UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Application of Tabu Search to the Travelling Salesman Problem (TSP)

Student:

Santos Ambrosio Martín Pérez

Supervisor:

Dr. Alaa Khamis

Subject:

Metaheuristic Optimization and its Applications in Robotics and Automation

Programme:

Master in Robotics and Automation

Academic Course: 2010-2011

INDEX:

1. Summary

The main objective of this project is conducting a study on how to solve the Traveling Salesman Problem (TSP) using Tabu Search (TS).

We are going to start talking about the travelling salesman problem and the tabu search. Later, we are going to explain how to solve the travelling salesman problem by applying tabu search.

After analyzing the problem, we will discuss the results obtained by implementing in Matlab a program that solves the problem by applying tabu search.

Finally, we will expose the personal conclusions.

2. Introduction

2.1 The Travelling Salesman Problem (TSP)

The Travelling Salesman Problem (TSP) is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science.

Given a list of n cities and the cost of travel between each pair of them, a travelling salesman must visit all of the cities only once and return home, making a loop (roundtrip). He would like to travel in the most efficient way, such as the cheapest way, the shortest distance, in the shortest period of time or another criterion.

Figure 2.1.1: Example of solution of the travelling salesman problema between 13 cities.

2.2 History

The origins of the travelling salesman problem are unclear. A handbook for travelling salesmen from 1832 mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment.

The travelling salesman problem was defined in the 1800s by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman. Hamilton's Icosian Game was a recreational puzzle based on finding a Hamiltonian cycle. The general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard, notably by Karl Menger, who defines the problem, considers the obvious brute-force algorithm, and observes the non-optimality of the nearest neighbour heuristic.

Hassler Whitney at Princeton University introduced the name travelling salesman problem soon after. In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the USA. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson at the RAND Corporation in Santa Monica, who expressed the problem as an integer linear program and developed the cutting plane method for its solution. With these new methods they solved an instance with 49 cities to optimality by constructing a tour and proving that no other tour could be shorter. In the following decades, the problem was studied by many researchers from mathematics, computer science, chemistry, physics, and other sciences.

Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete, which implies the NP-hardness of TSP. This supplied a mathematical explanation for the apparent computational difficulty of finding optimal tours.

Great progress was made in the late 1970s and 1980, when Grötschel, Padberg, Rinaldi and others managed to exactly solve instances with up to 2392 cities, using cutting planes and branch-and-bound.

In the 1990s, Applegate, Bixby, Chvátal, and Cook developed the program Concorde that has been used in many recent record solutions. Gerhard Reinelt published the TSPLIB in 1991, a collection of benchmark instances of varying difficulty, which has been used by many research groups for comparing results. In 2005, Cook and others computed an optimal tour through a 33,810-city instance given by a microchip layout problem, currently the largest solved TSPLIB instance. For many other instances with millions of cities, solutions can be found that are guaranteed to be within 1% of an optimal tour.

2.2 Description

TSP can be modeled as a weighted graph, such that cities are the graph's vertices. Depending on whether we consider the graph as an undirected or directed graph we will have two types of TSP:

1. Symmetric TSP: The distance between two cities is the same in each opposite direction, forming an undirected graph. In this case, for n = 6 cities we will have n!/2 = 6!/2 = 720/2 = 360 possible solutions.

2. Asymmetric TSP: Paths may not exist in both directions or the distances might be different, forming a directed graph. In this other case, for n = 6 cities we will have n! = 6! = 720 possible solutions.

In the travelling salesman problem the search space is very big. For example, for 21 cities there are 21! = 51090942171709440000 possible tours (in an asymmetric TSP). So talking about his computational complexity we can ay that is a NP-Hard problem.

The TSP is one of the most intensively studied problems in computational mathematics and yet no effective solution method is known for the general case.

2.3 TSP Formulation

A classical Traveling Salesman Problem (TSP) can be defined as a problem where starting from a node it is required to visit every other node only once in a way that the total distance covered is minimized. This can be mathematically stated as follows:

In the complete directed graph D = (N,A), with arc-cost c_{ij}, we seek the tour (a directed cycle that contains all n cities) of minimal length.

$$x_{ij} \begin{cases} 1 & \text{if arc(i,j) is in the tour} \\ 0 & \text{otherwise} \end{cases}$$

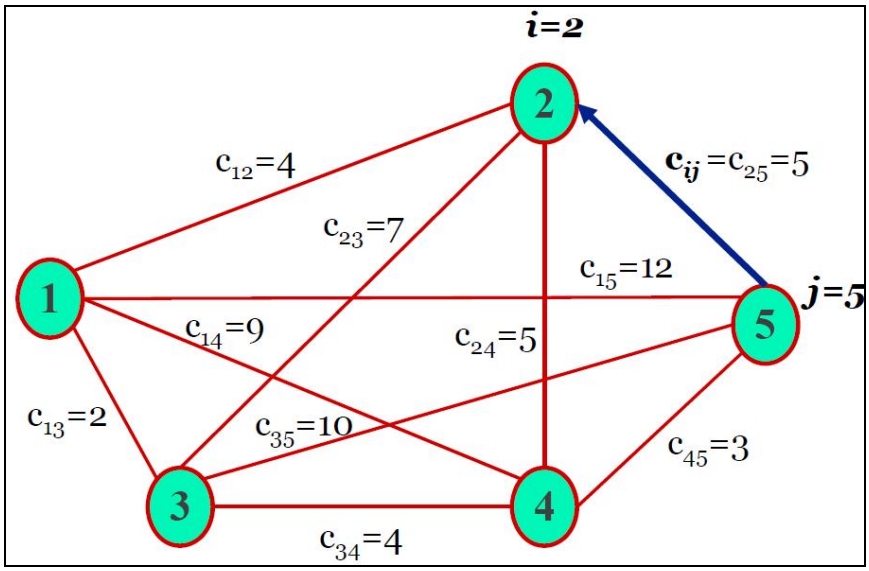


Figure 2.3.1: Example of directed graph.

