

# The Ant Colony System for the Freeze-Tag Problem

Dan George Bucatanschi  
Department of Computer Science  
Slayter Box 413  
Denison University  
Granville, OH 43023  
bucata\_d@denison.edu

## Abstract

In the Freeze-Tag problem there is a group of  $n$  sleeping robots lying in the Cartesian plane. One awake robot can wake up another sleeping robot by physically touching it, after which the newly awakened robot can assist the first robot in waking other robots. The goal is to minimize the overall time needed to waken all sleeping robots in the plane. We present an Ant Algorithm approach to solving the problem that improves upon previously known results.

## 1. Introduction

In the Freeze-Tag Problem a number of  $n$  robots are sleeping, or in stand-by mode, with the exception of one robot which is awake. The awake robot can awaken any sleeping robot by touching it, after which both robots can help each other awaken other robots. The purpose of the problem is to minimize the *makespan*, or the total time needed to awaken all the robots. Throughout this paper we consider the makespan to be the performance measure of algorithms that create awakening schedules. Thus, an algorithm with a smaller makespan has a higher performance.

The name of the problem comes from the children's game of Freeze-Tag, where someone is "it" and needs to touch another person who is not "it", after which both people are "it". Similarly, in our problem, the awake robot needs to touch another robot in order to become awake.

The problem can be considered as a parallel version of the Traveling Salesman Problem in which the salesman recruits other salesmen in every city he visits, salesmen who in turn help him visit other cities and recruit more salesmen. Thus, there is a cascading effect produced by the ever increasing number of salesmen recruited.

The goal of the computer is to give an awakening schedule of all the robots, i.e. the robots that each robot awakes and at what particular time. For simplicity, we will consider that all robots travel with one unit length per one unit time. The awakening schedule can be represented as a binary tree with the root having only one child and the edges being weighted with the distances that the robots have to travel. In Figure 1 below we give an example of a solution to a problem (left) and the awakening schedule associated with it (right). The makespan for this solution is the sum of the weights of the path ABDF, which is 4.5.

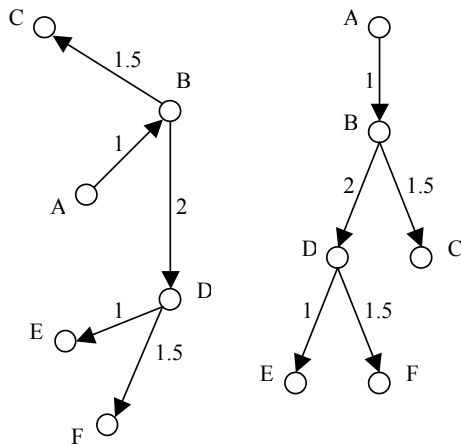


Figure 1 – A solution to a problem (left) and its associated awakening schedule (right).

### 1.1 Related Work

Arkin first introduced the problem in 2002 [1]. They show that the problem is NP-hard, they give a natural greedy strategy (called Greedy Fixed in this paper) to solve the problem, and prove that it has a worst-case performance ratio within  $7/3$  of the optimal solution on star graphs. They also show that there exists a Polynomial-Time Approximation Scheme (PTAS), running in nearly linear time, for geometrical instances of the problem.

Sztainberg introduces several new heuristic algorithms [6]. They present an improved greedy strategy (called Greedy Dynamic in this paper), Bang-for-the-Buck, which tries to make up for the limitations of the greedy strategies, Random Sector Selection, which is, as the name suggests, a randomized version of the greedy algorithm based on a number of sectors in which the plane is divided for each robot, and Opposite Cone, which sends two robots in almost opposite directions to awaken other robots.

For all problem instances given to these algorithms, Greedy Dynamic performs the best, followed in order by Bang-for-the-

Buck, Random Sector Selection and Opposite Cone. Our comparisons are exclusively against Greedy Dynamic, because it outperforms all the other algorithms by a significant margin.

