Optimization Methods/Meta-Heuristics
Artificial Intelligence – Prof. Luís Paulo Reis

Gabriel Carvalhal e Wagner Ceulin

Mestrado em Engenharia e Ciência de Dados – Universidade do Porto

Abstract. This paper explores the TRAVELLING SALESMAN PROBLEM. This problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". Hill-climbing, simulated annealing, tabu search and genetic algorithms were used to analyze the best route for different sets of cities.

1 Specification of the Work Performed

1.1 Practical work topic

The project goal is to answer the practical work Topic 3: Metaheuristics for Optimization/Decision Problems. As indicated in the practical work paper, “an optimization problem is characterized by the existence of a (typically large) set of possible solutions, comparable to each other, of which one or more are considered (globally) optimal solutions. Depending on the specific problem, an evaluation function allows you to establish this comparison between solutions. In many of these problems, it is virtually impossible to find the optimal solution, or to ensure that the solution found is optimal and, as such, the goal is to try to find a local optimal solution that maximizes/minimizes a given evaluation function to the extent possible”. [1]

1.2 Problem definition

Our team choose to study the TRAVELLING SALESMAN PROBLEM. This problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?".

The problem was first formulated (mathematically) in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, many heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded inside an optimal control problem. [2]

This problem is easy to declare but very difficult to solve. When completed in exact time the required computation will increase exponentially as the problem increases.

TSP can be expressed as a problem in finding the minimum distance of a closed journey to a number of n cities where the cities are only visited once. TSP is represented by using a complete and weighted graph G = (V, E) with V set of vertices representing a set of points, and E is the set of edges. Each edge (r, s) ∈ E is the value (distance) rs which is the distance from city r to city s, with (r, s) ∈ V. In a symmetric TSP (distance from city r to point s is equal to distance from point S to point r), drs = drs for all edges (r, s) ∈ E . In a graph, the TSP is drawn as shown in Figure 1 below: [3]

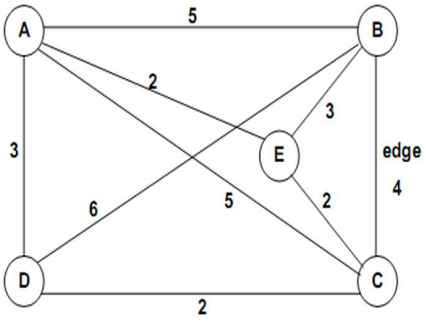


Fig.1 – The illustration of TSP

2 Related Works

There are lots of papers, articles and thesis regarding the TSP. We used some of these texts as a guide for our work. We also used the slides presented by the teacher in the classroom.

We do not intend to create new technologies in this work, but to use best practices. That's why we highlight below the best definitions we found, indicating the authors.

2.1 Hill-Climbing

"Hill climbing is a mathematical optimization algorithm, which means its purpose is to find the best solution to a problem which has a (large) number of possible solutions. In the Travelling salesman problem, the distances between each pair of cities are known, and we need to find the shortest route. The goal is to find the best (i.e. the shortest) solution. It tries to find the best solution to this problem by starting out with a random solution, and then generate neighbours: solutions that only slightly differ from the current one. If the best of those neighbours is better (i.e. shorter) than the current one, it replaces the current solution with this better solution. It then repeats the pattern by again creating neighbours. If at some point no neighbour is better than the current solution, it returns the then current solution". [4]

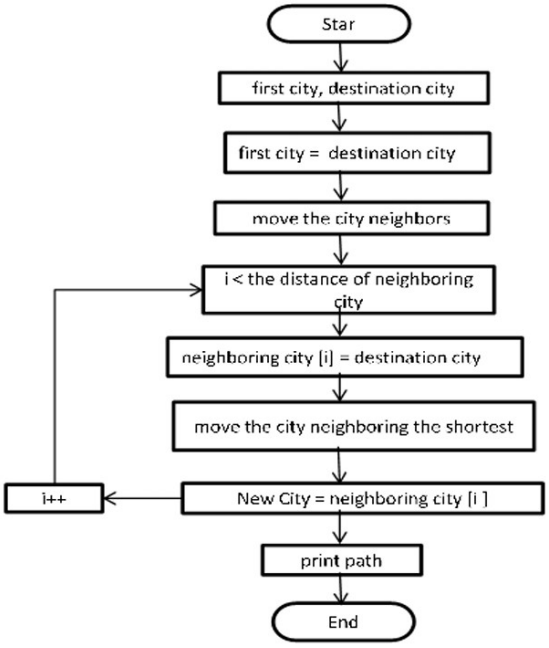


Fig.2 – Hill-Climbing flowchart [3]

2.2 Simulated Annealing

"The Simulated Annealing (SA) approach begins with a random solution and every iteration forms a random closely solution. If the closely solution is an over top answer, it would substitute by the present solution, but if it is a worsen answer, it may be chosen to substitute the present solution with a probability that will be dependent on the temperature variable (parameter). As the approach continues, the temperature variable (parameter) reduces, thus giving worse answer a lower chance of substituting the present

answer. Allowing worse answers assists to stay away from converging to a local minimum rather than the global minimum". [5]

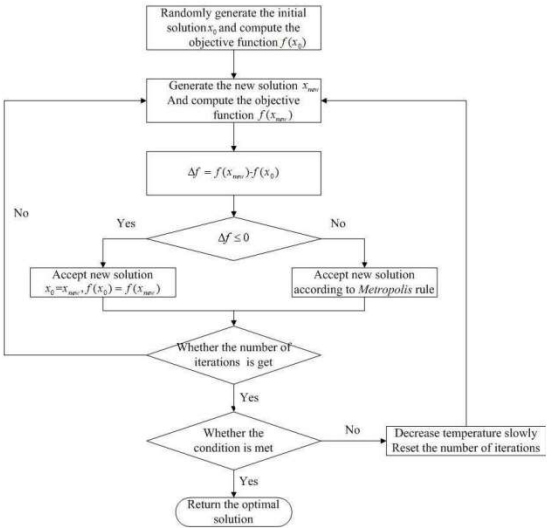


Fig.3 – Simulated Annealing flowchart [6]

2.3 Tabu Search

"Tabu Search is a Global Optimization algorithm and a Metaheuristic for controlling an embedded heuristic technique. Tabu Search algorithm is to force an embedded heuristic from returning to recently visited areas of the search space, which is called as cycling. The plan of this approach is to maintain a temporary memory for all the changes of recent moves within the search space and preventing future moves from undoing those changes. In addition to this, an intermediate memory will be used for other search spaces. Tabu search is based on local search. The search starts with an initial solution for the problem. That initial solution is taken as a current solution and searches for the best solution i.e. a group of tours that can be simply reached from the current solution. After this step, it takes the best solution from the neighborhood as the current solution and the search process continues. Tabu search is terminated when it meets the maximum iteration count conditions, solution quality objectives or involving in the execution time". [7]

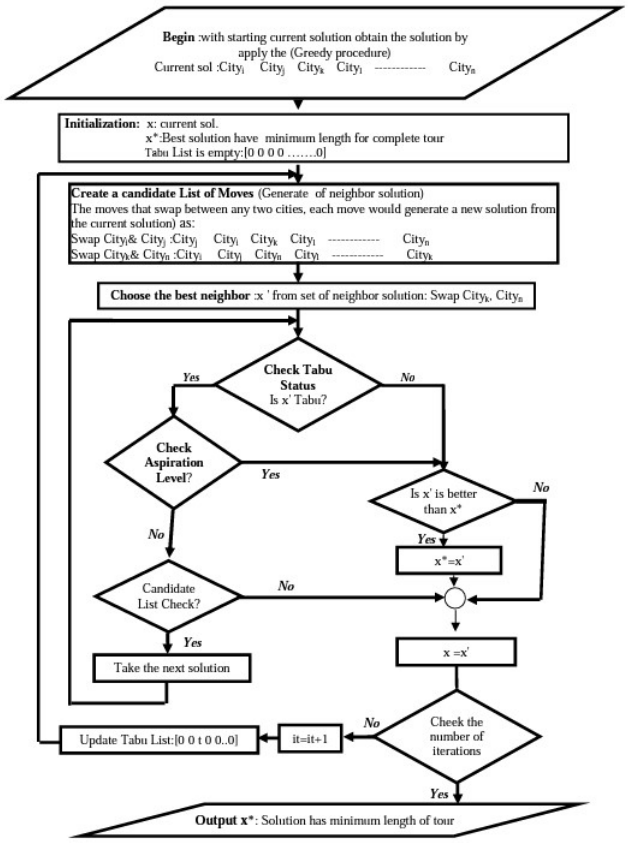


Fig.4 – Tabu Search flowchat [8]

2.4 Genetic Algorithms

"Genetic Algorithm (GA) is a search-based algorithm inspired by Charles Darwin's theory of natural evolution. GA follows the notion of natural selection. The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found. We can use this notion for search-based algorithm. We can formally state this process in as following phases: 1. Initial population, 2. Fitness Function, 3. Selection, 4. Cross-over and 5. Mutation." [9]

"Genetic algorithms appear to find good solutions for the traveling salesman problem, however it depends very much on the way the problem is encoded and which crossover and mutation methods are used". [10]