The TSP can be defined as a minimization problem as follows:

$$\text{Min} \quad \sum_{i,j} c_{ij} x_{ij}$$

Subject to two types of constraints:

1. Assignment constraints:

$$\sum_i x_{ij} = 1 \quad \forall i$$
$$\sum_j x_{ij} = 1 \quad \forall j$$
$$0 \le x_{ij} \le 1 \quad x_{ij} \text{ integer}$$

2. Subtour constraints:

$$u_i = 1$$
$$2 \le u_i \le n \qquad \forall i \ne j$$
$$u_i - u_j + 1 \le (n-1)(1 - x_{ij}) \qquad \forall i \ne 1, \forall j \ne 1$$
$$x_{ij} \in \{0,1\}$$

2.4 TSP Applications

Much of the work on the TSP is motivated by its use as a platform for the study of general methods that can be applied to a wide range of discrete optimization problems.

Some of the TSP applications:

- Microchips manufacturing: It is very important to determine the best order in which a laser will drill a lot of holes in a circuit board.
- Transportation and logistics: There are hundreds of applications in these fields.
- Assignment of routes for planes of a specific fleet.
- Permutation flow shop scheduling problem (PFSP).

3. Proposed Solution

3.1 The Tabu Search (TS)

Tabu Search (TS) is a search stochastic optimization technique belonging to the trajectory methods.

It can be considered as the combination of local search and memory structures, and is originally proposed to allow local search to overcome local optima.

Tabu search is an iterative neighborhood search algorithm, where the neighborhood changes dynamically.

To prevent the process from cycling in a small set of solutions, some attributes of recently visited solutions are stored in a Tabu List, which prevents their occurrence for a limited period. By avoiding already visited points, loops in search trajectories are avoided and local optima can be escaped.

The main feature of TS is the use of an explicit memory. The memory is used in two ways in order to achieve two objectives:

- Prevent the search from revisiting previously visited solutions.
- Explore the unvisited areas of the solution space: diversification.

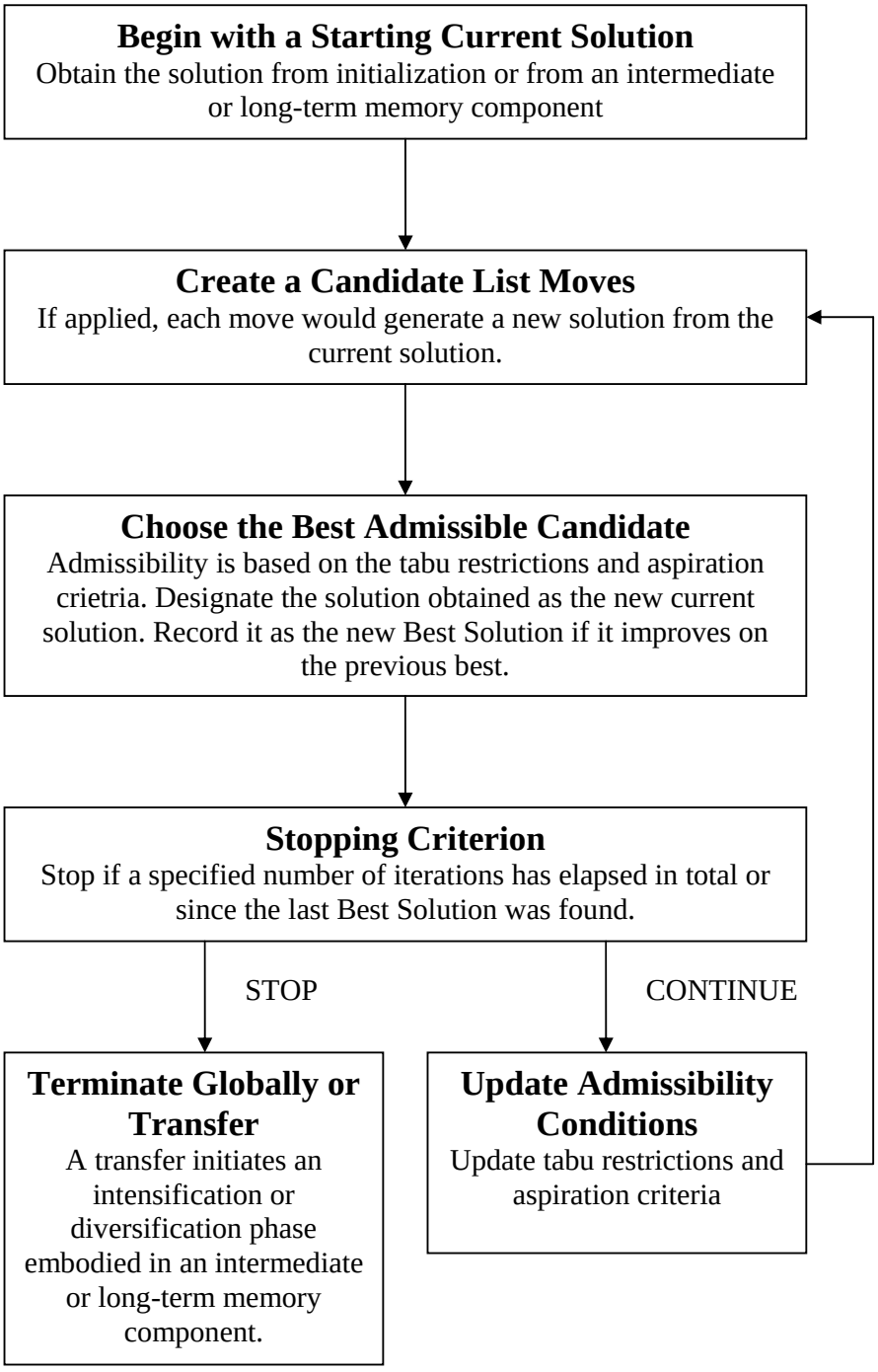
A simple TS usually implements two forms of memory:

