = total number of iteration
```

Picture 3 Algorithm of an Ant Colony

The source code of the proposed *Ant*, *AntColony* and *AntGraph* classes and the API's reference manual are available for download at: http://www.ugochirico.com.

In the next paragraphs we'll see how the classes *Ant* and *AntColony* have been specialized to solve TSP and the Multicasting Problem in a network.

3 JACSF applied on TSP

To apply JACSF to the Travel Salesman Problem we took the instance of ACS, defined in [1], solving the Travel Salesman Problem and we implemented it using JACSF.

The Picture 4 shows the equations, described in [1], which define the *State Transition Rule* (Eq. **(1)** and **(2)**), the *Local Updating Rule* (Eq. **(3)**) and the *Global updating Rule* (Eq. **(4)**)

$$s = \begin{cases} \arg\ \max\left\{ [\tau(r,u)] \cdot [\eta(r,u)]^{\beta} \right\} & \text{if } q \le q_0 \\ S \ \text{(selected by using b))} & \text{otherwise} \end{cases} \quad \textbf{(1)}$$

$$p_k(r,s) = \begin{cases} \dfrac{[\tau(r,u)] \cdot [\eta(r,u)]^{\beta}}{\sum_{u \in J_k} [\tau(r,u)] \cdot [\eta(r,u)]^{\beta}} & \text{if } s \in J_k \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(2)}$$

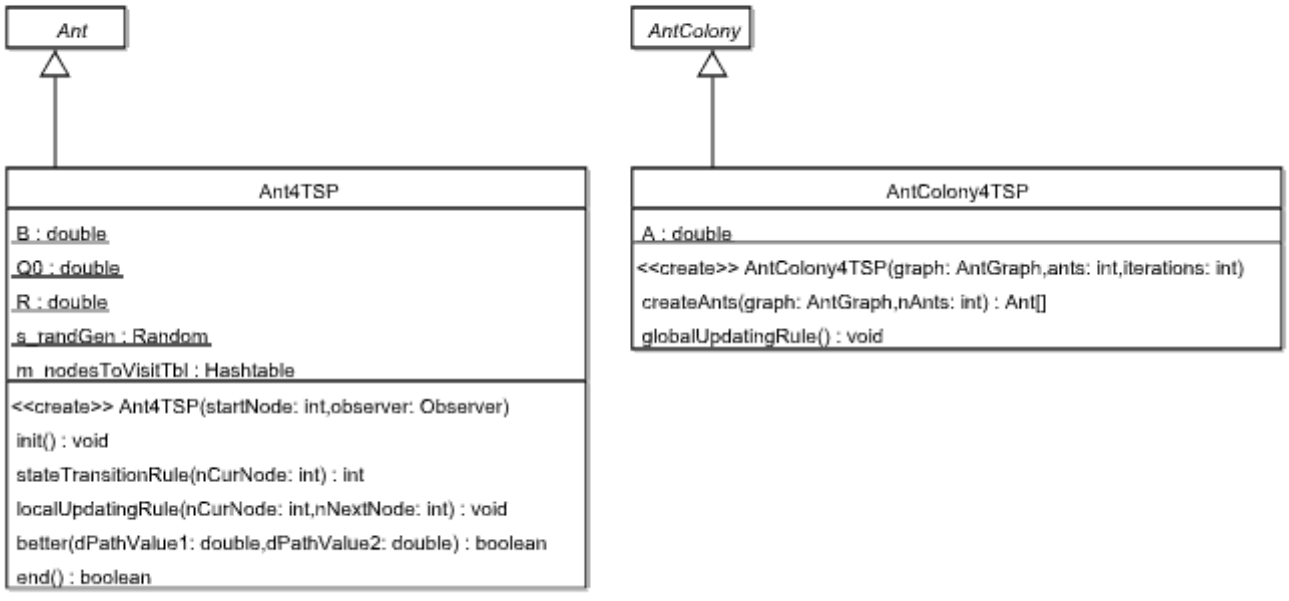
$$\tau(r,s) = (1-\rho) \cdot \tau(r,s) + \rho \cdot \Delta\tau(r,s) \quad \textbf{(3)}$$

$$\tau(r,s) = (1-\alpha) \cdot \tau(r,s) + \alpha \cdot \Delta\tau(r,s) \quad \textbf{(4)}$$

$$\text{where } \Delta\tau(r,s) = \begin{cases} (L_{gb})^{-1} & \text{if } (r,s) \in \text{global} - \text{best} - \text{tour} \\ 0 & \text{otherwise} \end{cases}.$$

Picture 4 Set of rules of ACS as described in [1]. **(1)** and **(2)** compose the *State Transition Rule*; **(3)** is the *Local Updating Rule*; **(4)** is the *Global Updating Rule*; J_k is the set of nodes not yet visited by the ant k.

We designed two concrete classes *Ant4TSP* and *AntColony4TSP*, derived respectively from the classes *Ant* and *AntColony*, which implement the transition and updating rules described above. The object model of the resulting class hierarchy, reported in UML notation, is shown in Picture 5.



Picture 5 Object Model derived from JACSF for TSP

AntColony4TSP implements in the method *globalUpdatingRule* the rule **(4)** in Picture 4 and defines in the member variable *A* the value of the parameter α. Picture 11 in appendix A shows the implementation of such a method in java. *Ant4TSP* implements in the method *stateTransitionRule* the *State Transition Rule* mixing the rule **(1)** and **(2)** in Picture 4 while in the method *localUpdatingRule* it implements the rule **(3)**. In the member variables *B*, *R*, *Q0* it defines respectively the values for the parameters β, ρ, q_0. Picture 12 and Picture 13 in appendix A show a possible implementation of both methods in java. The source code for *Ant4TSP* and *AntColony4TSP* classes is available at: http://www.ugochirico.com.
As suggested in [1] we set $A = 0.1$, $B = 2$, $R = 0.1$. The parameter q_0 has been set to $Q0 = 0.8$ because, after several tests, we found a better behavior of the colony with this value. Finally, the initial value for $\tau(r,s)$, τ_0, has been set using the Eq. **(5)** and the starting node for each ant is selected randomly.

$$\tau_0 = \frac{2 \cdot n}{\sum \delta(r,s)} \quad \text{where } n \text{ is the number of nodes in } G \qquad \textbf{(5)}$$

Picture 6 Initial value for $\tau(r, s)$

In order to test the proposed system we launched a colony composed by 10, 20 and 30 ants on a graph containing 50 cities for 2500 iterations. The results we found, shown in the Table 1, matches with the ones we expected and agrees with the results presented in [1].