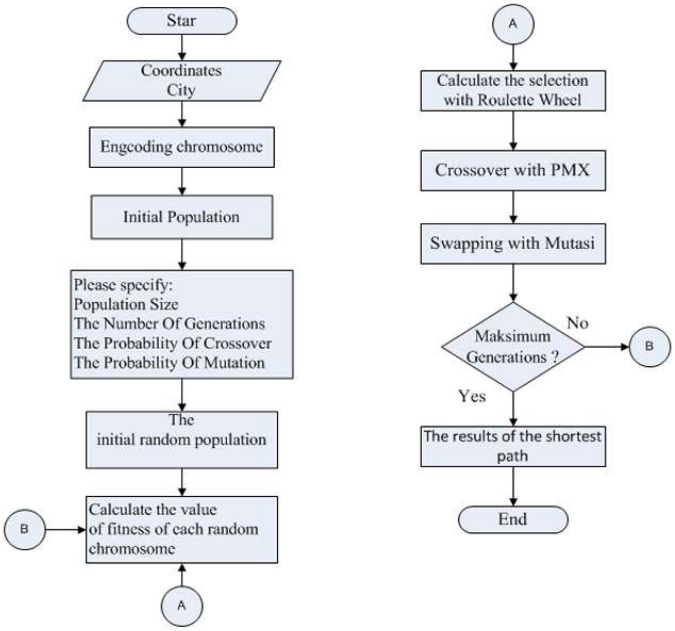


Fig. 5 – Genetic Algorithms flowchart [3]

2.5 Solving Optimization Problems

"Heuristic algorithms do not guarantee to find the optimal solution and do not have a bound on quality, that is, they can return a solution that is arbitrarily bad compared to the optimal solution. However, in practice, heuristic algorithms (heuristics for short) have proven very successful in solving large real-world problems". [11]

3 Formulation of the Problem as an Optimization Problem

The idea of the TSP is to find a tour of a given number of cities/locations, visiting each of then exactly once and returning to the starting city/location where the length of this tour is minimized.

In this work, hill-climbing, simulated annealing, tabu search and genetic algorithms has been used to analyze the best route and the time spent to process the solution for the following sets:

- **4 x 4 matrix (used for testing)**: travel between Porto, Braga, Aveiro and Viana do Castelo.
- **14 x 14 matrix**: travel between Lisboa, Porto, Braga, Setubal, Coimbra, Evora, Aveiro, Leiria, Faro, Viana do Castelo, Beja, Braganca, Castelo Branco and Guarda.
- **37 x 37 matrix**: travel between Amsterdam, Antwerp, Athens, Barcelona, Berlin, Bern, Brussels, Calais, Cologne, Copenhagen, Edinburgh, Frankfurt, Geneva, Genoa, Hamburg, Le Havre, Lisbon, London, Luxembourg, Lyon, Madrid, Marseille, Milan, Munich, Naples, Nice, Paris, Prague, Rome, Rotterdam, Strasbourg, Stuttgart, The Hague, Turin, Venice, Vienna and Zurich.

Those datasets were obtained at the website https://pt.melhoresrotas.com/ [12] for the Portuguese cities and in https://www.engineeringtoolbox.com/ [13] for the European cities. Latitude and longitude data were found at the web site https://simplemaps.com/ [14]. More details could be found in the python notebook presented as part of this work.

We aim to present a comparative study among those algorithms for solving TSP. Some algorithms have not found a better solution to the traveling salesman problem than is already known. It is very difficult to say that what moderate sized instance is unsolvable, so an incorporation of good local search technique to the algorithm may solve exactly the other instances, which is under our investigation and is categorized under future work.

4 Implementation Details

All the work has been developed in Python, using the Jupyter Notebook (anaconda3) to run the program. The following libraries has been used: numpy, pandas, gmplot, time, random, math, matplotlib, copy, sys, datetime, warnings, sys, random, copy and dataframe_image.

The datasets used were obtained at the internet, see references [12], [13] and [14].

The results presented were obtained on a desktop with the following configurations: • MS Windows 10 Pro OS – Version 10.0.19043 Build 19043 • System type X64-based PC • Intel(R) Core (TM) i7-4771 CPU @ 3.50GHz, 3501Mhz, 4 Cores, 8 Processors • BIOS Version/Date American Megatrends Inc. 3503, 4/18/18 • BaseBoard GRYPHON Z97 ARMOR EDITION (ASUS) • Memory (RAM) 32.0 GB • Graphics Card NVIDIA GeForce GTX 1650 4Gb.

5 Approach and Algorithms Implemented

All selected algorithms were executed with 500, 1.000, 5.000 and 10.000 iterations. The specific functions of each algorithm, such as stopping criteria, evaluation functions and other operators are shown below.

5.1 Hill-Climbing:

Hill climbing tries to find the best solution to this problem by starting out with a random solution, and then generate neighbours: solutions that only slightly differ from the current one. If the best of those neighbours is better (i.e. shorter) than the current one, it replaces the current solution with this better solution. It then repeats the pattern by again creating neighbours. If at some point no neighbour is better than the current solution, it returns the then current solution.

- Iterations: 500, 1.000, 5.000 and 10.000.
- Neighborhood: random swap among two neighbors.
- Mutation rate: (optional), a number, from 0.00 to 1.00. Randomly discards neighbors depending on the chosen rate, with 1 being equivalent to not discarding any. (mutation_rate = 0.7, i.e discarding 30% of the neighbors).

STEP 1: Hill-Climbing starts generating a random solution, so we define a function to create a random solution generator.

```
def get_random_solution(matrix:[], home:int, city_indexes:[]
    # Create a list with city indexes
    cities = city_indexes.copy()
    # Remove the home city
    cities.pop(home)
    # Create a population
    population = []
    for i in range(size):
        # Shuffle cities at random
        random.shuffle(cities)
        # Create a state
        state = State(cities[:])
        state.update_distance(matrix, home)
        # Add an individual to the population
        population.append(state)
    # Sort population
    population.sort()
    # Return the best solution
    return population[0]
```

STEP 2: We want our Hill climber to find the shortest solution, so we need a function calculating the length of a specific solution.

```
# Update distance
def update_distance(self, matrix, home):

    # Reset distance
    self.distance = 0
    # Keep track of departing city
    from_index = home
    # Loop all cities in the current route
    for i in range(len(self.route)):
        self.distance += matrix[from_index][self.route[i]]
        from_index = self.route[i]
    # Add the distance back to home
    self.distance += matrix[from_index][home]
```

STEP 3: Create a function to defining neighbors. To make our hill climbing less greedy we can randomly select among neighbors, discarding some. Setting value to 1 or greater deactivate the function.

```
# Mutate a solution
def mutate(matrix:[], home:int, state:State, mutation_rate:float=0.01):

    # Create a copy of the state
    mutated_state = state.deepcopy()
    # Loop all the states in a route
    for i in range(len(mutated_state.route)):
        # Check if we should do a mutation
        if(random.random() < mutation_rate):
            # Swap two cities
            j = int(random.random() * len(state.route))
            city_1 = mutated_state.route[i]
            city_2 = mutated_state.route[j]
            mutated_state.route[i] = city_2
            mutated_state.route[j] = city_1
    # Update the distance
    mutated_state.update_distance(matrix, home)
    # Return a mutated state
    return mutated_state
```

STEP 4: Create a function finding the best neighbour. The best solution is mutated in each iteration and a mutated solution will be the new best solution if the total distance is less than the distance for the current best solution.

```
# Get best solution by distance
def get_best_solution_by_distance(matrix:[], home:int):

    # Variables
    route = []
    from_index = home
    length = len(matrix) - 1
    # Loop until route is complete
    while len(route) < length:
        # Get a matrix row
        row = matrix[from_index]
        # Create a list with cities
        cities = {}
        for i in range(len(row)):
            cities[i] = City(i, row[i])
        # Remove cities that already is assigned to the route
        del cities[home]
        for i in route:
            del cities[i]
        # Sort cities
        sorted = list(cities.values())
        sorted.sort()
        # Add the city with the shortest distance
        from_index = sorted[0].index
        route.append(from_index)
    # Create a new state and update the distance
    state = State(route)
    state.update_distance(matrix, home)
    # Return a state
    return state
```

STEP 5: Hill climbing algorithm creates a random solution and calculate its route length. We then create the neighbouring solutions, and find the best one. From there on, as long as the best neighbour is better than the current solution, we repeat the same pattern with the current solution each time being updated with the best neighbour. When this process stops, we return the current solution (and its route length).

```
# Setting mutation rate:
mut_rate = 0.7

# Hill climbing algorithm
def hill_climbing(matrix:[], home:int, initial_state:State, max_iterations:int, mutation_rate:float=0.01):
    # Keep track of the best state
    best_state = initial_state
    # An iterator can be used to give the algorithm more time to find a solution
    iterator = 0
    #List row
    row =[]
    # Create an infinite loop
    while True:
        # Mutate the best state
        neighbor = mutate(matrix, home, best_state, mutation_rate)
        # Check if the distance is less than in the best state
        if(neighbor.distance >= best_state.distance):
            iterator += 1
            if (iterator >= max_iterations):
                break
        if(neighbor.distance < best_state.distance):
            best_state = neighbor
        row = row+ [[iterator,neighbor.distance]]
    # Return the best state
    return best_state, row
```

STEP 6: At the end, we define the function to run with our dataset getting the solution.

```
#Get time of processing
start_time = time.time()

# Distances in km between cities, same indexes (i, j) as in the cities array
matrix = cities4_4_hill_climbing
# Cities to travel
cities = ['Porto','Braga','Aveiro','Viana do Castelo']
city_indexes = [0,1,2,3]

# Index of start location
home = random.randrange(0,4)

# Max iterations
max_iterations = num_inter

# Distances in km between cities, same indexes (i, j) as in the cities array
matrix = cities4_4_hill_climbing

# Run hill climbing to find a better solution
state = get_best_solution_by_distance(matrix, home)
state,iterations = hill_climbing(matrix, home, state, max_iterations, mut_rate)

print('-- Hill climbing solution --')
print(cities[home], end='')
df = [cities[home]]
for i in range(0, len(state.route)):
    print(' -> ' + cities[state.route[i]], end='')
    df.append(cities[state.route[i]])
print(' -> ' + cities[home], end='')
df.append(cities[home])
print('\n\nTotal distance: {0} km'.format(state.distance))
print()

#time
hill_teste_time = time.time() - start_time
print("--- %s seconds ---" % (hill_teste_time))
```

Stopping criteria's of Hill-Climbing algorithm is:

- The algorithm performed all the maximum iterations established.