### 1.2 Our Contribution

We present an adaptation of the Ant Colony System algorithm by Dorigo to the Freeze-Tag Problem [2]. We find that, in terms of performance, the Ant Colony System is better than Greedy Dynamic on most problem instances, while in terms of running time, Greedy Dynamic is faster than Ant Colony System.

### 1.3 Preliminaries

The version of the problem that we are concerned with considers all robots to lie in the Cartesian plane, thus the robots are able to travel to any other robot. Moreover, we note that all algorithms mentioned throughout the paper are online algorithms in the sense that each robot decides where to go only after it has been awakened by another robot. However, the algorithms simulate the robot behavior in the computer so that they can create better solutions than an algorithm running directly in real time on the robots themselves. At the end of their execution, they return a solution to a particular instance of the problem, which might be used later to awaken a group of real robots.

Notice that we can represent the problem as the complete graph with vertices in the initial positions of the robots. This way all robots can go to any other robot by following the graph. The graph is weighted with the distances between all pairs of robots.

We use problem instances that were originally developed for the Traveling Salesman problem from the library of

Traveling Salesman Problems, TSPLIB [5]. We also create three other problem instances developed specifically for the Freeze-Tag Problem in order to study the behavior of different algorithms for instances with some special properties.

We make the following observation. An optimal schedule will never have a path from the root's child (A) to any leaf (F, E, or C) that contains more than one node with one child or less (B) (see Figure 2 below). The reason is that you can always send two robots from node B to awaken the robots at both node D and E, and still perform at least as well as, or even better than before. The reason can also be checked geometrically with the triangle inequality. This means that only leaves can be single children in an optimal solution; all the other nodes must have a sibling. Thus, good solutions will exhibit this property.

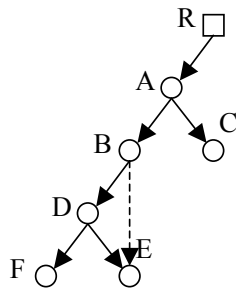


Figure 2 – By choosing edge BE, robot E is awakened at least as fast as going through BDE. (the root R is denoted by a square)

## 2. Previous Algorithms

### 2.1 Greedy Fixed

In the Greedy Fixed algorithm the next idle robot (robot that has just been awakened) chooses the next closest sleeping robot that has not been claimed by any other robot, and it commits to its choice [1].

In Figure 3 we can see the disadvantage of Greedy Fixed. The problem is that once robots E or F are scheduled to be awakened by the robots at C, they cannot be rescheduled to be awakened by the robots at D, which are closer than the current position of the robots from C.

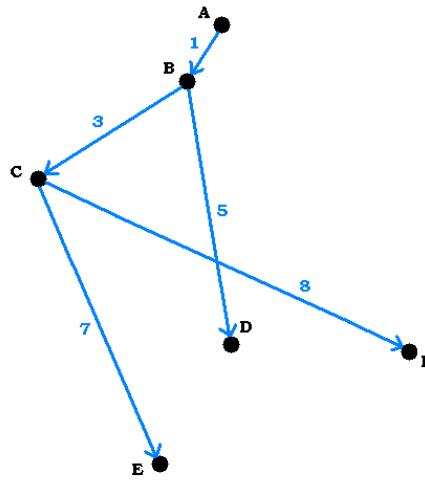


Figure 3 – Greedy Fixed Solution. The makespan is 12 and is given by the path ABCF.

### 2.2 Greedy Dynamic

Greedy Dynamic improves over the previous greedy algorithm [6]. Figure 4 presents the problem in Figure 3 solved by Greedy Dynamic. The robots still choose the closest unscheduled sleeping robot, but their commitment may dynamically change later. In this situation, the robots at point D can arrive at E and F before the robots at C, so the awakening schedule is changed to reflect this. The algorithm exhibits higher computational time, but much better performance over Greedy Fixed.

