2023

Using Cplex and Meta-Heuristic Algorithm for TSP

COMPARISON OF THE 2 METHODS

MARCO NARDELOTTO 2044588

Sommario

Introduction

The document presents a combinatorial optimization problem that involves minimizing the total drilling time for a company that produces boards with holes used to build electric panels. To solve this problem, 2 different methods were used, which in this report are divided into 2 parts, the results of which will be compared in this report.

In the first part is proposes an Integer Linear Programming (ILP) model based on a graph representation of the problem, where the nodes correspond to the positions where holes need to be made and the arcs represent the trajectory of the drill moving from one hole to another. The problem can be viewed as finding the minimum weight Hamiltonian cycle on the graph, which is a Travelling Salesman Problem (TSP). it is suggested a compact formulation for the TSP based on network flows, and we aim to implement this model using the Cplex API. The goal is to determine the ability of the model to provide exact solutions for the problem and to test it for different sizes (i.e., number of holes) to see how long it takes to solve the problem within different time constraints.

In the second part a meta-heuristic algorithm has been implemented, to define if the hole production problem can also be solved using an inexact algorithm, but which still leads to a good but not optimal solution in many cases.

Testing Environment

The development and test environments were of two different types:

- **Domestic**: in the home tests a virtual machine with Ubuntu operating system, 2048 MB of dedicated ram and 3 dedicated processors was used.
- **University**: in the university laboratories the tests were carried out on machines with the Ubuntu operating system, 8GB of dedicated ram and 4 dedicated processors.

For both environments I tried to carry out tests for each part of the exercise to also compare the times obtained on different machines.

Cplex studio version

For the implementation of the initial analysis, the CPLEX Studio program version 221 was used, the tests were then carried out on the home computer with version 221 and in the university laboratory where version 128 is present.

First part

Model

The mathematical model implemented is the one defined in the specification of the exercise and is represented below:

Sets:

- N = graph nodes, representing the holes.
- A = arcs (i, j), $\forall$ i, j $\in$ N, representing the trajectory covered by the drill to move from hole i to hole j.

Parameters:

- C_{ij} = time taken by the drill to move from i to j, $\forall$ (i, j) $\in$ A;

Decision Variables:

- X_{ij} = amount of the flow shipped from i to j, $\forall$ (i, j) $\in$ A.

- Y_{ij} = 1 if arc (i, j) ships some flow, 0 otherwise, ∀ (i, j) ∈ A.

$$\min \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} \quad (1)$$

$$s.t. \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \quad \forall\, k \in N \setminus \{0\} \quad (2)$$

$$\sum_{j:(i,j)\in A} y_{ij} = 1 \quad \forall\, i \in N \quad (3)$$

$$\sum_{i:(i,j)\in A} y_{ij} = 1 \quad \forall\, j \in N \quad (4)$$

$$x_{ij} \leq (|N| - 1) y_{ij} \quad \forall\, (i,j) \in A, j \neq 0 \quad (5)$$

$$x_{ij} \in \mathbb{R}_+ \, \forall\, (i,j) \in A, j \neq 0 \quad (6)$$

$$y_{ij} \in \{0,1\} \, \forall\, (i,j) \in A \quad (7)$$

The mathematical model presented is a formulation of the Traveling Salesman Problem (TSP) as a network flow model. This model is easily convertible with our case study as the cities in the TSP problem can be replaced by the holes to be created in our case, and the arcs of the distances between the cities are seen as the arcs for the distance between a hole and the other.

In this model, the holes are represented as nodes in a graph, and the arcs between the nodes represent the possible trajectories between holes. The model aims to find the minimum total time (in seconds) traveled by the drill by selecting a minimum cost subset of the arcs.

The model consists of several sets and parameters. The set N represents the nodes in the graph, which correspond to the holes in the problem. The set A represents the arcs between the nodes, which represent the possible trajectories between the holes. The parameter c_{ij} represents the cost or distance associated with traveling along arc (i, j), for all (i, j) in A.

The model also includes several decision variables. The variable x_{ij} represents the amount of flow shipped from node i to node j, for all (i, j) in A. The variable y_{ij} is a binary variable that takes on the value of 1 if arc (i, j) is included in the minimum cost solution, and 0 otherwise, for all (i, j) in A.