5.2 Simulated Annealing:

The Simulated Annealing algorithm is commonly used when we're stuck trying to optimize solutions that generate local minimum or local maximum solutions, for example, the Hill-Climbing algorithm.

- Iterations: 500, 1.000, 5.000 and 10.000.
- Cooling schedule: Initial temperature: sated to 1000º / Final temperature: 0º / Rate: ~0.005. The temperature is a parameter that affects two aspects of the algorithm: (a) The distance of a trial point from the current point and (b) The probability of accepting a trial point with higher objective function value.
- Neighborhood: random swap among two neighbors.
- Mutation rate: (optional), a number, from 0.00 to 1.00. Randomly discards neighbors depending on the chosen rate, with 1 being equivalent to not discarding any. (mutation_rate = 0.7, i.e discarding 30% of the neighbors).

STEPS 1 to 4: The same we use for Hill-Climbing.

STEP 5: We have to determine how we will reduce the temperature on each iteration. We started with a temperature of 1000 degrees, and we will decrease the current temperature at a -0,5% rate trying to reach the final temperature of 0 degrees. We will repeat this process until the current temperature is less than the final temperature.

```
# Setting mutation rate and temperature:
mut_rate = 0.5
initial_temp = 1000
# Schedule function for simulated annealing
def exp_schedule(k=initial_temp, lam=0.005, limit=num_inter):
    return lambda t: (k * np.exp(-lam * t) if t <= limit else 0)
```

STEP 6: If the new solution is better, we will accept it, but if the new solution is not better, we will still accept it if the temperature is high. With this approach, we will use the worst solution in order to avoid getting stuck in local minimum. But we will get a neighbor that is not much worse than the current state. We can determine defining these functions:

```
# Return true with probability p
def probability(p):
    return p > random.uniform(0.0, 1.0)
```

```
# Calculate the change in e
delta_e = best_state.distance - neighbor.distance
# Check if we should update the best state
if delta_e > 0 or probability(np.exp(delta_e / T)):
    best_state = neighbor
```

STEP 7: This is the big picture for Simulated Annealing algorithm, which is the process of taking the problem and continuing with generating random neighbors. We'll always move to a neighbor if it's better than our current state. But even if the neighbor is worse than our current state, we'll sometimes move there depending on the temperature and how bad the neighbor is. Finally, we stop if the maximum iteration has been reached.

```
# Setting mutation rate and temperature:
mut_rate = 0.7
initial_temp = 1000

# Return true with probability p
def probability(p):
    return p > random.uniform(0.0, 1.0)

# Schedule function for simulated annealing
def exp_schedule(k=initial_temp, lam=0.005, limit=num_inter):
    return lambda t: (k * np.exp(-lam * t) if t <= limit else 0)

# Simulated annealing
def simulated_annealing(matrix:[], home:int, initial_state:State, mutation_rate:float=0.01, schedule=exp_schedule()):
    # Keep track of the best state
    best_state = initial_state
    #list row
    row =[]
    # Loop a large number of times (int.max)
    for t in range(sys.maxsize):
        # Get a temperature
        T = schedule(t)
        # Return if temperature is 0
        if T == 0:
            return best_state, row
        # Mutate the best state
        neighbor = mutate(matrix, home, best_state, mutation_rate)
        # Calculate the change in e
        delta_e = best_state.distance - neighbor.distance
        # Check if we should update the best state
        if delta_e > 0 or probability(np.exp(delta_e / T)):
            best_state = neighbor
        row = row +[[t, neighbor.distance,T]]
    # Return the best state
    return best_state, row
```

STEP 8: At the end, we define the function to run with our dataset getting the solution.

```
#Get time of processing
start_time = time.time()

# Cities to travel
cities = ['Porto','Braga','Aveiro','Viana do Castelo']
city_indexes = [0,1,2,3]
# Index of start location
home = random.randrange(0,4)
# Distances in miles between cities, same indexes (i, j) as in the cities array
matrix = cities4_4_simulated_annealing

# Run simulated annealing to find a better solution
state = get_best_solution_by_distance(matrix, home)
state = simulated_annealing(matrix, home, state, mut_rate)
print('-- Simulated annealing solution --')
print(cities[home], end='')
df = [cities[home]]
for i in range(0, len(state.route)):
   print(' -> ' + cities[state.route[i]], end='')
   df.append(cities[state.route[i]])
print(' -> ' + cities[home], end='')
df.append(cities[home])
print('\n\nTotal distance: {0} km'.format(state.distance))

#time
SimuAnnealing_teste_time = time.time() - start_time
print("--- %s seconds ---" % (SimuAnnealing_teste_time))
```

Stopping criteria's of simulated annealing algorithm are:

- The algorithm performed all the maximum iterations established.
- A given minimum value of the temperature has been reached.

5.3 Tabu Search:

Tabu search uses a local or neighborhood search procedure to iteratively move from a solution to another solution until some stopping criterion has been satisfied. The aim is to explore regions of the search space that would be left unexplored by the local search procedure. Tabu search modifies the neighborhood structure of each solution as the search progresses. This metaheuristic guarantees a near optimal solution, using the concept of short term of memory space.

- Iterations: 500, 1.000, 5.000 and 10.000.
- Tabu tenure: the duration for which a move will be kept tabu. This parameter is selected depending on the size n of the problem by the algorithm T=sqrt(n)
    - 4 x 4 matrix = 2 iterations
    - 14 x 14 matrix = 4 iterations
    - 37 x 37 matrix = 6 iterations
- Neighborhood: identify neighborhood set by creating many solutions with move operation
    - 4 x 4 matrix = 1
    - 14 x 14 matrix = 1
    - 37 x 37 matrix: = 1

STEP 1: Tabu search is based on local search, which starts with an initial solution for the problem. That initial solution is taken as a current solution and searches for the best solution, in which a group of tours that can be simply reached from the current solution. After this step, it takes the best solution from the neighborhood as the current solution and the search process continues. With the sample project datasets, the algorithm starts with a range from 1 to the length of all the cities of the matrices plus one.

Input the number of cities (data), and the distances between them.

```
#Data
data = cities4_4.reset_index()
```

The *solution variable* is the initial solution, which is taken as a current solution

```
# Current solution initialization
solution = list(range(1, len(data) + 1)) # List of all job indexes from the file
```

Tabu search is terminated when it meets the maximum iteration count conditions, solution quality objectives or involving in the execution time. Therefore, the algorithm uses three parameters.

Parameter 1: Iterations (iterations) is the number of times that the current solution will be revaluated with the goal of finding a better optimal solution. Criteria for stopping the algorithm after several iterations.

Parameter 2: Tabu length (*tabu_len*) keeps the last solution for a specific number of new interactions. This is usually obtained by keeping track of the last solutions in terms of the action used to transform one solution to the next. When an action is performed it is considered tabu for the next T iterations, where T is the tabu status length.

T {is number of iterations for move will be kept tabu where t= sqrt(n)}

Parameter 3: Identify Neighborhood set by creating many solutions with used move operation. The move operation in the TSP is making swap between two cities or n number of cities randomly of the current solution.

```
# Setting of the stop criterion
iterations = num_inter # Parameter 1
tabu_len = 2 # Parameter 2 (sqrt(n))
neighbourhood = 1 #Parameter 3
```

STEP 2: Define the current solution as the best solution in the variable (*solution*). After it, calculate the aspiration level with the function (*count_score*). To be the first benchmark used inside the loop iteration. Aspiration criterion is used to override the tabu list when there is a good tabu move aspiration criterion. So, the tabu move is accepted if during the iterations, it produces better solution than the best obtained so far.

```
# Current solution initialization
solution = list(range(1, len(data) + 1)) # List of all job indexes from the file

# Creating array in order to conduct goal function calculations
m = np.delete(data.to_numpy(), 0, 1) # The first arrays column removal, because it contains indexes

# Solution Score
score = count_score(solution)
```

Also, we define three list for the looping processes:

- Tabu: Final list with the best possible route found.
- Candidates: Store the candidates list for evaluation. List used in the candidates_generator function.
- Scores_Tabu_teste: Store the evaluation scores of the new solutions inside the loop. Important for the concept of short term of memory space.

```
# Tabu list initialization
tabu = [] # Tabu list

# Creation of a candidate list of neighbors to the current solution
candidates = [] # A list of all candidates
scores_tabu_teste = []
```

Function *count_score* will calculate the total distance of the current route found by the algorithm.

```
def count_score(o): # Functuion responsible for counting score
    time_travel, copied_solution = 0, o.copy()
    time_travel = np.double(time_travel)
    for i in range(0, len(m) - 1):
        first_city, second_city = copied_solution[i], copied_solution[i + 1]
        time_travel += m[first_city - 1, second_city - 1]
    time_travel += m[copied_solution[0] - 1, copied_solution[-1] - 1]
    return time_travel
```

STEP 3: At this point, the while stop criterion loop starts, in order to find the best possible solution. Inside the looping, the algorithm uses two more defined functions, such as *candidates_ generator* and *swap_method*.

