Discussion on the implementation of the Ant Colony System for the Traveling Salesman Problem.

Georgios Sarigiannidis

Delft University of Technology
Electrical Engineering, Mathematics and Computer Science
Department of Information Systems and Algorithms
(The Optimization Group)
ges@hiveworks.com

Abstract

The present paper explains the application of the Ant Colony System meta-heuristic to the Traveling Salesman Problem.

1. Introduction

In this paper, we consider a special case of the Traveling Salesman Problem (TSP):

Multidimensional Euclidean TSP.
Instance: A finite set $V \subseteq \mathbb{R}^m, |V| = n \geq 3$
Task: Find a Hamiltonian circuit T in the complete graph on V such that the total length $\sum_{(u,w)\in E(T)} \|u-w\|_2$ is minimum.

Here, $\|u-w\|_2$ denotes the Euclidean distance between u and w. It is then clear that [1] every optimum tour can be regarded as a polygon (it cannot cross itself). Since the Multidimensional Euclidean TSP is a special case of the metric TSP, it is also NP-hard. There is though a method proposed by Karp [1] (for the two dimensional case) that leads to $(1+e)$-approximation, that could easily be generalized for the multidimensional case. This method supposes that we have a set of n points in the unit square. We can then partition it by a regular grid such that each region contains few points, find an optimum tour within each region and then patch the subtours together.

Because of its nature (easily accessible), the TSP has been extensively studied and has been used for the study of many approaches to combinatorial optimization (approximation algorithms, local search, tabu search, neural networks, randomized algorithms, etc). See [2] and [3] for an extensive cover of these different methods.

This paper presents a new kind of nature-inspired metaheuristic, first described by M. Dorigo in 1996 [3], known as the 'Ant Colony System'. This metaheuristic is an analogy with the way ant colonies function and has been proven to be a very efficient new approach to stochastic combinatorial optimization and especially its application to the TSP. In what follows we present how ant colonies have inspired this method and discuss some of its main characteristics.

2. The Ant Colony System

It has been observed that ants, while almost blind, manage to establish shortest route paths from their colony to feeding sources and back [3], only by communicating information by laying on the ground a substance called pheromone . Since this paper is not concerned with the behavior of real ants, the reader is referred to [6] and [9] for a more complete discussion on real ant colonies and their emergence behavior.

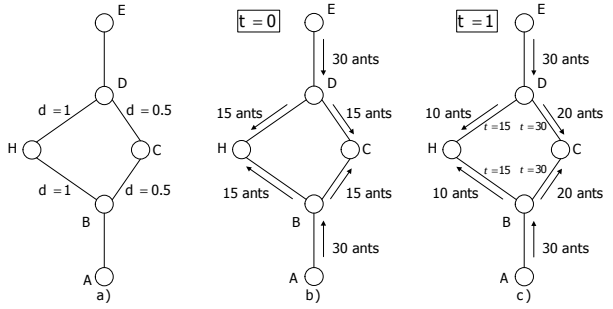

Figure 1

This behavior has inspired what is known as Ant Colony System (ACS) algorithms that use colonies of artificial ants. To illustrate the behavior of the ACS we use the following example [3]. Let as suppose that we have a graph as in Figure 1a) and an ACS as follows:

i. 30 new ants come from A to B and from E to D, at each time unit.
ii. Each ant walks at a speed of 1 per time unit and while walking, it lays down a pheromone trail of intensity 1.

At $t = 0$, there is no pheromone trail and the ants that are in B and D will choose their direction completely at random. Let us suppose then that 15 ants from each node will chose to go toward H and 15 toward C, as in Figure 1b).

At $t = 1$, as in Figure 1c), 30 new ants come to B and D. The ants that have chosen the direction of node C from B and D, have now reached D and B respectively. This means that on the path BCD, a pheromone trail of intensity 30 has been laid. On the other path BHD, the 15 ants that were at B and the 15 ants that were at D have only succeeded to reach H. This means that, on this path, the pheromone trail has intensity 15. The 30 new ants that are now on B will chose their direction based on the intensity of the pheromone trails, in such a way that the expected number of ants that will go to C will be the double of

those that will follow the direction of H. The same is also true for the 30 new ants that are now on D.