The model seeks to minimize the total cost of the solution, as represented by the objective function, which is the sum of the costs of all arcs that are included in the solution. The model also includes several constraints to ensure that the solution is feasible. Constraint (2) ensures that each node, except for the starting node, receives exactly one unit of flow and is visited only once by requiring that the total flow into each node, represented by the sum of xik for all (i, k) in A, minus the total flow out of each node, represented by the sum of xkj for all (k, j) in A where j is not equal to 0, is equal to 1 for all nodes k in N except for the starting node. Constraints (3) and (4) ensure that each node sends and receives at least one unit of flow, respectively, by requiring that the sum of yij for all (i, j) in A is equal to 1 for all nodes i and j. Constraint (5) limits the amount of flow that can be shipped along each arc to the amount of flow available at the starting node by requiring that xij is less than or equal to (|N| - 1)yij for all (i, j) in A where j is not equal to 0. Finally, constraints (6) and (7) ensure that the decision variables xij and yij are non-negative and binary, respectively, for all (i, j) in A where j is not equal to 0.

Implementation

The described model has been implemented using the C++ language using the available Cplex library. The implemented code has been divided into two header .h files and a main.cpp, the use of the file and its functions are defined below.

TSPFileContainer.h

This file contains the functionality to read a data file and store the initial data read from the file, for the problem described above. The file has the following structure:

Variables:

- int numberNodes: defines the number of nodes in the dataset.
- std::vector<std::vector<double>> values: it is a matrix that contains the travel times of the drill from one node to all the others.

Methods:

- void read(const std::string filename): the method that allows you to read a dataset (in the format defined in the Test Format section), in order to store the problem data.
- std::vector<std::string> split(const std::string& str, const char delim): auxiliary function that allows to divide a string defined by a limiter, the method then returns a vector containing the detected substrings.
- std::string trim(const std::string& str): auxiliary function that allows you to remove spaces from a string, returns the new string without spaces.

TSPData.h

The file allows you to divide the data of the problem obtained previously, so that they conform to the problem to be addressed and that define the parameters and sets necessary for the mathematical model described above. The file has the following structure:

Variables:

- const int N: defines the number of nodes in the dataset.
- const int A: defines the number of arcs in the dataset.
- std::string *nameN: defines the names of the nodes.
- double *C : defines the costs of each arc.

Methods:

- TSPData(int n, std::vector<std::vector<double>> values): the constructor of the class, which allows you to instantiate all the data starting from the numbered of nodes and the cost of each arc.

Main.cpp

This file contains all the code that uses the Cplex library, and which processes the data read to then carry out the operations of the mathematical model and generate the data for the solution of the problem. The execution time tests are also carried out in this file of each instance of the dataset. The comments in the file are used to better understand its usage, so we only write the main features in this report. In the file the instances of the two files previously described are used, TSPFileContainer to have an instance that contains the data initially detected by reading the test file, and TSPData to have a more precise formatting of the data, suited to the mathematical model implemented. Once all the data has been obtained, the main proceeds to iterate the problem-solving process for a number of iterations equal to the value of the constant variable TOTAL_REPETITION, at each iteration the execution time of the problem-solving process

is calculated and at the end of the total iterations, an average of the execution time is made. the main code (without comments) is visible in Fig. 1.

```
int main (int argc, char const *argv[])
{
  double objval; // Will contain the value for the objective function
  try{
    // Need filename parameter
    if (argc < 2) throw std::runtime_error("usage: ./main filename.dat");
    TSPFileContainer tspFileContainer;
    tspFileContainer.read(argv[1]); // Read data from file
    TSPData dataTSP(tspFileContainer.numberNodes, tspFileContainer.values);

    double totTime = 0;
    // Start benchmark
    for(int r = 0; r <TOTAL_REPETITIONS; r++){
      auto start = chrono::high_resolution_clock::now();
      ios_base::sync_with_stdio(false);
      try
      {
        DECL_ENV( env );
        DECL_PROB( env, lp );
        int numVars;
        setupLP(env, lp, numVars, dataTSP.N, dataTSP.A, dataTSP.nameN, dataTSP.C);
        CHECKED_CPX_CALL(CPXchgobjsen, env, lp, CPX_MIN);
        CHECKED_CPX_CALL( CPXmipopt, env, lp );
        CHECKED_CPX_CALL( CPXgetobjval, env, lp, &objval );
        int n = CPXgetnumcols(env, lp);
        std::vector<double> varVals;
        varVals.resize(n);
        CHECKED_CPX_CALL( CPXgetx, env, lp, &varVals[0], 0, n - 1 );
        CHECKED_CPX_CALL( CPXsolwrite, env, lp, "boardmaker.sol" );
        CPXfreeprob(env, &lp);
        CPXcloseCPLEX(&env);
      }
      catch(std::exception& e)
      {
          std::cout << ">>>EXCEPTION: " << e.what() << std::endl;
      }
      auto end = chrono::high_resolution_clock::now();
      double time_taken = chrono::duration_cast<chrono::nanoseconds>(end - start).count();

      time_taken *= 1e-9; // Round the time

      totTime += time_taken;
    }
    totTime /= TOTAL_REPETITIONS;

    std::cout<<"Total time: "<<totTime<<setprecision(9)<<endl;
    std::cout << "Objval: " << objval << std::endl;

    return 0;
  }catch(std::exception& e2)
  {
    std::cout << ">>>EXCEPTION: " << e2.what() << std::endl;
  }
}
```