Firstly, the while loop generates candidates, identifying neighborhood set by creating many solutions with used move operation. If the move is considered, it is held in the list until it moves is done and a new score is set. The move operation in TSP is making swap between any two cities randomly on the current solution.

```
def candidates_generator(n): # The function generating all candidates
    k, x, y = 0, 0, 0
    for k in range(0, neighbourhood):
        while True:
            while True:
                x, y = choice(n), choice(n)
                if x != y:
                    break
            if [x, y] not in candidates:
                candidates.append([x, y]) # Populating list with small lists of two numbers
                break
    k += 1
```

The *swap_method* function is responsible for changing positions in the list. It replaces a selected distance between two cities by another distance.

```
def swap_method(t1, y): # The first type of neighbourhood
    x = solution.index(t1)
    copied_solution = new_solution.copy() # Copied solution which is used for experiments
    copied_solution[x], copied_solution[y] = copied_solution[y], copied_solution[x] # Swapping positions of these jobs in
    neighbourhood_type = "swap"
    return copied_solution, neighbourhood_type
```

Swapping the positions, means that a new solution with a different score was generated by the function. Therefore, if the new score (total distance) is lower than the previous, the new value will be stored as a new current solution. Otherwise, it will check tabu length for preparing a new candidate list for the next iteration.

```
#Generate Solution
i = 0
while i != iterations:
    z = candidates_generator(solution) # Generating candidates
    min_solution, min_score = solution.copy(), score
    tabu_pair = [] #If the move is considered it is held in this list untill it move is done and a new socre is set

    j = 0
    while j != neighbourhood: # checking neighbourhood
        pair, new_solution, new_score = candidates[j].copy(), solution.copy(), 0 #candidates_score[j]
        if pair not in tabu:
            t1 = pair[0]
            y = new_solution.index(pair[1])
            new_solution, neighbourhood_type = swap_method(t1, y)
            new_score = count_score(new_solution)
            #Get distances
            scores_tabu_teste.append(min_score)
            if new_score < min_score: # Checking if a new socre is better than local socre
                min_solution, min_score, tabu_pair = new_solution.copy(), new_score, pair.copy()

        j += 1
    if min_score < score: # Checking if local score can replace previous score
        solution, score = min_solution.copy(), min_score
        tabu.append(tabu_pair)
    if len(tabu) == tabu_len: # Controlling the tabu list lenght
        del tabu[0]
    candidates.clear() # Preparing candidates list for the next iteration
    i += 1
print(f"The results of Tabu Search algorithm for Teste")
print(f"Solution: {solution}")

#Time
tabu_teste_time = time.time() - start_time
print("--- %s seconds ---" % (tabu_teste_time))
```

Stopping criteria's of tabu search algorithm are:

- Overused resources as money, time, and computation.
- The algorithm seems unable to find something better and the current solution satisfies the requirements.
- Maximum iteration count conditions.
- Solution quality objectives.
- Maximum execution time.

5.4 Genetic Algorithms:

- Iterations: 500, 1.000, 5.000 and 10.000
- names_list: list of cities used in genesis functions to generate randomly population routes
    - 4 x 4 matrix
    - 14 x 14 matrix
    - 37 x 37 matrix

- n_cities: length of cities in each matrix
    - 4 x 4 matrix = 4
    - 14 x 14 matrix =14
    - 37 x 37 matrix =37

- n_population: number of populations that will be generated in genesis function by the initial progenitor (n_population = 50)

- mutation_rate: Defines the percentage of progenitors that will be mutate for a new evaluation. (mutation_rate = 0.3)

```
#Data transformation
names_list = list(cities4_4)
names_list = np.array(names_list)

#Parameters
#number of cities
n_cities = len(cities4_4)
#Population generated
n_population = 100
#mutation rate of progenitors
mutation_rate = 0.1
```

At firsty, the algorithm uses the genesis function to randomly defines N possible solutions to the problem. It doesn't need to be the best, because they will be the seed upon which later the best solution will be found.

```
#define population set
population_set = genesis(names_list, n_population)
```

Genesis function creates a loop with n_population to generate randomly new route solutions.

```
# First step: Create the first population set
def genesis(city_list, n_population):

    population_set = []
    for i in range(n_population):
        #Randomly generating a new solution
        sol_i = city_list[np.random.choice(list(range(n_cities)), n_cities, replace=False)]
        population_set.append(sol_i)
    return np.array(population_set)
```

After generating a set of N possible solutions, the algorithm uses a fitness function to evaluate each of the route individually and access one by one.

```
#define fitnes list
fitnes_list = get_all_fitnes(population_set,cities_dict)
```

Fitness is a metric that represents how well each of the individual solutions performs for our model. In travel salesman problem, the algorithm is looking for the best possible route, which it means a minimization procedure. Therefore, the lowest fitness values return the best possible routes.

```
#define fitnes list

def get_all_fitnes(population_set, cities_dict):
    fitnes_list = np.zeros(n_population)

    #Looping over all solutions computing the fitness for each solution
    for i in  range(n_population):
        fitnes_list[i] = fitness_eval(population_set[i], cities_dict)

    return fitnes_list
def fitness_eval(city_list, cities_dict):
    total = 0
    for i in range(n_cities-1):
        a = city_list[i]
        b = city_list[i+1]
        total += compute_city_distance_names(a,b, cities_dict)
    return total
```

Fitness evaluation uses city distance names function, which demands latitude and longitude data to determinate the real distance between points. For each point in the population, a fitness value will be generated.

```
# list of coordenades representing each city
lat = cities4_4.latitude
lon = cities4_4.longitude
coordinates_list = [[x,y] for x,y in list(zip(lat,lon))]
cities_dict = { x:y for x,y in zip(names_list,coordinates_list)}
```

```
# Function to compute the distance between two points
def compute_city_distance_names(city_a, city_b, cities_dict):
    return compute_city_distance_coordinates(cities_dict[city_a], cities_dict[city_b])

def compute_city_distance_coordinates(a,b):
    return ((a[0]-b[0])**2+(a[1]-b[1])**2)**0.5
```

Based on the computed fitness, the progenitors are the lowest fitness values, because of their probabilities to mate and produce “descendants”. The algorithm selects a set of progenitors based on their mating probability. The selection process is executed as many times as necessary until we obtain enough progenitors to produce N kids to replace the original set of N solutions.

Genetics algorithm has different selection methods, such as Roulette wheel selection, rank selection, tournament selection etc. On this project, the roulette wheel is used for selecting potentially useful solutions for recombination, based on fitness proportionate. So, the function calculates a crossover individual probability, using individual`s fitness divided by the sum of all population fitness.

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$$

Pi (prob_list) is the probability of each chromosome equals the chromosome frequency divided by the sum of all fitness. Consequently, the probability of selecting a potential mate depends on your fitness with respect to the rest.

```
#define progenitor list
def progenitor_selection(population_set,fitnes_list):
    total_fit = fitnes_list.sum()
    prob_list = fitnes_list/total_fit

    #Notice there is the chance that a progenitor. mates with oneself
    progenitor_list_a = np.random.choice(list(range(len(population_set))), len(population_set),p=prob_list, replace=True)
    progenitor_list_b = np.random.choice(list(range(len(population_set))), len(population_set),p=prob_list, replace=True)

    progenitor_list_a = population_set[progenitor_list_a]
    progenitor_list_b = population_set[progenitor_list_b]


    return np.array([progenitor_list_a,progenitor_list_b])
```

Two different progenitors list are generated and expected to mate. The mating consists of merging the two solutions into one (offspring) keeping bits of each of the parents (prog_a, prog_b). The (if not), checks if no repetitions exist in the offspring, otherwise, it will need to be corrected for the solution to satisfy the problem constraints.

```
#Crossover
def mate_progenitors(prog_a, prog_b):
    offspring = prog_a[0:5]

    for city in prog_b:

        if not city in offspring:
            offspring = np.concatenate((offspring,[city]))

    return offspring

def mate_population(progenitor_list):
    new_population_set = []
    for i in range(progenitor_list.shape[1]):
        prog_a, prog_b = progenitor_list[0][i], progenitor_list[1][i]
        offspring = mate_progenitors(prog_a, prog_b)
        new_population_set.append(offspring)

    return new_population_set
```

After the reproduction, a new population set, the algorithm has a mutate step, which is important to prevent local optima solutions. For this reason, randomness function will change different parts of the solution arbitrarily. Higher the variation of the progenitors, higher the chances of achieving a better solution, otherwise it can converge to all the solutions being the same. The *mutation_rate* parameter defines the percentage of progenitors that will be mutate for a new evaluation.

```
#Motation
def mutate_offspring(offspring):
    for q in range(int(n_cities*mutation_rate)):
        a = np.random.randint(0,n_cities)
        b = np.random.randint(0,n_cities)

        offspring[a], offspring[b] = offspring[b], offspring[a]

    return offspring

def mutate_population(new_population_set):
    mutated_pop = []
    for offspring in new_population_set:
        mutated_pop.append(mutate_offspring(offspring))
    return mutated_pop
```

Stopping criteria's of genetic algorithm are:

- The algorithm performed all the maximum iterations established
- There is a solution that satisfies the minimum route, because was found a fitness equal or better than we expected
- Overused resources as money, time, and computation.
- The algorithm seems unable to find something better and the current solution satisfies the requirements.

The goal is merging all this criteria's to find the best stopping rule for the optimization route.

```
for i in range(num_inter):

    #Saving the best solution
    if fitnes_list.min() < best_solution[1]:
        best_solution[0] = i
        best_solution[1] = fitnes_list.min()
        best_solution[2] = np.array(mutated_pop)[fitnes_list.min() == fitnes_list]

    progenitor_list = progenitor_selection(population_set,fitnes_list)
    new_population_set = mate_population(progenitor_list)

    mutated_pop = mutate_population(new_population_set)

#Best Result
print(best_solution)

#Time
genetic_teste_time = time.time() - start_time
print("--- %s seconds ---" % (genetic_teste_time))
```

5.5 Algorithms Implemented

TSP has been studied for a long time. We decided to use some already published algorithms as a guide for our project. The authors and original websites are:

- HILL-CLIMBING: published by A Name Not Yet Taken AB at https://www.annytab.com/ [15]
- SIMULATED ANNEALING: published by A Name Not Yet Taken AB at https://www.annytab.com/ [16]

- TABU SEARCH: published by Bartłomiej Jamiołkowski *aka* MindBreaker20 at https://github.com/MindBreaker20 [17]
- GENETIC ALGORITHMS: published by Roc Reguant at https://medium.com/@rocreguant [18]

Although we researched existing codes, profound changes had to be made in order to answer our questions and meet the requirements of our work.