Table 1 Results of an ant colony for TSP composed by 10, 20 and 30 ants on 50 cities for 2500 iterations

Ants	Value of the found best route	Average value for the best route	Std. Dev.	Iterations needed to find the best route
10	1,828	1.903	0,060	1556
20	1,805	1,906	0,056	956
30	1,822	1,906	0,054	371

4 JACSF applied to Multicasting Problem in a network

Multicasting in a network is the targeting of a single data packet to a selected set of receivers in the network. It is in opposition to traditional communication modes *unicast* and *broadcast* that are, respectively, one-to-one and one-to-all communications.
Efficient multicasting is a fundamental issue for the success of several real-time internet services involving at the same time a given number of clients and a great amount of data, such as video conferencing, computer supported co-operative work applications, e-teaching, and so on. Such typical multimedia applications communicate using a collection of information formats, such as audio and video, which can be classified as continuous and time dependent media and which must be sent continuously to each client with a very few delay, in order to give him the impression of a real-time application.
Considering the needs of efficiency, expressed in term of low delay during transmissions and limited bandwidth, solving the Multicasting Problem means generating a tree of nodes from the sender to the given set of clients which has the minimal cost in term of time and bandwidth.
Because the Multicasting Problem in networks can be seen as an instance of Steiner Problem on a network [2], we moved our attention on it.
The Steiner Problem, as described in [2], can be represented by a graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs connecting the nodes, a set $Z \subseteq N$ (referring to Multicasting Problem Z is the set of clients which must receive the data packets) and a *cost measure* $\delta(r, s) : G \longrightarrow \boldsymbol{R}$ which determines the cost of the arc from the node r to the node s. Solving the Steiner Problem means finding a set $T \subseteq G$, such that $Z \subseteq T$, having the minimum cost. Formally:

1. $Z \subseteq T \subseteq G$
2. $\delta = \sum_{s,r \in T_{ver}} \delta(s,r)$ is minimum

If $\delta(r, s)$ always is positive the subset T is a tree called *Steiner Tree* and it gives a solution to the corresponding Multicasting Problem.
Because Steiner Problem is very similar to TSP, we adapted our ACS for TSP to solve it. We made the following assertions:

- *State Transition Rule*, *Local Updating Rule* and the *Comparison Function* are the same as the ACS for TSP described in the previous chapter;
- *Global Updating Rule* is modified in the following way: evaporation of pheromones is applied on an arc only if it brings to a node not in Z (see Eq. **(6)**). Such a modification allows to make "less desirable" arcs which doesn't bring to nodes in Z and its global effect is to limit the number of nodes in T which tend to the number of nodes in Z;
- *End of Activity Rule* returns true when an ant has covered all nodes in Z;
- the initial value $\tau(r, s)$ is τ_0 for each arc between nodes not in Z (as in ACS for TSP) and $\lambda\tau_0$ (where $0<\lambda<1$) for each arc between nodes in Z. In this way we suggest to our ants a "preferred tour" which covers the nodes in Z.
- the *starting node* for each ant is selected from Z.