Fig. 1 - Main code

In the try-catch inside the for loop, all the operations necessary to solve the problem with the mathematical model are visible, the actions performed are:

1. **DECL_ENV**: allows to declare the environment for CPLEX.
2. **DECL_PROB**: allows to declare the problem in the environment for CPLEX.
3. **setupLP**: function described in the next section, where the formalization of the model takes place using the data of the problem.
4. **CHECKD_CPX_CALL**: allows to call the functions of the CPLEX library, in particular the functions are called for:
    a. set the object function as a minimum function.
    b. Use optimizer for CPLEX and start the solving.
    c. Get the value of the calculated object function.
    d. Get the value of the decision variables.
    e. Write the data of the solved problem to a file (boardmaker.sol).
5. **CPXfreeprob**: allows to clean up the problem of the CPLEX environment.
6. **CPXcloseCPLEX**: Allows to close the created CPLEX environment.

These operations are the ones taken into consideration for calculating the execution time for solving the problem.

setupLP

This section describes the setupLP function, which contains the logic to define the mathematical model based on the problem data, creating the variables needed to solve them and defining the constraints as in the model.

The function receives as input:

- CEnv env: the environment for CPLEX.
- Prob lp: the problem space for the model
- int & numVars: number of variables, that will be incremented.
- const int& N: number of nodes for the problem.
- const int& A: number of arcs for the problem.
- string nameN[]: vector with the name for the nodes.
- double C[]: vector for the time constraint of the problem

```
/**
  Adding x vars to the problem:
  x_ij in R_+  forall (i,j) in A,j != 0
*/
/**
* Iterate throught all the position in the matrix of x's variables
* and add each Xij variablec
*/
for (int i = 0; i < N; i++)
{
  for(int j = 1; j < N; j++){ // x vars of type Xn0 are not considered
    if(j == i) continue; // Don't add variables of type: X11, X22, ... Xnn

    char htype = 'C'; // Define continuous variables
    double obj = 0.0; // Coefficient in the objective function (0 for x vars)
    double lb = 0.0; // Lower bound of the variables
    double ub = CPX_INFBOUND; // Upper bound of the variables
    snprintf(name, NAME_SIZE, "x_(%d,%d)", nameN[i], nameN[j]);
    char* hname = (char*)(&name[0]); // Name to identify the vars
    CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &htype, &hname ); // Create the variable in the problem environment

    // Map the position of the variable created in the matrix
    map_x[i][j] = cur_var_pos;
    cur_var_pos++;
  }
}
```

Fig. 2 - Add X vars logic

In the Fig. 2 is the code that allows you to create the number of variables of type x_{ij}, in this case all the variables are created which except for those with equal i and j values and excluding the value 0 for the j. In particular the variables are initialized as continuous ('C') and have as lower bound 0.0 and upper bound +infinity.

```
/**
    Adding y vars to the problem:
    y_ij in {0,1}  forall (i,j) in A
*/
int c_index = 0;
for (int i = 0; i < N; i++)
{
  for(int j = 0; j<N; j++){

    if(j == i) continue; // // Don't add variables of type: X11, X22, ... Xnn

    char htype = 'B'; // Define binary variables (0,1)
    double obj = C[c_index]; // Define the coefficient in the objective function (in this case the value in C parameters)
    double lb = 0.0; // Set lower bound
    double ub = 1.0; // Set upper bound
    snprintf(name, NAME_SIZE, "y_(%d,%d)", nameN[i], nameN[j]);
    char* hname = (char*)(&name[0]); // Name to identify the vars

    CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &htype, &hname ); // Add y var into problem environment

    c_index++;
    // Map the y var added in the matrix
    map_y[i][j] = cur_var_pos;
    cur_var_pos++;
  }
}
```

Fig. 3 - Add Y vars logic

In the Fig. 3 is the code that allows you to create the number of variables of type y_{ij}, in this case all the variables are created which except for those with equal i and j values. In particular the variables are initialized as Boolean or binary ('B') and have as lower bound 0.0 and upper bound 1.0.