The idea of the ACS is that, if this process continues for a long time, almost all of the new coming ants will follow the shortest path. This idea can then be applied with impressive results to the Traveling Salesman Problem [4] and to other combinatorial optimization problems (Quadratic assignment, Job-shop scheduling, Vehicle routing, Graph coloring etc) [5]. For its application to the TSP, an intuitive explanation of its behavior, is given by M. Dorigo in [4]: *"Once all the ants have generated a tour, the best ant deposits (at the end of iteration t) its pheromone, defining in this way a 'preferred tour' for search in the following algorithm iteration t+1. In fact, during iteration t+1 ants will see edges belonging to the best tour as highly desirable and will choose them with high probability. Still, guided exploration together with the fact that local updating 'eats' pheromone away (i.e., it diminishes the amount of pheromone on visited edges, making them less desirable for future ants) allowing for the search of new, possibly better tours in the neighborhood of the previous best tour. So ACS can be seen as a sort of guided parallel stochastic search in the neighborhood of the best tour."*

In the following paragraph we explain more thoroughly how the ACS is applied to the TSP.

3. The ACS algorithm

Given is a set of n nodes in an $m-$dimensional Euclidean space. We denote by d_{ij} the length of the direct path between nodes i and j. In the present case, if the position of the node v_i is defined by $v_i := \{x_1^i, x_2^i, \ldots, x_m^i\}$, d_{ij} equals the Euclidean distance between nodes i and j:

$$d_{ij} := \sqrt{\sum_{k=1}^{m} \left(x_k^i - x_k^j\right)^2} .$$

The ACS algorithm can now globally be described as follows [4]:

```
ACS_TSP Algorithm
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 is reached.
```

Let k the number of ants that are used by ACS_TSP and that all edges have a pheromone intensity of value t_0. At the beginning of the first iteration, each ant is positioned randomly at a starting node. Each ant moves then to a still not visited node according to the transition probability defined by

$$p_{sj}(t) := \frac{\left[t_{sj}(t)\right] \cdot \left[h(s,j)\right]^b}{\sum_{j \in J(s)} \left[t_{sj}(t)\right] \cdot \left[h(s,j)\right]^b}$$

where, $t_{sj}(t)$ is the intensity of the pheromone trail of the edge (s,j) at time t, $h(s,j) := 1/d_{sj}$ is the inverse of the distance between nodes s and j, and $b > 0$ is a parameter which determines the relative importance of pheromone versus distance. This rule applies of course differently to each ant, supposes that the given ant is at node s, and that the subset of the nodes still not visited by it is given by $J(s)$. In this way the ACS favors the choice of edges which are shorter and which have a greater amount of pheromone.

After each ant has chosen the next node of its path, a local pheromone updating rule is applied. If an ant was at node r and has chosen, by applying the state transition rule described above, to go to node s, then the pheromone level of the (r,s) edge is updated by

$$t(r,s) := (1-r)t(r,s) + rt_0 ,$$

where r and t_0 are parameters.

After all ants have completed their tours, they remain to their current position for the following iteration and a global pheromone updating rule is applied for each edge (r,s) in the graph:

$$t(r,s) := (1-a) \cdot t(r,s) + a \cdot \Delta t(r,s), \text{ with}$$

$$\Delta t(r,s) := \begin{cases} (L_{gb})^{-1}, & \text{if } (r,s) \text{ belongs to the global best tour} \\ 0 & \text{otherwise} \end{cases}$$

$0 < a < 1$ is in this case the pheromone decay parameter, and L_{gb} is the length of the globally best tour found up to that point.

The ACS proceeds then in this way until the end conditions are reached. Usually, the end conditions are the following:

i. A (user defined) maximum number of iterations is performed, or

ii. A tour of a given length (target length) is found.

For our implementation we have chosen to only apply the first end condition, and the target length is set to zero.

As you have noticed, this algorithm makes use of a number of parameters. Dorigo and Gambardella have experimented extensively trying to identify their optimal values, and have reported in [4] the following:

- Experimental observation shows that ACS works well when the number of ants used is equal to 10.
- The optimal values of the parameters are largely independent of the problem and experimental observation leads to the following values: $b = 2$, $a = r = 0.1$ and $t_0 = (n \cdot L_{NN})^{-1}$, where L_{NN} is the tour length produced by applying the Nearest Neighbor heuristic and n the total number of nodes. Note here that t_0 is also used as the initial pheromone intensity value for all edges in the graph.