$$\tau(r,s) = (1-\alpha)\cdot\tau(r,s) + \alpha\cdot\Delta\tau(r,s) - \alpha\cdot W\cdot\Delta\tau_k \qquad \textbf{(6)}$$

$$\text{where } W = \begin{cases} 1 & if \quad s \notin Z \\ 0 & otherwise \end{cases}$$

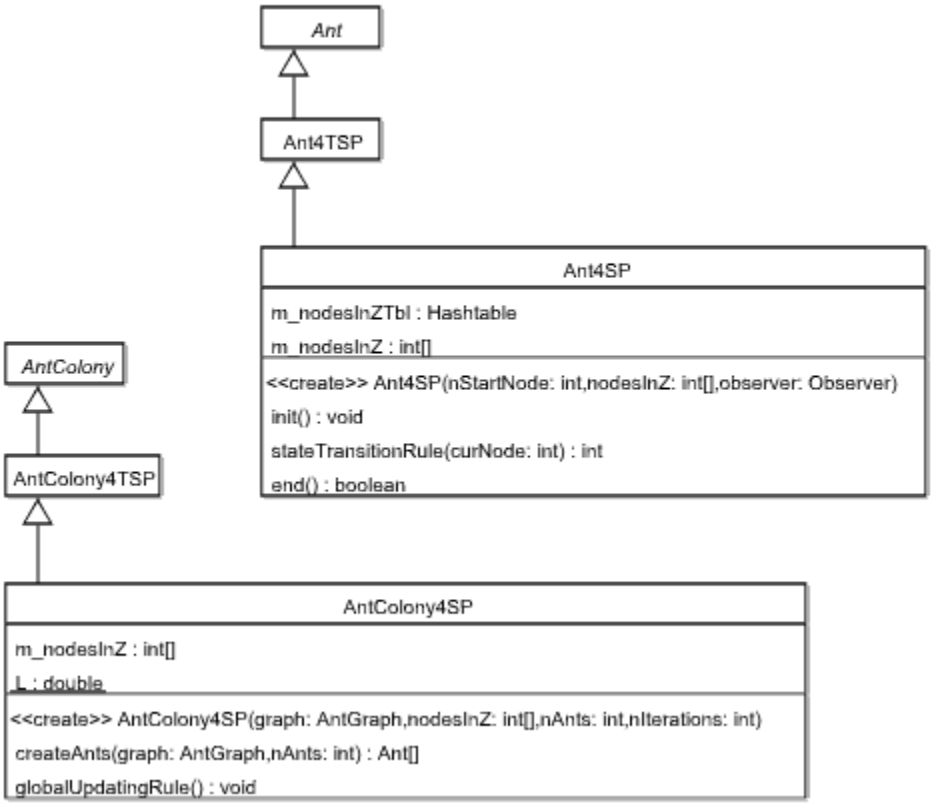
$$\text{and } \Delta\tau(r,s) = \begin{cases} (L_{gb})^{-1} & \text{if } (r,s) \in \text{global} - \text{best} - \text{tour} \\ 0 & \text{otherwise} \end{cases}.$$

Picture 7 *Global Updating Rule* modified to solve the Steiner Problem.

Then, we designed two classes *Ant4SP* and *AntColony4SP*, derived respectively from the *Ant4TSP* and *AntColony4TSP*. We opted to derive from the classes designed for TSP because there are very few changes to do respect to the base classes. The object model of the resulting class hierarchy is depicted using UML notation in Picture 8.
The class *AntColony4SP* differs from *AntColony4TSP* in the method *globalUpdatingRule*, defined in the Eq. **(6)**, and in the method *createAnts* in which it creates the ants setting for each of them a starting node selected from Z. Its *constructor* sets the initial value $\tau(r, s)$ to $\lambda\tau_0$ for each arc between nodes in Z and τ_0 for other arcs (as in ACS for TSP). Finally the parameter λ is defined by the member variable L. Picture 14 in appendix A shows a possible implementation of the Global Updating Rule in Java.
The class *Ant4TSP* differs from *Ant4TSP* only in the method *end* because the *End of Activity Rule* must return true when all nodes in Z has been covered (the source code for *Ant4SP* and *AntColony4SP* classes is available at: http://www.ugochirico.com)
We set $A = 0.1$, $B = 2$, $R = 0.1$ and $Q0 = 0.8$, $L = 0.5$ and we launched a colony composed by 10, 20 and 30 ants on a graph containing 50 nodes, with $|Z| = 10$ nodes, for 2500 iterations.
Table 2 reports the results we found. Again, the results we found matches with the one we expected.

Table 2 Results of an ant colony for the Steiner Problem composed by 10, 20 and 30 ants with $|G| = 50$ nodes and $|Z| = 10$ nodes for 2500 iterations.

Ants	Value of the found best route	Number of nodes in the best tour	Average value for the best route	Std. Dev.	Iterations needed to find the best route
10	0,665	22	0,709	0,059	1466
20	0,623	21	0.702	0,051	812
30	0,596	19	0,701	0,045	2208



Picture 8 Object Model, derived from JACSF, designed for Steiner Problem on networks

5 Conclusions

In this work we have explained how an Ant Colony System can be easily implemented in Java, proposing also an implementation for solving the Travel Salesman Problem and the Steiner Problem on a network. The proposed framework is only one of the possible implementation in Java but, we find that it is a good starting point to make a good implementation for a given problem. Currently we are using the implementation for the Steiner Problem to solve an actual Multicasting Problem on a network for one of our customer and now we are dealing with the values to give to the parameters in order to have the best performances. In the near future we hope to apply the framework to other optimization problems.

References:

[1] Dorigo, M., Gambardella, L.M., "Ant Colony System: A Cooperative Learning Approach to the Travel Salesman Problem", TR/IRIDIA/1996-5, Université Libre de Bruxelles
[2] Dreyfus S. E., Wagner, E.A., "*The Steiner Problem in graphs"*, Networks, **1**, 1972, pp. 195-207
[3] Kompella V.P., Pasquale, J.C., Polyzos, G.C., *"The Multicasting multimedia problem"*, Tech. Rep. CS93-313, September 1993

Appendix A: Implementation in Java