6 Experimental Results

6.1 Datasets

The datasets used in the code execution are shown below. The indicated distances are in kilometers. Unfortunately, it is not possible to make the 37 x 37 dataset more readable, however all the information shown below is available in the notebook (Jupyter).

	Porto	Braga	Aveiro	Viana do Castelo
Porto	0	54	75	76
Braga	56	0	126	61
Aveiro	73	125	0	146
Viana do Castelo	75	61	146	0

Fig.6 – Test dataset 4 x 4

	Lisboa	Porto	Braga	Setubal	Coimbra	Evora	Aveiro	Leiria	Faro	ana do Caste	Beja	Braganca	Castelo Branco	Guarda
Lisboa	0	312	364	49	205	133	255	145	277	385	176	484	224	319
Porto	315	0	54	349	118	405	75	181	549	76	448	205	258	200
Braga	367	56	0	400	170	456	126	233	601	61	500	217	309	252
Setubal	50	347	399	0	240	99	290	180	244	420	143	519	259	354
Coimbra	207	117	169	240	0	242	60	73	441	191	340	283	140	152
Evora	133	402	453	99	294	0	345	235	226	475	80	462	195	290
Aveiro	256	73	125	290	59	346	0	122	490	146	389	268	199	158
Leiria	147	181	233	181	74	237	124	0	381	254	281	353	165	222
Faro	278	548	599	245	440	226	490	381	0	621	146	720	460	554
Viana do Cas	387	75	61	420	190	476	146	253	621	0	520	271	329	271
Beja	177	446	498	143	338	79	389	279	147	519	0	539	273	368
Braganca	511	206	216	545	283	461	271	378	745	272	541	0	272	177
Castelo Bran	227	255	307	260	140	195	198	164	461	329	274	271	0	99
Guarda	321	199	251	355	154	289	160	259	555	273	369	177	100	0

Fig.7 – Portugal cities and distances dataset 14 x 14

	Amsterda	Antwerp	Athens	Barcelona	Berlin	Bern	Brussels	Calais	Cologne	Copenhag	Edinburgh	Frankfurt	Geneva	Genoa	Hamburg	Le Havre	Lisbon	London	Luxembou
Amsterdam	0	160	3082	1639	649	875	209	385	280	904	1180	471	1014	1310	455	670	2300	494	371
Antwerp	160	0	2766	1465	723	704	46	211	237	861	1005	427	840	1136	557	453	2126	337	280
Athens	3082	2766	0	3312	2552	2627	3021	2976	2562	3414	3768	2382	2692	2242	2758	3394	4578	3099	2744
Barcelona	1639	1465	3312	0	1899	913	1419	1399	1539	2230	2181	1284	758	946	1856	1336	1266	1512	1137
Berlin	649	723	2552	1899	0	986	782	936	575	743	1727	570	1141	1188	291	1189	3165	1059	767
Bern	875	704	2627	913	986	0	655	854	583	1392	1643	424	155	448	906	767	2179	975	429
Brussels	209	46	3021	1419	782	655	0	212	219	1035	996	409	674	1090	586	407	2080	328	233
Calais	385	211	2976	1399	936	854	212	0	431	1075	792	621	820	1213	798	284	2060	123	414
Cologne	280	237	2562	1539	575	583	219	431	0	730	1206	190	765	1061	446	576	2294	538	195
Copenhagen	904	861	3414	2230	743	1392	1035	1075	730	0	1864	799	1531	1552	321	1531	3115	1196	1106
Edinburgh	1180	1005	3768	2181	1727	1643	996	792	1206	1864	0	1395	1536	1922	1555	1074	2879	656	1206
Frankfurt	471	427	2382	1284	570	424	409	621	190	799	1395	0	585	881	497	758	2544	727	249
Geneva	1014	840	2692	758	1141	155	674	820	765	1531	1536	585	0	568	1082	757	2024	867	484
Genoa	1310	1136	2242	946	1188	448	1090	1213	1061	1552	1922	881	568	0	1378	1233	2212	1253	887
Hamburg	455	557	2758	1856	291	906	586	798	446	321	1555	497	1082	1378	0	1082	2666	887	761
Le Havre	670	453	3494	1336	1189	767	407	284	576	1531	1074	758	757	1233	1082	0	1894	406	512
Lisbon	2300	2126	4578	1266	3165	2179	2080	2060	2294	3115	2879	2544	2024	2212	2666	1894	0	2210	2165
London	494	337	3099	1512	1059	975	328	123	538	1196	656	727	867	1253	887	406	2210	0	538
Luxembourg	371	280	2744	1137	767	429	233	414	195	1106	1206	249	484	887	761	512	2165	538	0
Lyon	995	821	2774	644	1289	317	671	755	830	1586	1552	640	162	541	1137	692	1784	884	493
Madrid	1782	1608	3940	628	2527	1541	1562	1542	1776	2597	2372	1906	1386	1574	2409	1376	638	1704	1647
Marseille	1323	1149	2997	515	1584	598	999	1083	1208	1914	1860	1004	443	431	1465	1020	1781	1192	816
Milan	1154	980	2280	1102	1168	347	934	1057	905	1671	1883	725	412	156	1222	1087	2368	1215	731
Munich	875	832	2210	1349	604	436	811	998	592	1204	1743	383	591	707	755	1038	2515	1075	583
Naples	2070	1894	2784	1704	1806	1261	1848	1971	1819	2585	2664	1639	1326	758	2136	1991	2970	1996	1645
Nice	1435	1261	2570	685	1610	638	1277	1195	1265	2014	2015	1085	483	204	1565	1132	1951	1347	1106
Paris	515	340	3140	1125	1094	556	294	274	508	1329	1082	592	546	1006	880	211	1786	414	379
Prague	973	870	2198	1679	354	766	911	1082	659	1033	1872	552	954	1007	1235	1305	2945	1204	746
Rome	1835	1660	2551	1471	1573	897	1615	1738	1586	2352	2467	1406	1093	525	1903	1758	2737	1799	1400
Rotterdam	80	100	2826	1565	697	802	146	311	254	813	1100	444	940	1236	486	553	2226	432	361
Strasbourg	683	544	2581	1072	801	232	488	627	402	1158	1412	212	371	667	709	667	2212	744	220
Stuttgart	703	659	2428	1263	636	350	641	792	395	1178	1534	205	505	688	729	832	2377	866	322
The Hague	56	139	3061	1589	712	825	170	352	283	960	1142	463	964	510	560	577	2250	473	390
Turin	1264	1090	2250	892	1172	312	1044	1110	1015	1527	1786	835	304	186	1332	1047	2158	1118	860
Venice	1449	1275	1995	1327	1108	642	1229	1352	1072	1708	2146	1020	707	381	1259	1382	2593	1478	1026
Vienna	1196	1180	1886	1989	666	907	1134	1346	915	1345	2098	725	1055	983	896	1496	3255	1233	885
Zurich	861	687	2449	1036	863	123	641	764	612	1378	1631	432	278	641	929	767	2302	963	438