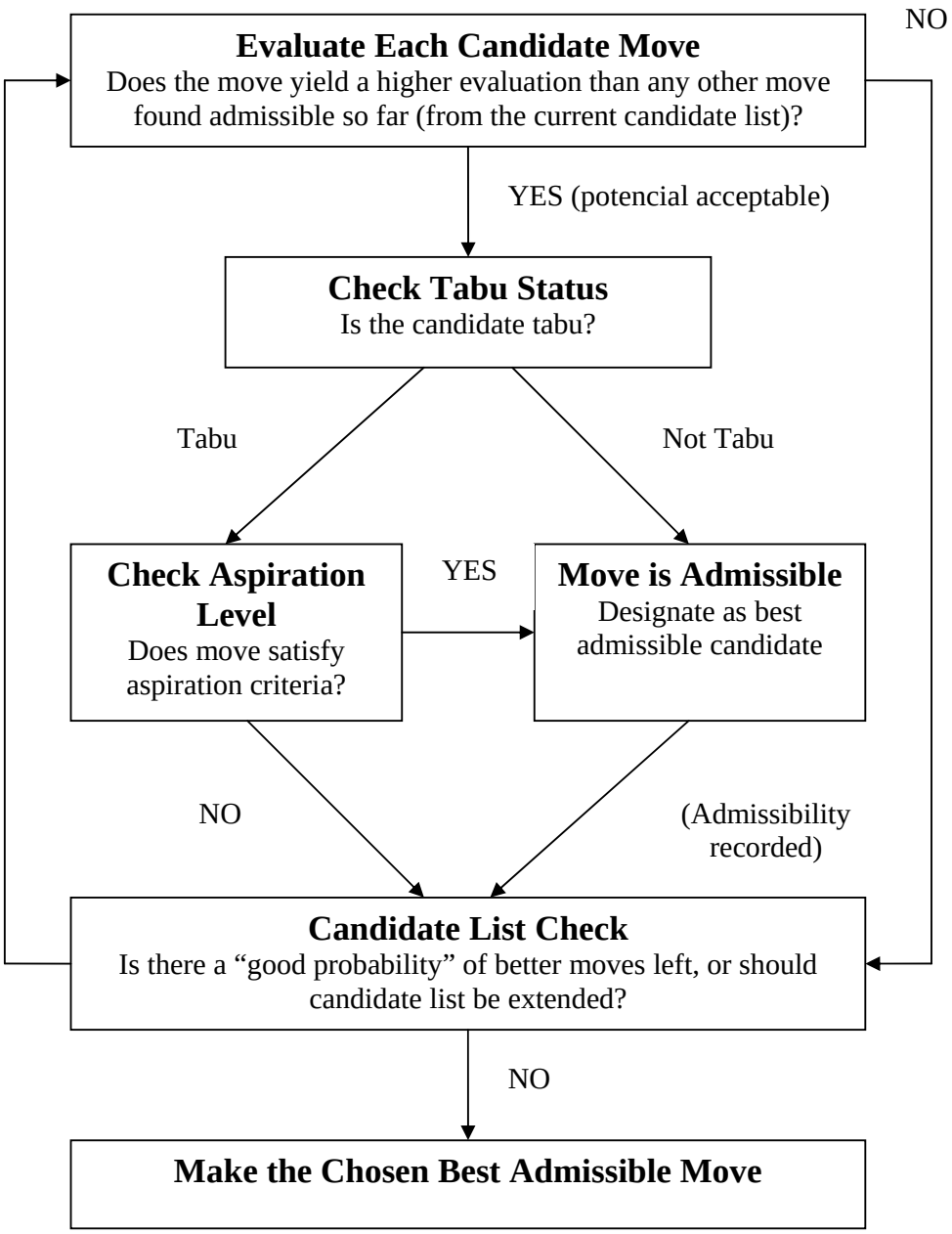
- A recency-based memory or short-term memory or tabu list, which maintains information about how recently a move has been made.
- A frequency based memory or long term memory, which maintains information about how often a move has been made during a specified time interval.

When a single attribute is marked as tabu, this one typically results being tabu in more than one solution.