The variables x and y when created are immediately mapped using the method described in the section Implementation Choices.

Subsequently, the number of variables created is saved using the following line of code:

```
numVars = CPXgetnumcols(env, lp);
```

```
/**
  Adding flow constraint:
  [ forall k in N \ {0}, sum{i : (i,k) in A} x_ik - sum{j: (k,j) j != 0} = 1 ]
*/
double one = 1.0; // var to indentify value 1
for (int k = 1; k < N; k++) // Exclude k = 0
{
  std::vector<int> idx;
  std::vector<double> coef;
  // Get x vars from X0k to Xn-1k to and set coefficient of each x vars to 1
  for(int i = 0; i < N; i++){
    if(map_x[i][k] < 0) continue; // Avoid variables not mapped
    idx.push_back(map_x[i][k]);
    coef.push_back(1.0);
  }
  // Get x vars from Xk1 to Xkn-1 to and set coefficient of each x vars to 1
  for(int j = 1; j < N; j++){
    if(map_x[k][j] < 0) continue; // Avoid variables not mapped
    idx.push_back(map_x[k][j]);
    coef.push_back(-1.0);
  }

  char sense = 'E'; // Define constraint of type equal (=)
  int matbeg = 0; // Define the beginning of the value in the matrix

  // Add row to the problem environment
  if(idx.size() != 0)
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &one, &sense, &matbeg, &idx[0], &coef[0], 0, 0 );
}
```

Fig. 4 - Adding constrains in (2)

To add all the constraints specified in line (2) of the formula, the code in Fig. 4 was used. Practically in the first for loop the first summation of the constriction is defined in which the indices of the variables involved are extracted using the mapping of the variables x, while the coeffients are all equal to 1 in this case. In the second for loop the second summation is defined in which the indices of the variables involved are extracted using the mapping of the x variables, while the coeffients are all at -1 in this case since they have to be subtracted from the first summation.

```
/*
  Adding yij ships inside constraints:
  [ forall i in N, sum{j:  (i,j) in A} y{i,j} = 1 ]
*/
for(int i  = 0; i < N; ++i){
  std::vector<int> idx;
  std::vector<double> coef;
  // Get index of all y vars from Yi0 to Yin-1 and set coefficient equal to 1
  for(int j = 0; j < N; j++){
    if(map_y[i][j]<0) continue; // Avoid variables not mapped

    idx.push_back(map_y[i][j]);
    coef.push_back(1.0);

  }
  char sense = 'E'; // Define constraint of type equal (=)
  int matbeg = 0; // Define the beginning of the value in the matrix

  // Add row to the problem environment
  if(idx.size() != 0)
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &one, &sense, &matbeg, &idx[0], &coef[0], 0, 0 );
}
```

Fig. 5 - Adding constrains in (3)

To add all the constraints specified in line (3) of the formula, the code in Fig. 5 was used. In the nested 2 for loops the summation of the y variables involved takes place, whose indices are detected thanks to the map_y function to map the y variables, while the coefficients in this case too are all 1.

```
/**
  Addding yij ships outside constraints [ forall j in N, sum{i:  (i,j) in A} y{i,j} = 1 ]
*/
for(int j  = 0; j < N; ++j){
  std::vector<int> idx;
  std::vector<double> coef;
  // Get index of all y vars from Y0j to Yn-1j and set coefficient equal to 1
  for(int i = 0; i < N; i++){
      if(map_y[i][j]<0) continue; // Avoid variables not mapped

      idx.push_back(map_y[i][j]);
      coef.push_back(1.0);
  }

  char sense = 'E'; // Define constraint of type equal (=)
  int matbeg = 0; // Define the beginning of the value in the matrix

  // Add row to the problem environment
  if(idx.size() != 0)
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &one, &sense, &matbeg, &idx[0], &coef[0], 0, 0 );
}
```