4. Experimental Results

For the purposes of the course 'Optimization in Logistics', the ACS algorithm was implemented using the programming language C++ (see Appendix). The main functions used by this implementation have as follows:

GetDistanceMatrix reads from an input file (**TSP.in**) an integer N indicating the number of nodes in the input graph, an integer M indicating the dimension of the Euclidean space of the given TSP instance, and a matrix in $\mathbb{R}^{N\times M}$ containing the exact positions of the N nodes in $\mathbb{R}^{M}$.

NearestNeighborSolve constructs a solution of the given TSP instance, using the Nearest Neighbor heuristic (starting by a randomly chosen node) and returns the solution (TSP tour) and its length.

ACSTSP solves the given TSP instance, using the ACS algorithm. It requires as input the number of iterations that the algorithm must perform and returns the solution (TSP tour) and its length.

The output of the program consists of two files:

a) A file called '**TSP.out**' containing the length of the found solution, the CPU time that was needed for its construction (depending on the chosen number of iterations that the algorithm must perform) and the solution itself

b) A file called '**res.out**' containing information regarding the performance of the algorithm in a matrix form: it contains a number of rows equal to the number of iterations that the algorithm has performed, with each row containing the iteration index, the length of the best solution found during the iteration, the length of the best solution found up to this iteration and the index of the ant that had the best performance during the iteration

In the present paragraph, the performance of the algorithm is illustrated using as a test case the berlin52 TSP instance (taken from TSPLIB [7]), that consists of 52 nodes in $\mathbb{R}^2$. The optimal tour for this instance has length 7452 and has as in Figure 2.

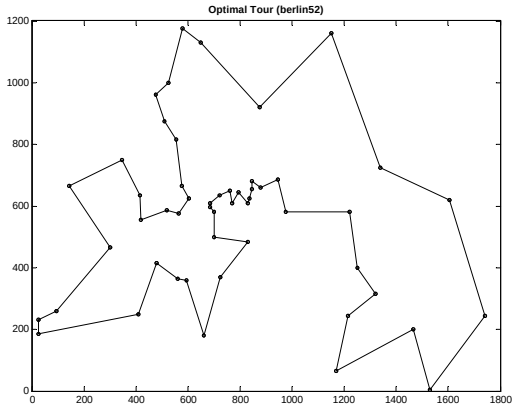

Figure 2

The ACS algorithm succeeded in finding a solution with length 7861.52, after 37761 iterations (note that other runs might perform better or worse). The ACS solution is presented in Figure 3.

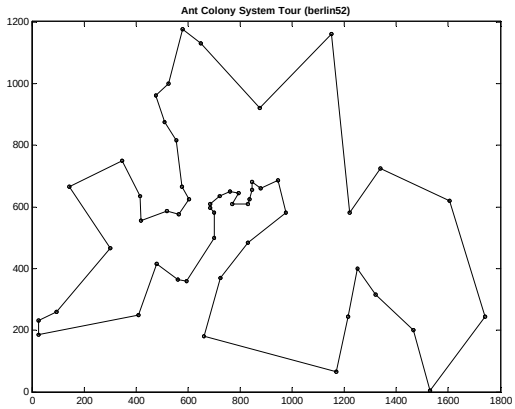

Figure 3

The following figures illustrate the performance of the ACS algorithm for the given instance (berlin52) during a random run. Figure 4 presents the best solution that the ACS has found up to each iteration, while Figure 5 presents the moving average of the best solutions found per iteration up to each iteration.