	Lyon	Madrid	Marseille	Milan	Munich	Naples	Nice	Paris	Prague	Rome	Rotterdar	Strasbour	Stuttgart	The Hagu	Turin	Venice	Vienna	Zurich
Amsterdam	995	1782	1323	1154	875	2070	1435	515	973	1835	80	683	703	56	1264	1449	1196	861
Antwerp	821	1608	1149	980	832	1894	1261	340	870	1660	100	544	659	139	1090	1275	1180	687
Athens	2774	3940	2997	2281	2210	2784	2570	3140	2198	2551	2826	2581	2428	3061	2250	1995	1886	2449
Barcelona	644	628	515	1102	1349	1704	685	1125	1679	1471	1565	1072	1263	1589	892	1327	1989	1036
Berlin	1289	2527	1584	1168	604	1806	1610	1094	354	1573	697	801	636	712	1172	1108	666	863
Bern	317	1541	598	347	436	1261	638	556	766	897	802	232	350	825	312	642	907	123
Brussels	671	1562	999	934	811	1848	1277	294	911	1615	146	488	641	170	1044	1229	1134	641
Calais	755	1542	1083	1057	998	1971	1185	274	1082	1738	311	627	792	352	1110	1352	1346	764
Cologne	830	1776	1208	905	592	1819	1265	508	659	1586	254	402	395	283	1015	1072	915	612
Copenhagen	1586	2597	1914	1671	1204	2585	2014	1329	1033	2352	813	1158	1178	960	1527	1708	1345	1378
Edinburgh	1552	2372	1860	1883	1743	2664	2015	1082	1872	2467	1100	1412	1534	1142	1786	2146	2098	1631
Frankfurt	640	1906	1004	725	383	1639	1085	592	552	1406	444	212	205	463	835	1020	725	432
Geneva	162	1386	443	412	591	1326	483	546	954	1093	940	371	505	964	304	707	1055	278
Genoa	541	1574	431	156	707	758	204	1006	1007	525	1236	667	688	510	186	381	983	461
Hamburg	1137	2409	1465	1222	755	2136	1565	880	1235	1903	486	709	729	560	1332	1259	896	929
Le Havre	692	1376	1020	1087	1038	1991	1132	211	1305	1758	553	667	832	577	1047	1382	1496	767
Lisbon	1784	638	1781	2368	2515	2970	1951	1786	2945	2737	2226	2212	2377	2250	2158	2593	3255	2302
London	884	1704	1192	1215	1075	1996	1347	414	1204	1799	432	744	866	473	1118	1478	1233	963
Luxembourg	493	1647	816	731	583	1645	1106	379	746	1400	361	220	322	390	860	1026	885	438
Lyon	0	1272	328	494	753	1299	440	481	1116	1066	921	428	667	945	355	789	1217	404
Madrid	1272	0	1143	1730	1877	2332	1313	1268	2307	2099	1708	1700	1891	1732	1520	1955	2617	1664
Marseille	328	1143	0	587	1034	1189	227	809	1397	956	1249	814	948	1273	434	812	1414	721
Milan	494	1730	587	0	551	914	360	850	875	681	1080	511	532	1104	139	295	887	305
Munich	753	1877	1034	551	0	1202	911	827	363	969	827	371	218	870	691	504	458	313
Naples	1299	2332	1189	914	1202	0	962	1764	1603	233	1994	1425	1446	2018	934	799	1401	1219
Nice	440	1313	227	360	911	962	0	921	1294	729	1361	868	892	1385	207	585	1187	665
Paris	481	1268	809	850	827	1764	921	0	1094	1531	440	456	621	464	836	1145	1285	557
Prague	1116	2307	1397	875	363	1603	1274	1094	0	1370	858	638	773	1028	991	798	312	676
Rome	1066	2099	856	681	969	233	729	1531	1370	0	1761	1192	1213	1785	701	566	1168	986
Rotterdam	921	1708	1249	1080	827	1994	1361	440	858	1761	0	644	676	47	1190	1335	1169	787
Strasbourg	428	1700	814	511	371	1425	868	456	638	1192	644	0	165	668	621	806	829	218
Stuttgart	667	1891	948	532	218	1446	892	621	773	1213	676	165	0	673	642	680	676	227
The Hague	945	1732	1273	1104	870	2018	1385	464	1028	1785	47	668	673	0	1167	1399	1193	811
Turin	355	1520	434	139	691	934	207	836	991	701	1190	621	642	1167	0	434	1026	444
Venice	789	1955	812	295	504	799	585	1145	798	566	1335	806	680	1399	434	0	610	600
Vienna	1217	2617	1414	887	458	1401	1187	1285	312	1168	1169	829	676	1193	1026	610	0	784
Zurich	404	1664	721	305	313	1219	665	557	676	986	787	218	227	811	444	600	784	0

Fig.8 – European cities and distances dataset 37 x 37

After executing the proposed algorithms, we performed the comparison between the optimal routes obtained and the respective execution times of each one.

6.2 Experimental Results

Below are the tables with the results obtained using 500, 1.000, 5.000 and 10.000 iterations for each algorithm, took on March 30th 2022. The values may be slightly different from those shown on the notebook as there are small changes with each run.

Best routes:

500 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	34943.000000	2403.000000	334.000000
Hill-Climbing	20554.000000	1996.000000	334.000000
Simulated Annealing	21148.000000	2684.000000	334.000000
Tabu Search	22193.000000	1814.000000	334.000000

1.000 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	35131.000000	2334.000000	338.000000
Hill-Climbing	20554.000000	2160.000000	334.000000
Simulated Annealing	21532.000000	2710.000000	336.000000
Tabu Search	21067.000000	1928.000000	334.000000

5.000 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	35433.000000	2408.000000	338.000000
Hill-Climbing	21209.000000	2160.000000	334.000000
Simulated Annealing	22903.000000	2146.000000	334.000000
Tabu Search	19583.000000	2149.000000	334.000000

10.000 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	31715.000000	2715.000000	338.000000
Hill-Climbing	21433.000000	1853.000000	334.000000
Simulated Annealing	21983.000000	2219.000000	334.000000
Tabu Search	17339.000000	1695.000000	334.000000

Fig.9 – Best route in Km. In all runs (~30x). Best results highlighted in green.

Best run times:

500 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	2.061837	0.823812	0.254758
Hill-Climbing	0.056075	0.026979	0.007990
Simulated Annealing	0.050189	0.020968	0.085912
Tabu Search	0.050417	0.067387	0.005980

1.000 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	4.460856	1.727222	0.529009
Hill-Climbing	0.104893	0.033966	0.021976
Simulated Annealing	0.103894	0.123472	0.043954
Tabu Search	0.075913	0.062918	0.009981

5.000 iterations

Example Algorithm	Europe	Portugal	Teste
Genetic	21.023144	7.705487	2.349869
Hill-Climbing	0.447905	0.215780	0.134584
Simulated Annealing	0.435094	0.255421	0.245766
Tabu Search	0.218802	0.102895	0.046951

10.000 iterations

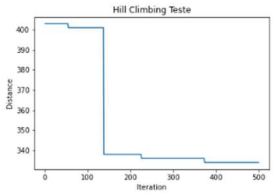
Example Algorithm	Europe	Portugal	Teste
Genetic	43.742670	15.833771	4.824254
Hill-Climbing	0.969232	0.664925	0.397601
Simulated Annealing	0.986990	0.660251	0.513638
Tabu Search	0.277762	0.161844	0.091907

Fig.10 – Time of execution in seconds. In all runs (~30x). Best results highlighted in green.

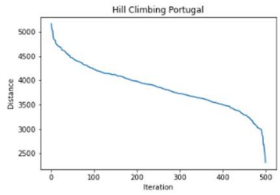
6.3 Iterations x Route Graphs (plus Temperature for Sim. Annealing only)