```
// creates all ants
m_ants  = createAnts(m_graph, m_nAnts);

m_nIterCounter = 0;

// loop for all iterations
while(m_nIterCounter < m_nIterations)
{
    // run an iteration
    m_nAntCounter = 0;
    m_nIterCounter++;

    // start all ants
    for(int i = 0; i < m_ants.length; i++)
    {
        m_ants[i].start();
    }

    /// wait while all ants have completed their trip
    try
    {
        wait();
    }
    catch(InterruptedException ex)
    {}

    // synchronize the access to the graph
    synchronized(m_graph)
    {
        // apply global updating rule
        globalUpdatingRule();
    }
}
```

Picture 9 Possible implementation in Java of the algorithm of an Ant Colony

```
// repeat while End of Activity Rule returns false
while(end() == false)
{
    int nNewNode;

    // synchronize the access to the graph
    synchronized(graph)
    {
        // apply the State Transition Rule
        nNewNode = stateTransitionRule(m_nCurNode);

        // update the length of the path
        m_dPathValue += graph.delta(m_nCurNode, nNewNode);
    }

    // add the current node the list of visited nodes
    m_pathVect.addElement(new Integer(nNewNode));
    m_path[m_nCurNode][nNewNode] = 1;

    // synchronize the access to the graph
    synchronized(graph)
    {
         // apply the Local Updating Rule
         localUpdatingRule(m_nCurNode, nNewNode);
    }

    // update the current node
    m_nCurNode = nNewNode;
}

// synchronize the access to the graph
synchronized(graph)
{
    // if the current tour is better then the current best tour
    // update the best tour value
```

```
    if(better(m_dPathValue, s_dBestPathValue))
    {
        s_dBestPathValue         = m_dPathValue;
        s_bestPath               = m_path;
        s_bestPathVect           = m_pathVect;
        s_nLastBestPathIteration = s_antColony.getIterationCounter();

    }
}
```

Picture 10 Algorithm of a single artificial ant implemented in Java

```
protected void globalUpdatingRule()
{
    double dEvaporation = 0;
    double dDeposition  = 0;

    for(int r = 0; r < m_graph.nodes(); r++)
    {
        for(int s = 0; s < m_graph.nodes(); s++)
        {
            if(r != s)
            {
                // get the value for deltatau
                double deltaTau =
                         ((double)1 / Ant4TSP.s_dBestPathValue) * (double)Ant4TSP.s_bestPath[r][s];