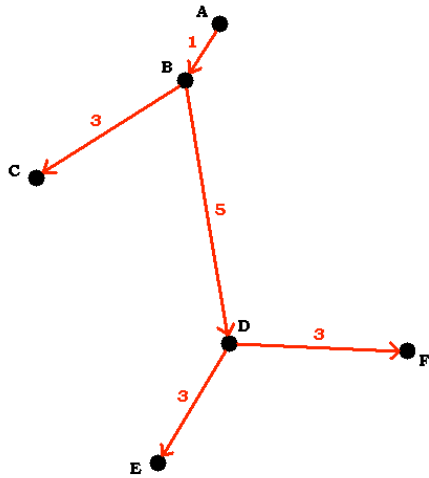


Figure 4 - Greedy Dynamic Solution. Notice the reduced makespan (9), which is determined by either the path ABDF or ABDE.

### 3. Ant Colony System

Ant Algorithms derive their name from the way ants search for food. Real-life ants use pheromone trails to guide themselves back and forth from their food source to their home. Ants are attracted to pheromone trails and use them to find the shortest path to a food source. Ant Algorithms simulate ants through a number of agents that have a limited ability to create a solution to a problem and leave pheromone trails on a map so that they can guide themselves to search for other solutions in the vicinity of the previous solutions.

We will now present the Ant Colony System with its main ideas. For the pseudo-code of the algorithm please refer to Appendix A.

For the Freeze-Tag Problem, each of the  $m$  ants has to keep track of its own awakening schedule (or solution to the problem) and what nodes it has already processed. Moreover, each ant will leave pheromone trails in two places: on the edges

of the graph to reflect the chosen awakening schedule and on the nodes of the graph to reflect whether both or only one robot left the node to awaken other robots.

Each ant will choose the next robot to be awakened at node  $j$  from node  $i$ , at iteration  $t$  with some probability depending on the following:

1. Whether or not the node  $j$  has already been visited (awakened). For each ant  $k$  we keep track of the set  $J_i^k$  of nodes that still need to be processed. This way an ant can avoid processing the same node more than once.
2. The inverse of the distance  $d_{ij}$  from node  $i$  to node  $j$ , called visibility and denoted by  $\eta_{ij} = 1/d_{ij}$ . Visibility can be used to direct the ants search for next robots to be awakened only in the neighborhood of the current node, thus getting better, greedy-like results from the very beginning, instead of completely random results. However, the sole use of visibility to direct ants' search would not improve over the greedy method by too much.
3. The amount of virtual pheromone  $\tau_{ij}(t)$  on the edge that connects node  $i$  to node  $j$ . The *pheromone trail* is updated dynamically, as the algorithm executes, and it represents the *learned desirability* of choosing node  $j$  when processing node  $i$ .
4. The amount of virtual pheromone  $\tau_{i^r}(t)$  found at node  $i$ . This *pheromone trail* represents the learned desirability to choose to send  $r = 2$  robots from node  $i$  to awaken other robots or to send only  $r = 1$  robot. Similarly to the previous pheromone, this trail is updated

dynamically as the algorithm executes.

### 3.1 Transition rule

The transition rule determines how an ant  $k$  processing node  $i$  will choose to send a robot at node  $j$ :

$$j = \begin{cases} \arg \max_{u \in J_i^k} \{ \tau 1_{iu}(t) \cdot [\eta_{iu}]^\beta \} & \text{if } q \leq q_0; \\ J & \text{if } q > q_0, \end{cases}$$

where  $q$  is a random variable uniformly distributed over  $[0, 1]$ ,  $q_0$  is a tunable parameter ( $0 \leq q_0 \leq 1$ ),  $\beta$  is a parameter that allows tweaking of how much the ants consider visibility over pheromone, and  $J$  in  $J_i^k$  is a node that is randomly selected according to the probability:

$$P_{ij}^k(t) = \frac{[\tau 1_{ij}(t)] \cdot [\eta_{ij}]^\beta}{\sum_{l \in J_i^k} [\tau 1_{il}(t)] \cdot [\eta_{il}]^\beta}.$$