Hill Climbing

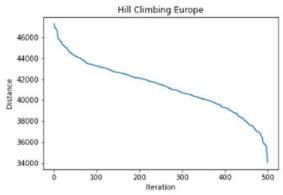
4 x 4 – 500 iterations



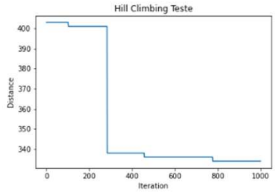
14 x 14 – 500 iterations



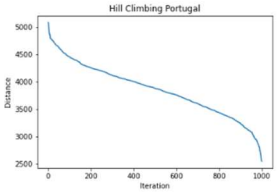
37 x 37 – 500 iterations



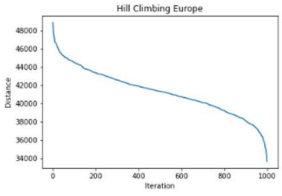
4 x 4 – 1.000 iterations



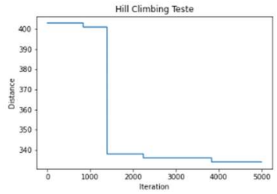
14 x 14 – 1.000 iterations



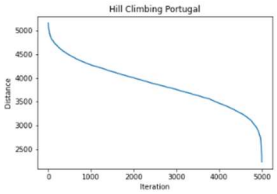
37 x 37 – 1.000 iterations



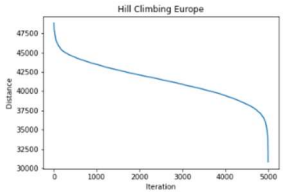
4 x 4 – 5.000 iterations



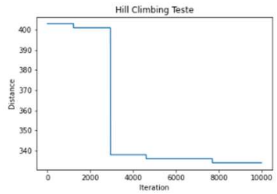
14 x 14 – 5.000 iterations



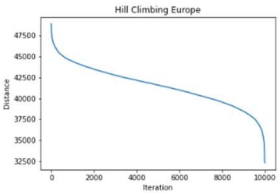
37 x 37 – 5.000 iterations



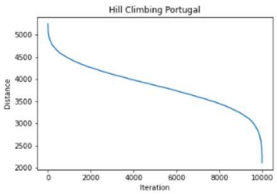
4 x 4 – 10.000 iterations



14 x 14 – 10.000 iterations

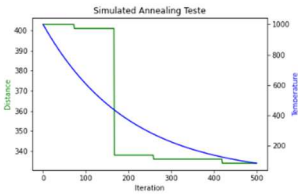


37 x 37 – 10.000 iterations

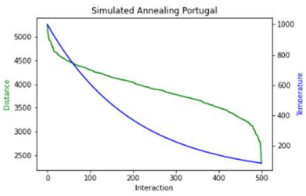


Simulated Annealing

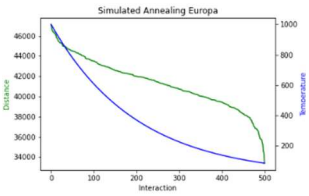
4 x 4 – 500 iterations



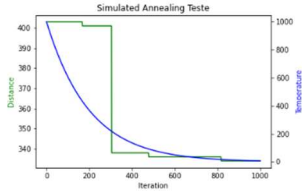
14 x 14 – 500 iterations



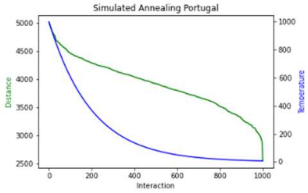
37 x 37 – 500 iterations



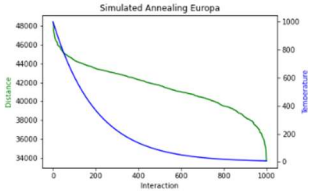
4 x 4 – 1.000 iterations



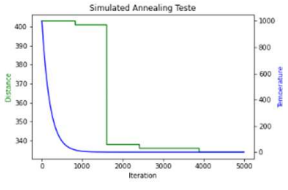
14 x 14 – 1.000 iterations



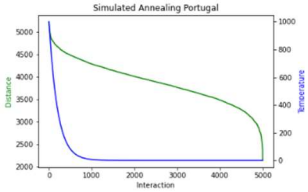
37 x 37 – 1.000 iterations



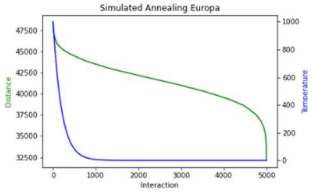
4 x 4 – 5.000 iterations



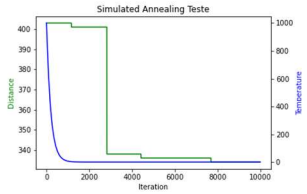
14 x 14 – 5.000 iterations



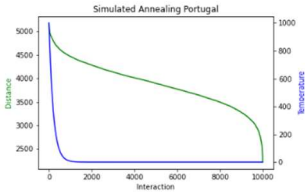
37 x 37 – 5.000 iterations



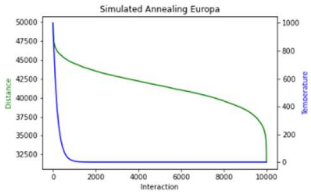
4 x 4 – 10.000 iterations



14 x 14 – 10.000 iterations

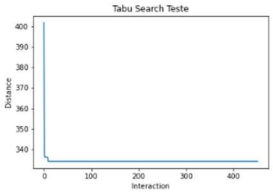


37 x 37 – 10.000 iterations

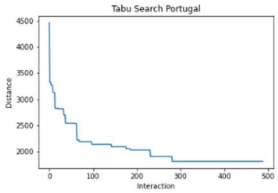


Tabu Search

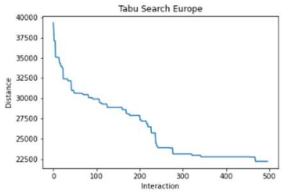
4 x 4 – 500 iterations



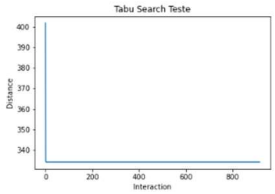
14 x 14 – 500 iterations



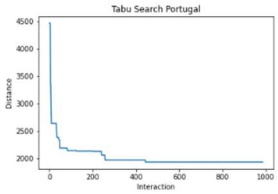
37 x 37 – 500 iterations



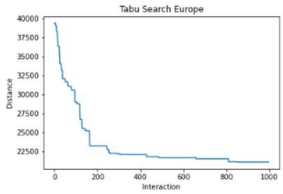
4 x 4 – 1.000 iterations



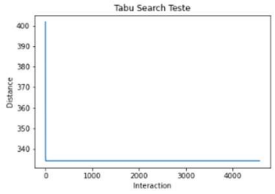
14 x 14 – 1.000 iterations



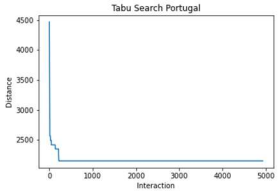
37 x 37 – 1.000 iterations



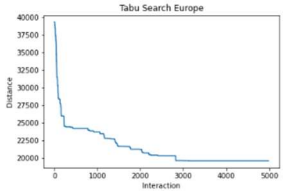
4 x 4 – 5.000 iterations



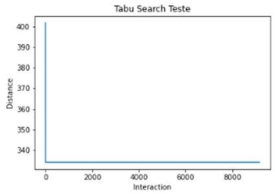
14 x 14 – 5.000 iterations



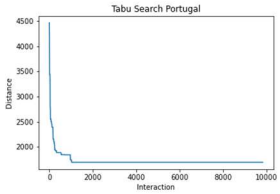
37 x 37 – 5.000 iterations



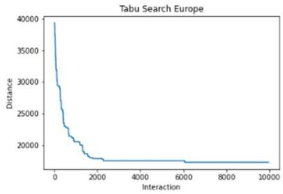
4 x 4 – 10.000 iterations



14 x 14 – 10.000 iterations

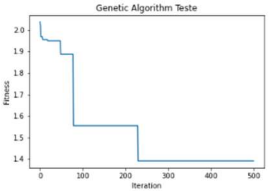


37 x 37 – 10.000 iterations

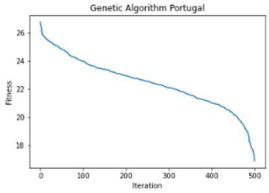


Genetic Algorithm

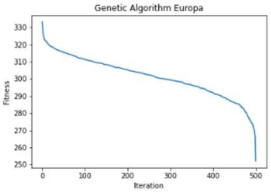
4 x 4 – 500 iterations



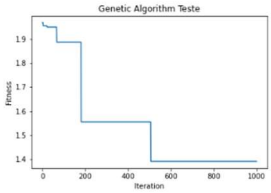
14 x 14 – 500 iterations



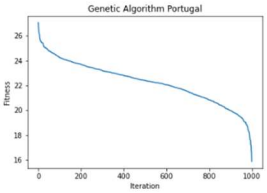
37 x 37 – 500 iterations



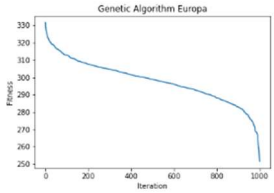
4 x 4 – 1.000 iterations



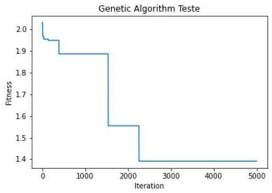
14 x 14 – 1.000 iterations



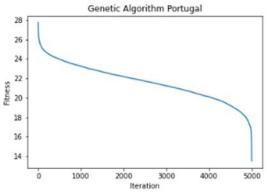
37 x 37 – 1.000 iterations



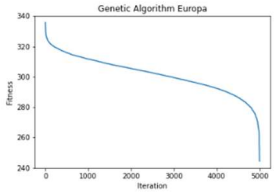
4 x 4 – 5.000 iterations



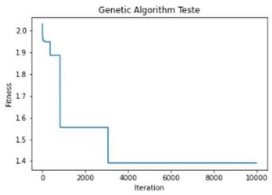
14 x 14 – 5.000 iterations



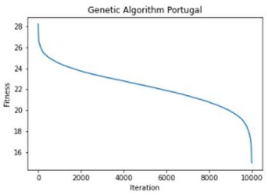
37 x 37 – 5.000 iterations



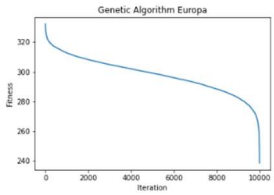
4 x 4 – 10.000 iterations



14 x 14 – 10.000 iterations



37 x 37 – 10.000 iterations



6.4 Ilustrative Route Maps

These routes maps were obtained as a result of each simulation with best results collected at Fig. 9. We can see the many differences in the "interpretation" of each algorithm to draw the route.

HC 4 x 4 1000 it
334 Km

HC 14 x 14 1000 it
2160 Km

Hill Climbing 37 x 37 1000 it
20554 Km

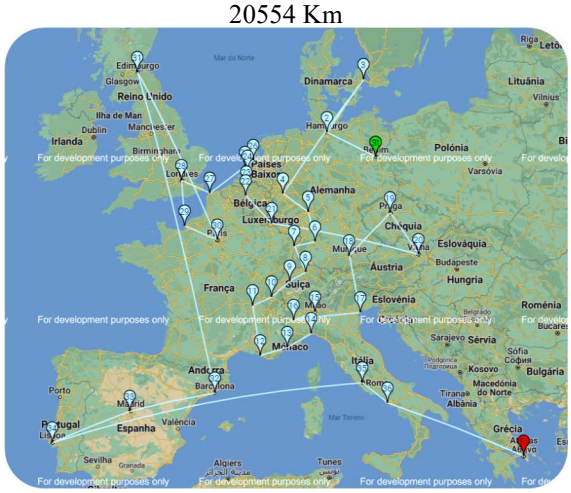
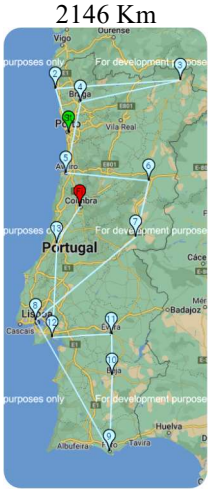


Fig.11 – Hill-Climbing route map after 1.000 iteraction

SA 4 x 4 5000 it
334 Km

SA 14 x 14 50000 it
2146 Km

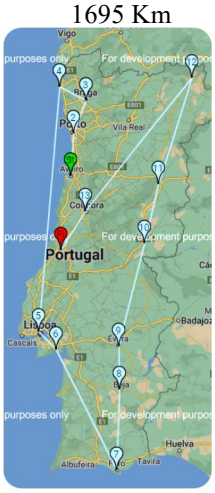


Simulated Annealing 37 x 37 5000 it
22903 Km

Fig.12 – Simulated Annealing route map after 5.000 iterations

TS 4 x 4 10000 it
334 Km

TS 14 x 14 100000 it
1695 Km

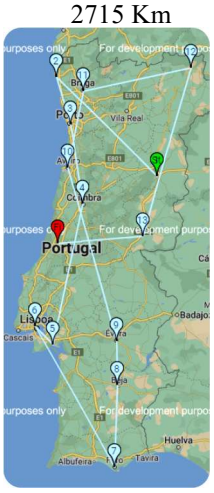


Tabu Search 37 x 37 10000 it
17339 Km

Fig.13 – Tabu Search route map after 10.000 iterations

GA 4 x 4 10000 it
338 Km

GA 14 x 14 10000 it
2715 Km



Genetic Algorithm 37 x 37 10000 it
31715 Km

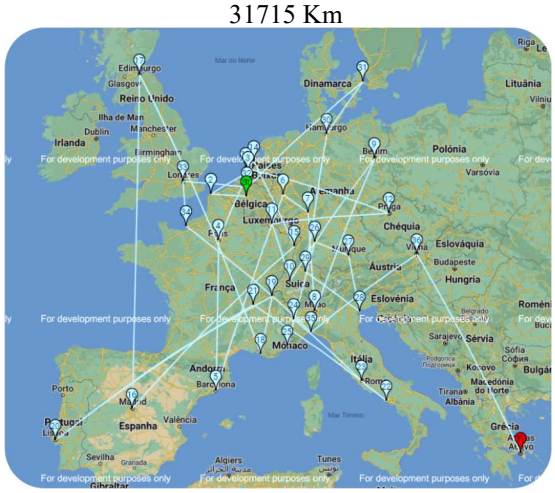


Fig.14 – Genetic Algorithm route map after 10.000 iterations

7 Conclusion

The results were obtained with the following algorithms parameters specification:

- Hill Climbing:
  - Iterations: 500, 1.000, 5.000 and 10.000.
  - Neighborhood: random swap among two neighbors.
  - Mutation rate: mutation_rate = 0.7 (discarding 30% neighbourhood).