                // get the value for phermone evaporation
                dEvaporation = ((double)1 - A) * m_graph.tau(r,s);

                // get the value for phermone deposition
                dDeposition  = A * deltaTau;

                // update tau
                m_graph.updateTau(r, s, dEvaporation + dDeposition);
            }
        }
    }
}
```

Picture 11 Possible implementation in Java of the Global Updating Rule

```
public int stateTransitionRule(int nCurNode)
{
    // generate a random number
    double q    = s_randGen.nextDouble();
    int nMaxNode = -1;

    if(q <= Q0)  // Exploitation
    {
        double dMaxVal = -1;
        double dVal;
        int nNode;

        // search the max of the value as defined in Eq. a)
        Enumeration enum = m_nodesToVisitTbl.elements();
        while(enum.hasMoreElements())
        {
            // select a node
            nNode = ((Integer)enum.nextElement()).intValue();

            // get the value
            dVal = graph.tau(nCurNode, nNode) * Math.pow(graph.etha(nCurNode, nNode), B);

            // check if it is the max
            if(dVal > dMaxVal)
            {
                dMaxVal  = dVal;
                nMaxNode = nNode;
            }
        }
    }
    else  // Exploration
    {
        double dSum = 0;
        int nNode = -1;
```

```
        // get the sum at denominator of eq. b)
        Enumeration enum = m_nodesToVisitTbl.elements();
        while(enum.hasMoreElements())
        {
            nNode = ((Integer)enum.nextElement()).intValue();

            // Update the sum
            dSum += graph.tau(nCurNode, nNode) * Math.pow(graph.etha(nCurNode, nNode), B);
        }

        // get the everage value of the numerator of eq. b)
        double dAverage = dSum / (double)m_nodesToVisitTbl.size();

        // search the node in agreement with eq. b)
        enum = m_nodesToVisitTbl.elements();
        while(enum.hasMoreElements() && nMaxNode < 0)
        {
            nNode = ((Integer)enum.nextElement()).intValue();

            // get the value of p as difined in eq. b)
            double p =
                (graph.tau(nCurNode, nNode) * Math.pow(graph.etha(nCurNode, nNode), B)) / dSum;

            // if the value of p is greater the the average value the node is good
            if((graph.tau(nCurNode, nNode) * Math.pow(graph.etha(nCurNode, nNode), B)) > dAverage)
            {
                nMaxNode = nNode;
            }
        }

        //if there isn't any node that satisfy eq. b) select the last one
        if(nMaxNode == -1)
            nMaxNode = nNode;
    }

    // delete the selected node from the list of node to visit
    m_nodesToVisitTbl.remove(new Integer(nMaxNode));

    return nMaxNode;
}
```

Picture 12 Possible implementation in Java of the State Transition Rule

```
public void localUpdatingRule(int nCurNode, int nNextNode)
{
    // get the value of the Eq. c)
    double val =
        ((double)1 - R) * graph.tau(nCurNode, nNextNode) + (R * (graph.tau0()));

        // update tau
        graph.updateTau(nCurNode, nNextNode, val);
}
```

Picture 13 Possible implementation in Java of the Local Updating Rule

```
protected void globalUpdatingRule()
{
    double dEvaporation = 0;
    double dDeposition  = 0;

    for(int r = 0; r < m_graph.nodes(); r++)
    {
        for(int s = 0; s < m_graph.nodes(); s++)
        {
            if(r != s)
            {
                // get the value for deltatau
                double deltaTau =
                    ((double)1 / Ant4TSP.s_dBestPathValue) * (double)Ant4TSP.s_bestPath[r][s];

                // get the value for phermone evaporation
                dEvaporation = ((double)1 - A) * m_graph.tau(r,s);

                // get the value for phermone deposition
                dDeposition  = A * deltaTau;

                // see if s is in Z
```

```
                int i = 0;
                for (i = 0; i < m_nodesInZ.length && s != m_nodesInZ[i]; i++);

                // if s isn't in Z subtract some pheromone
                if(i == m_nodesInZ.length)
                    dEvaporation -= A * deltaTau;

                // update tau
                m_graph.updateTau(r, s, dEvaporation + dDeposition);
            }
        }
    }
}
```

Picture 14 Possible implementation in Java of the Global Updating Rule to solve the Steiner Problem