Notice that when  $q_0$  is set closer to 1, the ants will exploit the known information about the nodes and the pheromone trails, whereas, when  $q_0$  is set closer to 0, the ants will explore more in the neighborhood of node  $i$ . This parameter is important in choosing the default behavior of the ants. While constant exploration is not desirable, because the ants might need too much time to find a good solution, constant exploitation is not desirable either, because the ants might converge to a locally optimal solution too fast, and thus not give a good globally optimal solution.

Similarly, the way ants choose the number  $r$  of robots to send from node  $i$  is determined by the following:

$$r = \begin{cases} 1, & \text{if } q_r \leq \frac{\tau 2_{i1}(t)}{\tau 2_{i1}(t) + \tau 2_{i2}(t)}; \\ 2, & \text{if } q_r > \frac{\tau 2_{i1}(t)}{\tau 2_{i1}(t) + \tau 2_{i2}(t)}. \end{cases}$$

where  $q_r$  is a random variable uniformly distributed over  $[0, 1]$ .

Notice that for the root node the transition rules given above do not work. In this case, we are considering the ants to start from a common virtual point on the graph, which is not part of the original problem, that has equal distance to all of the vertices in the graph. Of course, such a point does not exist, but it eases the application of the transition rules to the root node. More specifically, having the same distance to the root node means that we do not care about the distance when computing the probabilities given, so we can just exclude it from the formulas and use just the  $\tau/l$  pheromone to guide the ants in making a choice as their root node.

Moreover, for the root node we have to choose only one child, so the  $\tau 2$  pheromone information does not matter. Thus, we do not have to look at this information when choosing a root node, nor do we have to update this pheromone for the root node.

### 3.2 Pheromone trail update rule

In ACS, only the best awakening schedule is reinforced with pheromone. The updating rule is:

$$\begin{aligned} \tau 1_{ij}(t) &\leftarrow (1 - \rho) \cdot \tau 1_{ij}(t) + \frac{\rho}{L^+} \\ \tau 2_{ir}(t) &\leftarrow (1 - \rho) \cdot \tau 2_{ir}(t) + \frac{\rho}{L^+} \end{aligned}$$

where  $(i, j)$ 's are the edges belonging to  $T^+$ , the best awakening schedule found so far,  $\rho$  is a parameter determining pheromone decay, and  $L^+$  is the makespan of  $T^+$ . This update is a global update done after each iteration of the algorithm. Local updates are also done after each ant takes a step in processing nodes.

### 3.3 Local updates of pheromone trail

While processing node  $i$ , ant  $k$  selects  $r$  node(s)  $j$  in  $J_i^k$  and updates the pheromone as follows:

$$\tau_{1_{ij}}(t) \leftarrow (1 - \rho) \cdot \tau_{1_{ij}}(t) + \rho \cdot \tau_0,$$

$$\tau_{2_{ir}}(t) \leftarrow (1 - \rho) \cdot \tau_{2_{ir}}(t) + \rho \cdot \tau_0,$$

where  $\tau_0$  is the initial amount of pheromone laid on all edges. For  $\tau_2$ , it has been found experimentally that laying an initial value of  $\tau_0$  for  $r = 1$  robot and  $2\tau_0$  for  $r = 2$  robots provides good results. As for the actual value of  $\tau_0$ , it has been found that setting  $\tau_0 = (n L_{gd})^{-1}$ , where  $n$  is the number of robots and  $L_{gd}$  is the makespan of the awakening schedule made by Greedy Dynamic, produces good results.