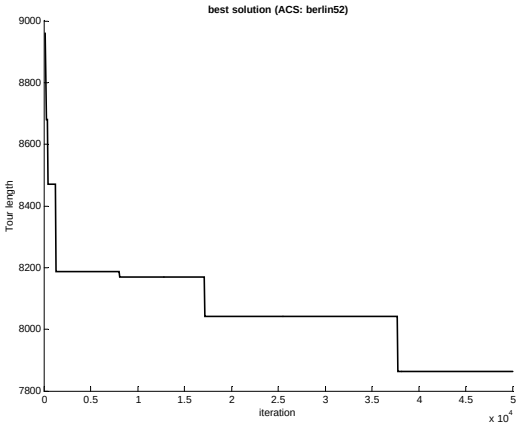

Figure 4

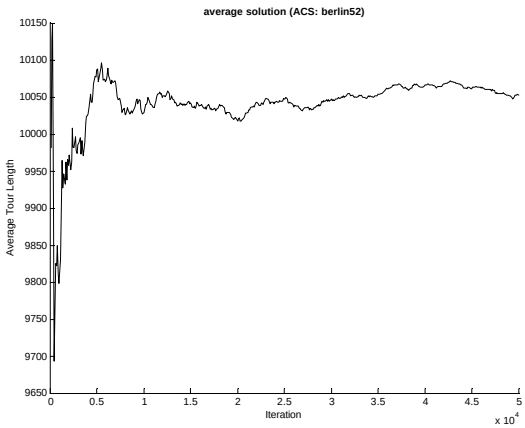

Figure 5

For a more extensive survey of the performance of the ACS, the reader is referred to [3] and [8].

5. Final Remarks

The Ant Colony System has become, because of its success, a very popular meta-heuristic for the solution of combinatorial problems and especially for the Traveling Salesman Problem; M. Dorigo, that introduced this meta-heuristic, has won the 'Italian Prize for Artificial Intelligence' and the 'Marie Curie Excellence Award' for this invention.
More modern applications of the ACS combine the algorithm described in the present paper with various local search approaches as the 2-Opt and 3-Opt (see [4]) or other evolutionary techniques (see [10]). It has been experimentally shown that use of such approaches enhances dramatically its performance [4].

6. References

[1] B. Korte and J. Vygen. "Combinatorial Optimization. Theory and Algorithms." Germany: Springer, 2002.
[2] H.N. Post. "Transport, Routing- en Schedulingproblemen" Delft: Delft University of Technology, 2004.
[3] M. Dorigo, V. Maniezzo and A. Colorni. " The Ant System: Optimization by a colony of cooperating agents" IEEE Transactions on Systems, Man, and Cybernetics – Part B, Vol. 26, No. 1 1996: 1-13.
[4] M. Dorigo, L.M. Gambardella. "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem" IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1 1997.
[5] M. Dorigo, L.M. Gambardella. "Ant Algorithms for Discrete Optimization" Artificial Life, Vol. 5, No. 3 1999: 137-172.
[6] E. Bonabeau, M. Dorigo and G. Theraulaz. "Swarm Intelligence: From Natural to Artificial Systems (Santa Fe Institute Studies in the Sciences of Complexity Proceedings)" Oxford University Press, 1999.
[7] TSPLIB: http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html
[8] M. Dorigo and L.M. Gambardella. "Ant colonies for the traveling salesman problem" Biosystems, 43 1997: 73-81.
[9] M. Dorigo and T. Sttzle. "Ant Colony Optimization (Bradford Books)" The MIT Press, 2004.
[10] H.M. Botee and E. Bonabeau. "Evolving Ant Colony Optimization" Advances in Complex Systems Vol. 1 1998: 149-159 (http://www.tbi.univie.ac.at/~studla/ACS/Contents.html)