Some of these solutions that must now be avoided could be of excellent quality and might not have been visited. This problem is known as stagnation. To prevent stagnation, sometimes it is useful allow a certain movement even if it is in the tabu list. Approaches used to cancel the tabu movements are referred to as Aspiration Criteria.

3.2 Tabu Search Algorithm




Evaluate Each Candidate Move
Does the move yield a higher evaluation than any other move found admissible so far (from the current candidate list)?
NO
YES (potencial acceptable)
Check Tabu Status
Is the candidate tabu?
Tabu
Not Tabu
Check Aspiration Level
Does move satisfy aspiration criteria?
YES
Move is Admissible
Designate as best admissible candidate
NO
(Admissibility recorded)
Candidate List Check
Is there a “good probability” of better moves left, or should candidate list be extended?
NO
Make the Chosen Best Admissible Move

3.3 Advantages and Disadvantages of TS

1. Advantages of Tabu Search:
   - Allows non-improving solution to be accepted in order to escape from a local optimum.
   - Can be applied to both discrete and continuous solution spaces.
   - For larger and more dificult problems (scheduling, quadratic assignment and vehicle routing), tabu search obtains solutions that often surpass the best solutions previously found by other approaches
2. Disadvantages of Tabu Search:
   - Too many parameters to be determined.
   - The number of iterations could be very large.
   - Global optimum may not be found, depends on parameter settings.

3.4 Example of application of Tabu Search to TSP

With this example we are going to explain how to solve the travelling salesman problem by applying tabu search.

3.4.1 Initial Parameters:

- Number of nodes n = 5 → n!/2 = 5!/2 = 120/2 = 60 possible solutions.
- Numer of iterations = 7.
- Tabu tenure (length) = 3.
- Aspiration criteria: To allow a move, even if it is tabu, if it results in a solution with an objective value better than the current best-known solution.
- Weigthed graph as follows:

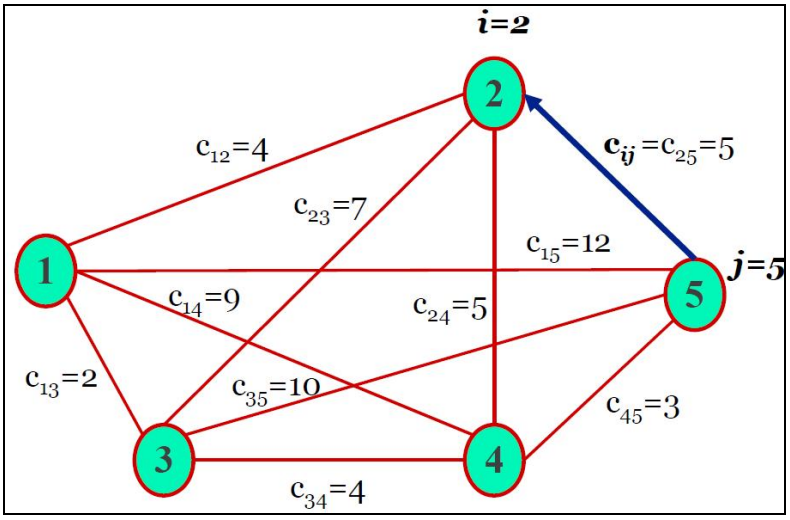


Figure 3.2.1: Weighted graph used.

3.4.2 Iterations:

1. Initial solution (iteration 0): We have got a random initial solution.

   Initial Solution: 1 - 2 - 4 - 3 - 5. Initial Cost: 35.

2. Iteration 1:

   Current Solution: 1 - 2 - 4 - 3 - 5. Current Cost: 35.

   Best Solution: 1 - 2 - 4 - 3 - 5. Best Cost: 35.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		0	0	0
3	0	0		0	0
4	0	0	0		0
5	0	0	0	0	

Candidate Neighborhood Solutions:

1	**2**	**4**	**3**	**5**	**Cost**
1	4	2	3	5	43 + 0 = 43
1	3	4	2	5	28 + 0 = 28
1	5	4	3	2	30 + 0 = 30
1	2	3	4	5	30 + 0 = 30
1	2	5	3	4	32 + 0 = 32
1	2	4	5	3	24 + 0 = 24☆

3. Iteration 2:

Current Solution:	1 - 2 - 4 - 5 - 3.	Current Cost:	24.
Best Solution:	1 - 2 - 4 - 5 - 3.	Best Cost:	24.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		0	0	0
3	0	0		0	3
4	0	0	0		0
5	0	0	1	0	

Candidate Neighborhood Solutions:

1	**2**	**4**	**5**	**3**	**Cost**
1	4	2	5	3	31 + 0 = 31
1	5	4	2	3	29 + 0 = 29
1	3	4	5	2	18 + 0 = 18 ☆
1	2	5	4	3	18 + 0 = 18
1	2	3	5	4	33 + 0 = 33
1	2	4	3	5	35 + 1 = 36 **T**

4. Iteration 3:

Current Solution:	1 - 3 - 4 - 5 - 2.	Current Cost:	18.
Best Solution:	1 - 3 - 4 - 5 - 2.	Best Cost:	18.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		3	0	0
3	0	1		0	2
4	0	0	0		0
5	0	0	1	0	

Candidate Neighborhood Solutions:

1	**3**	**4**	**5**	**2**	**Cost**
1	4	3	5	2	32 + 0 = 32
1	5	4	3	2	30 + 1 = 31 **T**
1	2	4	5	3	24 + 1 = 25 **T**
1	3	5	4	2	24 + 0 = 24 ☆
1	3	2	5	4	26 + 0 = 26
1	3	4	2	5	28 + 0 = 28

5. Iteration 4:

Current Solution:	1 - 3 - 5 - 4 - 2.	Current Cost:	24.
Best Solution:	1 - 3 - 4 - 5 - 2.	Best Cost:	18.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		2	0	0
3	0	1		0	1
4	0	0	0		3
5	0	0	1	1	

Candidate Neighborhood Solutions:

1	**3**	**5**	**4**	**2**	**Cost**
1	5	3	4	2	35 + 1 = 36 **T**
1	4	5	3	2	33 + 0 = 33
1	2	5	4	3	18 + 1 = 19 **T**
1	3	4	5	2	18 + 1 = 19 **T**
1	3	2	4	5	29 + 0 = 29 ☆
1	3	5	2	4	31 + 0 = 31

6. Iteration 5:

Current Solution:	1 - 3 - 2 - 4 - 5.	Current Cost:	29.
Best Solution:	1 - 3 - 4 - 5 - 2.	Best Cost:	18.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		1	0	3
3	0	1		0	0
4	0	0	0		2
5	0	1	1	1	

Candidate Neighborhood Solutions:

1	**3**	**2**	**4**	**5**	**Cost**
1	2	3	4	5	30 + 1 = 31 **T**
1	4	2	3	5	43 + 0 = 43
1	5	2	4	3	28 + 1 = 29
1	3	4	2	5	28 + 0 = 28 ☆
1	3	5	4	2	24 + 1 = 25 **T**
1	3	2	5	4	26 + 1 = 27 **T**

7. Iteration 6:

Current Solution:	1 - 3 - 4 - 2 - 5.	Current Cost:	28.
Best Solution:	1 - 3 - 4 - 5 - 2.	Best Cost:	18.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		0	3	2
3	0	1		0	0
4	0	1	0		1
5	0	1	1	1	

Candidate Neighborhood Solutions:

1	**3**	**4**	**2**	**5**	**Cost**
1	4	3	2	5	37 + 0 = 37
1	2	4	3	5	35 + 1 = 36
1	5	4	2	3	29 + 1 = 30 ☆
1	3	2	4	5	29 + 1 = 30 **T**
1	3	5	2	4	31 + 1 = 32 **T**
1	3	4	5	2	18 + 1 = 19 **T**

8. Iteration 7:

Current Solution:	1 - 5 - 4 - 2 - 3.	Current Cost:	29.
Best Solution:	1 - 3 - 4 - 5 - 2.	Best Cost:	18.

Tabu Structure:

	1	2	3	4	5
1		0	0	0	0
2	0		0	2	1
3	0	1		0	3
4	0	1	0		0
5	0	1	2	1	

Candidate Neighborhood Solutions:

1	**5**	**4**	**2**	**3**	**Cost**
1	4	5	2	3	26 + 1 = 27 ☆
1	2	4	5	3	24 + 1 = 25 **T**
1	3	4	2	5	28 + 2 = 30 **T**
1	5	2	4	3	28 + 1 = 29 **T**
1	5	3	2	4	43 + 0 = 43
1	5	4	3	2	30 + 1 = 31

3.4.3 Solution Obtained:

Solution: 1 - 3 - 4 - 5 - 2. Cost: 18.

4. Performance Evaluation

These are the results obtained by implementing the algorithm in Matlab and running the program using two different initial solutions.

4.1 Example A

- Number of nodes n = 20.
- Symmetric TSP.
- T = Tabu Tenure (length).
- Optimal Cost: 63 (obtained by brute-force).
- Aspiration criteria: To allow a move, even if it is tabu, if it results in a solution with an objective value better than the current best-known solution.
- Results obtained with different number of iterations and tabu length:

Number of Iterations	T = 3	T = 4	T = 5	T = 6
200	66	66	68	68
100	66	66	68	68
60	66	66	68	68
30	70	70	70	70
20	70	70	70	70
10	78	78	78	78
5	87	87	87	87

4.2 Example B

- Number of nodes n = 20.
- Symmetric TSP.
- The initial solution is different than the initial solution used in example A.
- T = Tabu Tenure (length).
- Optimal Cost: 63 (obtained by brute-force).
- Aspiration criteria: To allow a move, even if it is tabu, if it results in a solution with an objective value better than the current best-known solution.
- Results obtained with different number of iterations and tabu length:

Number of Iterations	T = 3	T = 4	T = 5	T = 6
200	68	68	67	65
100	68	68	67	67
60	66	66	69	69
30	69	69	69	69
20	69	69	69	69
10	77	77	77	77
5	85	85	85	85

5. Conclusions

After analyzing both the Travelling Salesman Problem (TSP) and the Tabu Search (TS), and having implemented the TS algorithm in Matlab to solve the problem, we have reached the conclusions which will be discussed below.

Although we have not obtained the optimal solution, making a large number of iterations we have achieved almost optimal results.

In our experiment, the initial solution obtained has proved to be important, because with different initial solutions we have got different results. Therefore, it is a good idea to use a method to obtain a good initial solution, such as the nearest unvisited node method, rather than a solution randomly.