When an ant takes one step, the local update rule of pheromone actually lowers the amount of pheromone for the particular node processed and the edge(s) chosen by the ant. This ensures more exploration from the other ants, which would take the same path with a lower probability than before. This prevents ants from converging to a locally optimal solution very fast, thus making them explore more, and reinforcing only the best awakening schedule found so far.

### 3.4 Use of a candidate list

In order to make ACS run on very big problem instances without a big impact on performance, we use a data structure called a candidate list. A candidate list is a list of preferred nodes to be considered first from a particular node. In our implementation of a candidate list we simply choose the  $cl$  closest robots for each particular node. When processing nodes, each ant actually looks at the nodes in the candidate list first, and applies all the local updating rules and transition rules as if only the nodes in the

candidate list existed. If all the nodes in the candidate list have already been processed, then the ant looks at the nodes in the whole graph and applies the local updating rules and transition rules to those. We have found experimentally that setting  $cl$  to equal 15 provides good results for problems of up to 300 robots, while we need  $cl$  to equal 20 or more for problems of 500 or more robots. This way, the number of computational steps necessary for an ant to take a step in processing a node is most of the times constant and proportional to  $cl$ , thus making the algorithm run faster. Only towards the end of the construction of an awakening schedule an ant might consider all the nodes in the problem.

### 3.5 Local improvement algorithm

It has been found experimentally that using a local improvement algorithm after one full iteration to improve each of the ant's awakening schedule is very beneficial to the final solution given by ACS.

The algorithm that we are using tries to shorten the makespan of an already created awakening schedule by linking the last node to be awakened (the one that determines the makespan) to the best possible parent that has only one child [3]. The best here means the one parent that will create the highest difference in makespans.

## 4. Experimental Results

We have implemented the ACS algorithm in C++ on a Pentium 4 machine at 2.4GHz running Linux Red Hat 9. The problem set used from the TSPLIB is composed of eil51, eil76, kroA100, d198, lin318, att532, and rat783. The numbers associated with each instance represent the number of robots in that particular instance. Also, we had three more problems created

specifically for the Freeze-Tag Problem: twoCirc100, tenCirc300 and moon1000.

In implementing the ACS algorithm, we had several versions of it, but which did not prove to create significantly different solutions. This is why we actually include data only from one of these versions, the one that performed constantly well throughout all the tests run, which was also the original version that we came up with.

For all problem instances the parameters for ACS that we used were the following:

$\beta = 4$  (visibility exponent)

$\rho = 0.01$  (pheromone decay rate)

$q_0 = 0.1$  (bias set for exploration)

$m = 10$  (number of ants)

$cl = 20$  (candidate list size)

$numberOfIterations = 3000$

By running several tests, we have realized that setting the number of iterations to 3000 provides enough time for the ants to converge to a certain solution, without waiting for a solution to be generated for too long.

Being a stochastic algorithm, for each problem instance we took the best makespan that the algorithm produced over 20 runs and we compared it to the Greedy Dynamic solution. The Greedy Dynamic algorithm was implemented by Dr. Matt Kretchmar [4]. Here is a chart with the results.

As we can see from Figure 5, ACS provides better solutions than Greedy Dynamic for all problem instances except moon1000. Also there is a less dramatic improvement over Greedy Dynamic for problems such as d198, att532 and rat783. The reason as to why ACS performs less stellar on these problems might rely on the fact that these problem instances have a specific pattern in the way the robots are located. Generally it has been observed that

ACS improves upon Greedy Dynamic much more when the instance of the problem is composed of uniformly distributed robots in the Cartesian plane, but this is something to be researched further.