7. APPENDIX: C++ Implementation of the ACS algorithm

```
#include <stdio.h>
#include <iostream>
#include <limits>
#include <list>
#include <string>
#include <fstream>
#include <time.h>
#include <stdlib.h>
#include <cstdlib>
#include <cmath>

using namespace std;

typedef list<int> Tour;
const double INF = numeric_limits<double>::infinity( );

template <class _vt> void ReleaseMatrix(const int nodeCount, _vt** ppmatrix)
{
	for(int n = 0; n < nodeCount; n++)
	{
		delete [] ppmatrix[n];
	}
	delete [] ppmatrix;
}

double** GetDistanceMatrix(char* szFilename, int& pNodeCount)
{
	FILE* pFile = fopen( szFilename, "r" );
	if (pFile==NULL) exit (1);
	fseek(pFile, 0L, SEEK_SET );

	float fp;
	int dim;
	int nodes;
	fscanf(pFile, "%d", &nodes );
	fscanf(pFile, "%d", &dim );
	double** coordinates = new double*[nodes];
	for (int n = 0; n < nodes; n++)
	{
		coordinates[n] = new double[dim];
		for(int m = 0; m < dim; m++)
		{
			fscanf(pFile, "%f", &fp );
			coordinates[n][m] = fp;
		}
	}

	fclose (pFile);
	pNodeCount = nodes;
	double** ppmatrix = new double*[nodes];
	//initialization
	for(int i = 0; i < nodes; i++)
	{
		ppmatrix[i] = new double[nodes];
		for(int j = 0; j < nodes; j++)
		{
			double distance = 0;
			for(int k = 0; k < dim; k++)
			{
				double cur_c = (coordinates[i][k]-coordinates[j][k]);
				distance += cur_c * cur_c;
			}
			ppmatrix[i][j] = sqrt(distance);
		}
	}

	ReleaseMatrix(nodes, coordinates);

	return ppmatrix;
}

Tour setdiff(Tour set1, const Tour& set2)
{
	Tour::const_iterator cit;
	for (cit = set2.begin(); cit != set2.end(); cit++)
		set1.remove(*cit);
```

```
        return set1;
}

Tour NearestNeighborSolve (double** d, const int n, double& TourLength)
{
        Tour::iterator cit;
        Tour cur_Tour;
        Tour V;
        int next_node;
        double cur_L = 0;

        //set diagonal to infinity
        for (int i = 0; i < n; i++)
                d[i][i] = INF;

        //select begin node
        srand( (unsigned)time( NULL ) );
        int m = rand() * n / RAND_MAX;

        for (int i = 0; i < n; i++)
                V.push_back(i);
        cur_Tour.push_back(m);

        Tour to_visit = setdiff(V, cur_Tour);
        while (!to_visit.empty())
        {
                const int cur_end = cur_Tour.back();

                cit = to_visit.begin();
                cit++;
                next_node = to_visit.front();
                for (cit; cit != to_visit.end(); cit++)
                        if (d[cur_end][*cit] < d[cur_end][next_node])
                                next_node = *cit;
                cur_L = cur_L + d[cur_end][next_node];
                cur_Tour.push_back(next_node);
                to_visit.remove(next_node);
        }
        cur_L = cur_L + d[cur_Tour.back()][cur_Tour.front()];
        cur_Tour.push_back(cur_Tour.front());
        TourLength = cur_L;
        return cur_Tour;
}

Tour ACSTSP(double** distances_matrix, const int NrOfNodes, int MaxIterations, double target_length)
{
        int m = 10; //number of ants
        double alpha = 0.1;
        double beta = 2;
        double rho = 0.1;
        int n = NrOfNodes;
        int t_max = MaxIterations;
        double L_target = target_length;
        double** d = distances_matrix;
        /* ============================
                initialization
        ============================ */

        double L_best = INF;
        double L_nn = 0;
        Tour T_best = NearestNeighborSolve (d, n, L_nn);
        L_best = L_nn;
        //initial pheromone trails
        double c = 1 / (n * L_nn);
        double** tau = new double*[int(n)];
        for(int i = 0; i < n; i++)
        {
                tau[i] = new double[n];
                for(int j = 0; j < n; j++)
                {
                        tau[i][j] = c;
                }
        }
        //place m ants in n nodes
        int** ant_tours = new int*[int(m)];
        for(int i = 0; i < m; i++)
        {
                ant_tours[i] = new int[n+1];
                for(int j = 0; j < n; j++)
                {
                        ant_tours[i][j] = 0;
```

```
        }
}
for(int i = 0; i < m; i++)
{
        double r = (double)rand() / ((double)RAND_MAX + (double)1);
        ant_tours[i][0] = (int)(r * n);
}

/* ============================
        ITERATIONS - LOOP
============================ */

int t = 1;
ofstream out_file("res.out");
int* visited = new int[n];
double* p = new double[n];
int* to_visit = new int[n];

srand((unsigned)time(NULL));

while ((t <= t_max) & (L_target < L_best))
{
/* ============================
        CREATE TOURS
============================ */
        for (int s = 1; s < n; s++)
        {
                int c_tv = n-s;
                for (int k = 0; k < m; k++)
                {
                        int current_node = ant_tours[k][s-1];

                        for (int i = 0; i < s; i++)
                                visited[i] = ant_tours[k][i];
                        int to_visit_index = 0;
                        for (int i = 0; i < n; i++)
                        {
                                bool found = false;
                                int j = 0;
                                while ((j < s) & !found)
                                {
                                        found = (i == visited[j]);