It is very important to a proper choice of parameters such as number of iterations or tabu tenure (length).

12. References

1. GLOVER, F. – *"Tabu Search: A Tutorial"*, Center for Applied Artificial Intelligence, University of Colorado.
   http://www.cse.unt.edu/~garlick/teaching/4310/assign/TS%20-%20Tutorial.pdf

2. JAYASWAL, S. – *"A Comparative Study of Tabu Search and Simulated Annelaing for Traveling Salesman Problem"*, Department of Management Sciences, University of Waterloo.
   http://www.eng.uwaterloo.ca/~sjayaswa/projects/MSCI703_project.pdf

3. KHAMIS, A. – *"Trajectory-based Optimization: Tabu Search"*, Universidad Carlos III de Madrid, 2011.

4. KHAMIS, A. – *"Trajectory-based Optimization: Simulated Annealing"*, Universidad Carlos III de Madrid, 2011.

5. COOK, W. – *"The Traveling Salesman Problem"*. http://www.tsp.gatech.edu

6. Wikipedia – *"Travelling salesman problem"*, Wikipedia, 2011.
   http://en.wikipedia.org/wiki/Travelling_salesman_problem

APPENDIX A: Matlab code - TSP with Tabu Search

```
function [best_solution, best_cost] = tabuSearchTSP (cost_matrix,
tabu_tenure, num_iterations)


    [initial_solution, initial_cost] = getInitialSolution(cost_matrix,1);

    % Si queremos empezar con una determinada solución inicial tenemos
    % que comentar la línea anterior e inicializar los parámetros:
        % initial_solution = [1 2 4 3 5];
        % initial_cost = 35;

    disp('-------------------------------------------------------------------
-----------------------------------------');
    fprintf('-------- INITIAL SOLUTION: '); disp(initial_solution);
    fprintf('-------- INITIAL COST: %d \n',initial_cost);
    disp('-------------------------------------------------------------------
-----------------------------------------');

    % Obtenemos el número de nodos.
    [num_rows, num_columns] = size(cost_matrix);    num_nodes = num_rows;

    % Inicializamos la lista tabú con todos sus valores a cero.
    tabu_list = initializeTabuList(num_nodes);

    current_solution = initial_solution; % La solución actual es la
solución inicial.
    best_solution = initial_solution; % La mejor solución es la solución
inicial
    best_cost = initial_cost; % El menor coste es el de la solución
inicial.

    for i=1:num_iterations
        fprintf('\n--------------- ITERATION %d ---------------\n',i);
        % Obtenemos la mejor solución de vecindad.
        [best_neighborhood_solution, best_neighborhood_cost, tabu_list] =
getNeighborhood (current_solution, best_cost, tabu_list, tabu_tenure,
cost_matrix);

        % La mejor solución de vencidad obtenida pasará a ser la solución
        % actual.
        current_solution = best_neighborhood_solution;

        % Comprobamos si se trata de la mejor solución encontrada hasta%
        % el momento.
        if (best_neighborhood_cost < best_cost)
            best_solution = best_neighborhood_solution;
            best_cost = best_neighborhood_cost;
        end
    end

    disp('-------------------------------------------------------------------
-----------------------------------------');
```

```
    fprintf('-------- THE BEST SOLUTION OBTAINED IN %d ITERATIONS:
',num_iterations); disp(best_solution);
    fprintf('-------- COST OF THE BEST SOLUTION: %d \n',best_cost);
    disp('-----------------------------------------------------------------
-----------------------------------------');
end
```

```
function [best_neighborhood_solution, best_neighborhood_cost, tabu_list]
= getNeighborhood (current_solution, best_cost, tabu_list, tabu_tenure,
cost_matrix)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que obtiene la mejor solución de vecindad realizando una serie
% de modificaciones en la solución actual. Generaremos las distintas
% soluciones de vecindad intercambiando la posición de dos de sus nodos
% en la solución actual y nos quedaremos con aquella solución que tenga
% un menor coste.
% Parámetros de entrada:
%   - current_solution: Solución actual.
%   - best_cost: Coste de la mejor solución obtenida hasta el momento.
%   - tabu_tenure: Tiempo que va a estar un swapping en la lista tabú.
%   - tabu_list: Lista de soluciones tabú.
%   - cost_matrix: Matriz de costes.
% Parámetros de salida:
%   - best_neighborhood_solution: La mejor solución de vecindad obtenida.
%   - best_neighborhood_cost: Coste de la solución de vecindad obtenida.
%   - tabu_list: Lista de soluciones tabú.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    fprintf('\nTHE BEST KNOWN SOLUTION COST: %d \n',best_cost);
    fprintf('\nCURRENT SOLUTION: ');disp(current_solution);

    best_neighborhood_solution = [];
    best_neighborhood_cost = 0;
    best_node1 = 0;
    best_node2 = 0;

    min_cost = 0;
    num_nodes = length(current_solution);

    for i=2:num_nodes
        % Obtenemos todas las posibles soluciones de vecindad.
        for j=i+1:num_nodes
            neighborhood_solution = current_solution;
            % Realizamos el swapping entre los nodos (i, j).
            neighborhood_solution(i) = current_solution(j);
            neighborhood_solution(j) = current_solution(i);

            fprintf('- CANDIDATE NEIGHBORHOOD SOLUTION:');
disp(neighborhood_solution);

            % Comprobamos que el swapping no es tabú.
            neighborhood_cost = getCostSolution (neighborhood_solution,
cost_matrix);
            % Penalizaremos los movimientos que se realicen con más
            % frecuencia para de este modo diversificar la búsqueda.