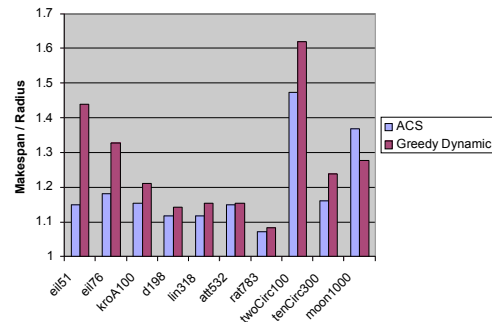


Figure 5 - Normalized Makespan of ACS and Greedy Dynamic

In terms of running time, ACS is much slower than Greedy Dynamic, but it is still a Polynomial time algorithm with a worst-case running time on the order of  $O(numberOfIterations \cdot m \cdot cl \cdot n^2)$ . If for all problem instances  $numberOfIterations$ ,  $m$ , and  $cl$  are held constant, the same way we did for our data collection, then we have an algorithm depending only on  $n$ , the number of robots, thus we get a  $O(n^2)$  algorithm.

In terms of memory requirements, ACS needs to store the table information with the pheromone maps, the candidate list for each node, and the list of cities to be visited by each ant, so its memory requirements are on the order of  $O(n^2 + cl \cdot n + m \cdot n)$ . If  $cl$  and  $m$  are kept constant again, the final memory requirements are on the order of  $O(n^2)$ .

## 5. Acknowledgements

We thank Matt Kretchmar for the selection of the research topic, guidance throughout the summer research, and implementation of several previously used algorithms. We thank Kevin Hutson for his guidance, and his idea and implementation

of the local improvement algorithm. We also thank Todd Feil for the mathematical support given throughout the summer. We acknowledge the Anderson Summer Research Fund for the given opportunity to research during the summer and the Summer Research Assistantship offered.

## 6. References

- [1] E. M. Arkin, M. A. Bender, S. P. Fekete, J. S. B. Mitchell, and M. Skutella. The Freeze-Tag Problem: How to wake up a swarm of robots. In *Proc. 13<sup>th</sup> ACM-SIAM Sympos. Discrete Algorithms*, pages 568-577, 2002.
- [2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.
- [3] K. Hutson. Through personal correspondence.
- [4] M. R. Kretchmar. Through personal correspondence.
- [5] G. Reinelt. Library of Travelling Salesman Problems. Updated November 20, 2002. Interdisciplinary Center for Scientific Computing of the Ruprecht-Karls-University of Heidelberg. Accessed June 2004. [<http://www.iwr.uni-heidelberg.de/group/s/comopt/software/TSPLIB95/index.html>].
- [6] M. Sztainberg, E. M. Arkin, M. A. Bender, and J. S. B. Mitchell. Analysis of heuristics for the Freeze-Tag Problem. In *Proc. Scandinavian Workshop on Algorithms*, Vol. 2368 of *Springer-Verlag LNCS*, pages 270-279, 2002.

## Appendix A: The ACS Algorithm

```
1. /* Initialization phase */
   For each pair (r,s)  $\tau_1(r,s) := \tau_0$  End-for
   For each node i let  $\tau_2(i,1) := \tau_0$  and  $\tau_2(i,2) := 2 * \tau_0$  End-for

2. /* This is the phase where each ant builds
   an awakening schedule. */
   For iteration:=1 to numberOfIterations do
     While there are still ants processing nodes do
       For k:=1 to m do
         Choose the number of robots to awake according to the
           transition rule applied on node i for ant k.
         Choose the node(s) j to awaken if possible
           from candidate list, if not, from the rest
           of the nodes according to the transition
           rule applied from node i to node j.
         Add the new nodes to the awakening schedule of ant k.
         Perform local update of pheromone trails for ant k.
       End-for
     End-while

3. /* In this phase global updating occurs */
   For k:=1 to m do
     Compute  $L_k$  /*  $L_k$  is the makespan of awakening schedule
                   of ant k. */
   End-for
   Compute  $L_{best}$ 
   Perform global update of pheromone of edges and vertices
     belonging to  $L_{best}$ 
   End-for

4. /* Print awakening schedule found */
   Print awakening schedule  $L_{best}$ 
```

NOTE: All variables mentioned are the same ones as defined above in section 3.