                                        j++;
                                }
                                if (!found)
                                {
                                        to_visit[to_visit_index] = i;
                                        to_visit_index++;
                                }
                        }
                        double sum_p = 0;
                        for (int i = 0; i < c_tv; i++)
                        {
                                p[i] = pow(tau[current_node][to_visit[i]],1) *
                                       pow(1/d[current_node][to_visit[i]],beta);
                                sum_p += p[i];
                        }
                        for (int i = 0; i < c_tv; i++)
                                p[i] /= sum_p;
                        for (int i = 1; i < c_tv; i++)
                                p[i] += p[i-1];

                        double r = (double)rand() / (RAND_MAX + 1);
                        int select = to_visit[c_tv-1];
                        i = 0;
                        while ( (i<c_tv) & (select == to_visit[c_tv-1]))
                        {
                                if (r <= p[i])
                                        select = to_visit[i];
                                i++;
                        }
                        int city_to_visit = select;
                        ant_tours[k][s] = city_to_visit;
                        tau[current_node][city_to_visit] = (1 - rho) *
                                                tau[current_node][city_to_visit] + rho * c;
                }
        }
/* ============================
        UPDATE
============================ */
        for (int k = 0; k < m; k++)
```

```
                        ant_tours[k][n] = ant_tours[k][0];
                double L_min = INF;
                int best_ant = 0;
                for (int k = 0; k < m; k++)
                {
                        double L_current = 0;
                        for (int i = 0; i < n; i++)
                        {
                                L_current += d[ant_tours[k][i]][ant_tours[k][i+1]];
                        }
                        if (L_current < L_min)
                        {
                                L_min = L_current;
                                best_ant = k;
                        }
                }
                for (int i = 0; i < n; i++)
                {
                        for (int j = 0; j < n; j++)
                        {
                                tau[i][j] *= 1 - alpha;
                        }
                }
                for (int i = 0; i < n+1; i++)
                {
                        tau[ant_tours[best_ant][i]][ant_tours[best_ant][i]] += alpha / L_min;
                }
                out_file << t << "\t" << L_min << "\t" << L_best << "\t" << best_ant << endl;
                t++;
                for (int k = 0; k < m; k++)
                        ant_tours[k][0] = ant_tours[k][n];
                if (L_min < L_best)
                {
                        L_best = L_min;
                        Tour T_min;
                        for (int i = 0; i < n+1; i++)
                                T_min.push_back(ant_tours[best_ant][i]);
                        T_best = T_min;
                }
        }
        delete [] visited;
        delete [] p;
        delete [] to_visit;

        // CLEAN UP
        ReleaseMatrix(n, tau);
        ReleaseMatrix(m, ant_tours);
        return T_best;
}

double path_length(Tour path, double** d)
{
        double L = 0;
        Tour::iterator cit;
        Tour::iterator dit;
        cit = path.begin();
        dit = path.begin();
        dit++;
        for (cit; dit != path.end(); cit++)
        {
                int b_node = *cit;
                int e_node = *dit;
                L = L + d[b_node][e_node];
                dit++;
        }
        return L;
}

int main( int argc, char *argv[], char *envp[] )
{
        clock_t start, finish;
        start = clock();

        int lNodeCount = 0;
        double NN_length = 0.0;
        double** ppmatrix = GetDistanceMatrix("TSP.in", lNodeCount);
        int max_iterations;
        cout << "Enter the number of iterations for the Ant Colony System: ";
        cin >> max_iterations;

        Tour ACS_sol = ACSTSP(ppmatrix, lNodeCount, max_iterations, 0.0);
```

```
        double ACS_length = path_length(ACS_sol, ppmatrix);

        Tour NN_sol = NearestNeighborSolve (ppmatrix, lNodeCount, NN_length);

        finish = clock();
        double duration = (double)(finish - start) / CLOCKS_PER_SEC;

        ofstream out_file("TSP.out");
        if (! out_file)
        { cerr << "oops! unable to open input file\n"; return -2; }
        out_file << "Nodes: " << lNodeCount << endl;
        out_file << "Time used (sec): " << duration << endl;
        out_file << "Nearest Neighbor Solution: " << NN_length << endl;
        out_file << "Ant Colony System Solution: " << ACS_length << endl;
        Tour::const_iterator cit;
        for (cit = ACS_sol.begin(); cit != ACS_sol.end(); cit++)
        {
               int i = *cit;
               out_file << i + 1 << endl;
        }
        out_file << "EOF" << endl;

        ReleaseMatrix(lNodeCount,ppmatrix);
        return 0;
}
```