- Simulated Annealing:
  - Iterations: 500, 1.000, 5.000 and 10.000.
  - Neighborhood: random swap among two neighbors.
  - Temperature: Initial 1000º, final 0º.
  - Mutation rate: mutation_rate = 0.7 (discarding 30% neighbourhood).
- Tabu Search:
  - Iterations: 500, 1.000, 5.000 and 10.000.
  - Tabu tenure:
    - Tabu len test = 2 iterations
    - Tabu len Portugal = 4 iterations
    - Tabu len Europe = 6 iterations
  - Neighborhood: set 1 for test, Portugal, and Europe.
- Genetic Algorithm:
  - Iterations: 500, 1.000, 5.000 and 10.000
  - n_cities: length of cities in each matrix (4, 14 and 37)
  - n_population: 50. number of populations that will be generated in genesis function by the initial progenitor
  - mutation_rate: 0.3. percentage of progenitors that will be mutate for a new evaluation.

As shown in Fig. 9 – Page 22, the all the algorithms presented the best results (334 Km) with 500 iterations to the test dataset. We also can see that:

- The best route for Portugal cities were found for Tabu Search with 10.000 iterations (1.695 Km). It's 6,5 % better than the best result with 500 iterations and to get this 6,5% improvement, we need to use 140% more time
- The best route for Europe cities were also found for Tabu Search with 10.000 iterations (17.339 Km). It's 15,6% better than the best result with 500 iterations and to get this 15,6% improvement, we need to use 395% more time.

	Portugal			Europe		
	Route (Km)	**Time (s)**	**Alg.**	**Route (Km)**	**Time (s)**	**Alg.**
FBR 500 it	1814	0,067387	Tabu S.	20554	0,056075	Hill Cli.
BR 10.000 it	1695	0,161844	Tabu S.	17339	0,277762	Tabu S.
Variation	**-6,56%**	**140,17%**		**-15,64%**	**395,34%**	

FBR = First Best Route, BR = Best Route

It is clear that these times are very small, less than 1 second, however, for future work with larger universes of cities or locations, the impact of this increase in processing time must be evaluated. Complete time processing at Fig. 10 – Page 23.

The Fig. 9 and 10 show the cost-benefit ratio of increasing the number of iterations with fixed algorithm parameters to get their best solution route. Most of the algorithms performed their best local solution in less than one second, except for the genetic, which

was much slower. Therefore, as most of the output processing time were fast, the individual parameters of each algorithm were analyzed and modified with the aim of searching for new possible better routes with less time in the European dataset and 10.000 iterations.

- Hill Climbing:
  - Iterations: 10.000.
  - Neighborhood: random swap among two neighbors.
  - Mutation rate: mutation_rate = 1 (no discarding neighb.).
- Simulated Annealing:
  - Iterations: 10.000.
  - Neighborhood: random swap among two neighbors.
  - Temperature: Initial 1000º, final 1º.
  - Mutation rate: mutation_rate = 1 (no discarding neighb.).
- Tabu Search:
  - Iterations: 10.000.
  - Tabu tenure:
    - Tabu len Europe = 2 iterations (keeping Tabu for 2 it.).
  - Neighborhood: 2 (generate neighbors two times swapping cities).
- Genetic Algorithm:
  - Iterations: 10.000.
  - n_cities: 37.
  - n_population: 100. Number of populations that will be generated in genesis function by the initial progenitor.
  - mutation_rate: 0,4. Percentage of progenitors that will be mutate for a new evaluation, avoiding local maxima.

There was no improvement in the results already obtained, as can be seen in the figure below:

Example Algorithm	Europe
Genetic	32668.000000
Hill-Climbing	21615.000000
Simulated Annealing	22132.000000
Tabu Search	19175.000000

Example Algorithm	Europe
Genetic	828.686554
Hill-Climbing	1.720625
Simulated Annealing	0.216883
Tabu Search	0.725888

Fig.15 – Route and time processing for Europe dataset, modifying parameters.

Finally, we performed a complete round with 1,000,000 iterations, keeping the parameters of the initial test (pages 28-29). Still, we didn't get better results.

Example Algorithm	Europe	Portugal	Teste
Genetic	33459.000000	2168.000000	338.000000
Hill-Climbing	21085.000000	1695.000000	334.000000
Simulated Annealing	22712.000000	1916.000000	334.000000
Tabu Search	19518.000000	1700.000000	334.000000

Example Algorithm	Europe	Portugal	Teste
Genetic	9038.901190	6849.501923	5361.153966
Hill-Climbing	8547.086568	8494.370667	8472.803038
Simulated Annealing	104.815753	102.602015	96.971790
Tabu Search	23.853625	12.248454	7.291541

Fig.16 – Route and time processing for all datasets, 1M iterations.

So, we concluded our work verifying that the best results were found using Tabu Search, with 10,000 interactions and with the following parameters:

- Tabu tenure:
  - Tabu len test = 2 iterations
  - Tabu len Portugal = 4 iterations
  - Tabu len Europe = 6 iterations
- Neighborhood: set 1 for test, Portugal, and Europe.

Example Algorithm	Europe	Portugal	Teste
Tabu Search	17339.000000	1695.000000	334.000000

Fig.17 – Best results. Run Time: 0.27s, 0,16s and 0,09s respectively.

Bibliographic References and Websites

1. Prof. Luis Paulo Reis. Practical Work 2021/22 at https://moodle.up.pt/mod/resource/view.php?id=141283
2. https://en.wikipedia.org/wiki/Travelling_salesman_problem
3. Mona Fronita, Rahmat Gernowo and Vincencius Gunawan. Comparison of Genetic Algorithm and Hill Climbing for Shortest Path Optimization Mapping, https://doi.org/10.1051/e3sconf/20183111017
4. Hein de Haan, Dec 8, 2020. https://towardsdatascience.com/how-to-implement-the-hill-climbing-algorithm-in-python-1c65c29469de
5. M. A. Rufai, r. M. Alabison, a. Abidemi and e.j. Dansu, Solution to The Travelling Salesperson Problem Using Simulated Annealing Algorithm. Electronic Journal of Mathematical Analysis and Applications Vol. 5(1) Jan. 2017, pp. 135-142. ISSN: 2090-729 (online) http://fcag-egypt.com/Journals/EJMAA/
6. Zhou, Ai-Hua & Zhu, Li-Peng & Hu, Bin & Deng, Song & Song, Yan & Qiu, Hongbin & Pan, Sen. (2018). Traveling-Salesman-Problem Algorithm Based on Simulated Annealing and Gene-Expression Programming. Information. 10. 7. 10.3390/info10010007.
7. E.Gangadevi – Feb. 2018. Implementing Tabu Search on Traveling Salesman Problem. http://www.ijarse.com/images/fullpdf/1519813763_NMCOE4098IJARSE.pdf
8. Isra Natheer Alkallak and Ruqaya Zedan Sha'ban. Tabu Search Method for Solving the Traveling salesman Problem. Raf. J. of Comp. & Math's. , Vol. 5, No. 2, 2008.
9. Dhawal Thakkar, March 17, 2021. https://crescointl.com/2021/03/17/using-a-genetic-algorithm-for-traveling-salesman-problem-in-python/
10. Research Paper on Travelling Salesman Problem and it's Solution Using Genetic Algorithm, Kartik Rai, Lokesh Madan and Kislay Anand. https://ijirt.org/master/publishedpaper/IJIRT101672_PAPER.pdf.
11. Prof. Luis Paulo Reis, Artificial Intelligence 2020/2021, Lecture 3a: Optimization and Local Search (based on Løkketangen, 2019), slide 13.
12. https://pt.melhoresrotas.com/tabela-de-distancias-entre-cidades/pt/
13. https://www.engineeringtoolbox.com/driving-distances-d_1029.html
14. https://simplemaps.com/data/world-cities
15. A Name Not Yet Taken AB at https://www.annytab.com/hill-climbing-search-algorithm-in-python/
16. A Name Not Yet Taken AB at https://www.annytab.com/simulated-annealing-search-algorithm-in-python/
17. Bartłomiej Jamiołkowski aka MindBreaker20 at https://github.com/MindBreaker20/Traveling-Salesman-Problem/blob/main/tabu_search.py
18. Roc Reguant in How to Implement a Traveling Salesman Problem Genetic Algorithm in Python at https://levelup.gitconnected.com/how-to-implement-a-traveling-salesman-problem-genetic-algorithm-in-python-ea32c7bef20f
19. https://www.researchgate.net/publication/331585233_Tabu_Search_Method_for_Solving_the_Traveling_salesman_Problem