```

```
            swapping_frecuency =
getSwappingFrecuency(neighborhood_solution(i),neighborhood_solution(j),ta
bu_list);
            diversification_cost = neighborhood_cost +
swapping_frecuency;

            if
(isTabu(neighborhood_solution(i),neighborhood_solution(j),tabu_list) ==
false)
                % No es una solución tabú.

                fprintf('--- COST: %d + %d =
%d\n\n',neighborhood_cost,swapping_frecuency,diversification_cost);

                if ((min_cost == 0) || (diversification_cost < min_cost))
                    % Es la mejor solución de vecindad analizada hasta
                    % el momento.
                    min_cost = diversification_cost;
                    best_neighborhood_solution = neighborhood_solution;
                    best_neighborhood_cost = neighborhood_cost;
                    best_node1 = neighborhood_solution(i);
                    best_node2 = neighborhood_solution(j);
                end
            else
                % Es una solución tabú:
                % Para evitar que se produzcan estancamientos aplicaremos
                % como criterio de aspiración el que si el coste de la
                % solución es menor que el mejor obtenido hasta el
                % momento, se aceptará la solución.
                if (diversification_cost < best_cost)
                    % Es la mejor solución obtenida hasta el momento.
                    min_cost = diversification_cost;
                    best_neighborhood_solution = neighborhood_solution;
                    best_neighborhood_cost = neighborhood_cost;
                    best_node1 = neighborhood_solution(i);
                    best_node2 = neighborhood_solution(j);
                    disp('- TABU SOLUTION ALLOWED - ASPIRATION
CRITERIA:'); disp(neighborhood_solution);
                    fprintf('- ALLOWED TABU COST: %d + %d =
%d\n',neighborhood_cost,swapping_frecuency,diversification_cost);
                else
                    % Solución tabú no permitida.
                    fprintf('- IS A TABU NEIGHBORHOOD SOLUTION:');
disp(neighborhood_solution);
                    fprintf('--- TABU COST: %d + %d =
%d\n\n',neighborhood_cost,swapping_frecuency,diversification_cost);
                end
            end
        end
    end

    % Una vez obtenida la mejor de las soluciones de vecindad,
    % actualizamos la lista tabú decrementando en una unidad el tiempo de
    % estancia en la lista de los swappings que se encuentren incluidos
    % en ella y añadiendo el nuevo swapping a la lista.

    tabu_list = updateTabuList(tabu_list);
    fprintf('\nADD (%d,%d) TO TABU LIST: \n',best_node1,best_node2);
```

```
    tabu_list =
addSwappingTabuList(best_node1,best_node2,tabu_list,tabu_tenure);

    disp(tabu_list);
end
```

```
function [tabu_list] = initializeTabuList (num_nodes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que inicializa la matriz que representa la lista tabú con todos
% sus valores a cero.
% Parámetros de entrada:
%   - num_nodes: Número de nodos.
% Parámetros de salida:
%   - tabu_list: Lista tabú inicializada a 0.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    for i=1:num_nodes
        for j=1:num_nodes
            tabu_list(i,j) = 0;
        end
    end
end
```

```
function [tabu_list] = updateTabuList (tabu_list)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que actualiza la lista tabú en cada iteración, decrementando en
% una unidad el tiempo de permanencia en la lista de cada swapping que se
% encuentre en ella.
% Parámetros de entrada:
%   - tabu_list: Lista de soluciones tabú.
% Parámetros de salida:
%   - tabu_list: Lista tabú actualizada.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    [num_rows,num_columns] = size(tabu_list);
    num_nodes = num_rows;

    for i=1:num_nodes
        for j=i+1:num_nodes
            if (tabu_list(i,j) > 0)
                tabu_list(i,j) = tabu_list(i,j) - 1;
            end
        end
    end
end
```

```
function [tabu_list] = addSwappingTabuList (node1, node2, tabu_list,
tabu_tenure)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que añade un swapping a lista tabú, controlando la frecuencia
con
% la que se hace dicho swapping.
% Parámetros de entrada:
%   - node1, node2: Nodos que hacen el swapping.
%   - tabu_list: Lista de soluciones tabú.
%   - tabu_tenure: Número de iteraciones durante las cuales el swapping
va
%     a estar en la lista tabú.
% Parámetros de salida:
%   - tabu_list: Lista tabú actualizada.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    if (node1 < node2)
        % Inicializamos al valor de tabu_tenure el tiempo que va a estar
el
        % swapping en la lista tabú.
        tabu_list(node1,node2) = tabu_tenure;
        % Incrementamos el número de veces que hemos hecho este swapping.
        tabu_list(node2,node1) = tabu_list(node2,node1) + 1;
    else
        % Inicializamos al valor de tabu_tenure el tiempo que va a estar
el
        % swapping en la lista tabú.
        tabu_list(node2,node1) = tabu_tenure;
        % Incrementamos el número de veces que hemos hecho este swapping.
        tabu_list(node1,node2) = tabu_list(node1,node2) + 1;
    end
end
```

```
function [is_tabu_solution] = isTabu (node1, node2, tabu_list)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que comprueba si una solución pertenece a la lista tabú.
% Parámetros de entrada:
%   - node1, node2: Nodos que hacen el swapping.
%   - tabu_list: Lista de soluciones tabú.
% Parámetros de salida:
%   - is_tabu_solution: Variable booleana que indica si la solución
%     pertenece a la lista de soluciones tabú.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    is_tabu_solution = false;

    if (node1 < node2)
        % Comprobamos la posición (node1,node2) de la tabla tabú.
        if (tabu_list(node1,node2) > 0)
            % Es una solución tabú.
            is_tabu_solution = true;
        end
    else
        % Comprobamos la posición (node2,node1) de la tabla tabú.
        if (tabu_list(node2,node1) > 0)
            % Es una solución tabú.
```

```
            is_tabu_solution = true;
        end
    end
end
```

```
function [swapping_frecuency] = getSwappingFrecuency (node1, node2,
tabu_list)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que obtiene la frecuencia con la que se ha realizado un
swapping
% entre dos nodos.
% Parámetros de entrada:
%   - node1, node2: Nodos que hacen el swapping.
%   - tabu_list: Lista de soluciones tabú.
% Parámetros de salida:
%   - tabu_list: Lista tabú actualizada.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    if (node1 < node2)
        % Obtenemos el valor de la posición (node2,node1) de la lista
tabú.
        swapping_frecuency = tabu_list(node2,node1);
    else
        % Obtenemos el valor de la posición (node1,node2) de la lista
tabú.
        swapping_frecuency = tabu_list(node1,node2);
    end
end
```

```
function [cost_solution] = getCostSolution (solution, cost_matrix)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que obtiene el coste de una solución.
% Parámetros de entrada:
%   - solution: Solución de la que queremos obtener su coste.
%   - cost_matrix: Matriz que contiene los costes entre nodos.
% Parámetros de salida:
%   - cost_solution: Coste de la solución.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    cost_solution = 0;
    for i=1:(length(solution) - 1)
        cost_solution = cost_solution +
cost_matrix(solution(i),solution(i+1));
    end
    % Calculamos el coste existente entre el último nodo y el inicial.
    cost_solution = cost_solution +
cost_matrix(solution(length(solution)),solution(1));
end
```

```
function [initial_path, total_cost] = getInitialSolution (cost_matrix,
initial_node)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
```

```
% Función que obtiene la solución inicial elegiendo el nodo más cercano.
% Parámetros de entrada:
%   - cost_matrix: Matriz con los costes entre nodos.
%   - initial_node: Nodo inicial del que partir.
% Parámetros de salida:
%   - initial_path: Ruta obtenida que será la solución inicial.
%   - total_cost: Coste total de la solución inicial.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    [num_rows, num_columns] = size(cost_matrix);

    num_nodes = num_rows; % Numer of nodes.

    initial_path(1) = initial_node;
    total_cost = 0; % Coste total.

    while (length(initial_path) < num_nodes)
        current_node = initial_path(length(initial_path)); % Nodo actual.

        % Obtenemos la lista de costes para el nodo actual:
        node_cost_list = cost_matrix(current_node,:);

        % Obtenemos el nodo más cercano al nodo ctual:
        [nearest_node, min_cost] = getNearestNode (node_cost_list,
initial_path);

        initial_path(length(initial_path) + 1) = nearest_node;
        total_cost = total_cost + min_cost;
    end

    % Incluimos de nuevo el nodo inicial.
    total_cost = total_cost + cost_matrix(nearest_node, initial_node);

end
```

```
function [is_visited_node] = inVisitedNodes (node, visited_nodes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que comprueba si un nodo pertenece a la lista de nodos
visitados.
% Parámetros de entrada:
%   - node: Nodo que queremos comprobar.
%   - visited_nodes: Lista de nodos visitados (ruta actual).
% Parámetros de salida:
%   - is_visited_node: Variable booleana que indica si el nodo pertenece
a
%   la lista de nodos visitados.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    is_visited_node = false;
    num_visited_nodes = length(visited_nodes);

    for i=1:num_visited_nodes
        if (node == visited_nodes(i))
```

```
            is_visited_node = true;
            break;
        end
    end
end
```

```
function [nearest_node, min_cost] = getNearestNode (node_cost_list,
initial_path)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% Función que obtiene el nodo más cercano al nodo actual que no haya sido
% visitado previamente.
% Parámetros de entrada:
%   - node_cost_list: Vector con los costes del nodo actual.
%   - initial_path: Ruta obtenida hasta el momento.
% Parámetros de salida:
%   - nearest_node: Nodo más cercano.
%   - min_cost: Coste de ir al nodo más cercano.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

    num_nodes = length(node_cost_list); % Número de nodos.
    visited_nodes = initial_path;
    min_cost = 0; % Coste mínimo.
    nearest_node = 0; % Nodo seleccionado.

    for node=1:num_nodes
        % Comprobamos que no hemos visitado el nodo.
        if (inVisitedNodes(node, visited_nodes) == false)
            % No hemos visitado el nodo
            if ((min_cost == 0) || (node_cost_list(node) < min_cost))
                % Es el nodo más cercano de los analizados hasta ahora.
                min_cost = node_cost_list(node);
                visited_nodes(size(visited_nodes) + 1) = node;
                nearest_node = node;
            end
        end
    end
end
```