Fig. 6 - Adding constrains in (4)

To add all the constraints specified in line (4) of the formula, the code in Fig. 6 was used. In the nested 2 for loops the summation of the y variables involved takes place, whose indices are detected thanks to the map_y function to map the y variables, while the coefficients in this case too are all 1. it should be noted that in this case the outermost loop controls the j positions while the internal one controls the i positions, contrary to the previous implementation for the constraints (3).

```
/**
* Add flow amount constrains [Forall (i,j) in A, j != 0] => xij <= (|N|-1)yij
*/
double zero = 0.0; // var to identify value 0
// Define the constrain in the form xij - (|N|-1)yij <= 0
for(int i  = 0; i < N; ++i){
  for(int j = 1; j < N; ++j){ // Avoid select var with j = 0
    if(map_x[i][j]<0) continue; // Avoid variables not mapped

    if(map_y[i][j]<0) continue; // Avoid variables not mapped

    std::vector<int> idx(2);
    std::vector<double> coef(2);

    idx[0] = map_x[i][j];
    idx[1] = map_y[i][j];

    coef[0] = 1.0; // Define the coefficient for the x var (1)
    coef[1] = (N-1) * (-1); // Define the coefficient for the y var (-(|N|-1))
    char sense = 'L'; // Define constraint of type lower (<=)
    int matbeg = 0; // Define the beginning of the value in the matrix

    // Add row to the problem environment
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &zero, &sense, &matbeg, &idx[0], &coef[0], 0, 0 );
  }
}
```

Fig. 7 - Adding constrains in (5)

To add all the final constraints specified in line (5) of the formula, the code in Fig. 7 was used. In this case, in the nested loops, 2 variables are used at each iteration, one x and one y, for this reason the vectors of the indices and coefficients have dimension 2. The first index entered concerns that of the variable x while the second of the variable y and respectively have the coefficients set to 1 and (N-1) * (-1), this is because in Cplex it is necessary to transform the original formula $x_{ij} \leq (|N| - 1)y_{ij}$ into $x_{ij} - (|N| - 1)y_{ij} \leq 0$.

Implementation Choices

This section shows the implementation choices made for the implementation of the mathematical model. The first choice was to divide the files according to the functionality, in fact, as described in the previous section, in addition to the main file which contains all the logic for solving the problem, two files were

created containing two classes, TSPFileContainer and TSPData which define the logic, respectively, for obtaining data from a test file and storing that data in a format more suited to the mathematical model.

Then, in calls to the CPLXaddrows function, the two variables shown below were used to insert when a contraints should be <,= or > of 0/1:

```
    double one = 1.0;
    double zero = 0.0;
```

```
vector<vector<int>> map_x;
map_x.resize(N);
for(int i = 0; i < N; i++){
    map_x[i].resize(N);
    for(int j = 0; j<N; j++){
        map_x[i][j] = -1;
    }
}
```

Fig. 8 - mapping variables

Later, in Fig. 8 it shows how the decision variables have been mapped (the code refers to the x_{ij} variables but is identical for the y_{ij} variables), in practice a matrix of integers is defined, of dimension N x N where N corresponds to the number of nodes (holes) in the test file. This matrix is then initialized by setting the value -1 in all its positions, and every time a new variable is added within the problem environment, the number of the variable is saved, replacing this data in the position relative to the variable x_{ij}, therefore in position map_x[i][j].

Finally, among the implementation choices there is the choice of the method to measure the execution time of the problem. The method used to record the running time uses the chrono::high_resolution_clock::now() method contained in the <chrono> library and which returns a time point representing the current point in time. This method is then used to get the time at the start and end of executing the code for the problem in each iteration, and in each iteration, the difference between the final and initial times is saved to calculate the actual time of execution, all these times are then added to an accumulator variable and at the end of the iterations this variable will be divided by the total number of repetitions performed, in order to obtain the average execution time of a single instance of the problem. To obtain a result with identical precision for each test file tested, the setprecision() method was used, contained in the <iomanip> library, which allows you to define the number of decimal digits for the precision of the calculated time.

Second part

Algorithm Implemented

The implemented meta-heuristic algorithm is based on the genetic algorithm specifications thus following the pseudocode specification in Fig. 9

Encode solutions *of the specific problem.*
Create an initial set of solutions (*initial population*).
Repeat
  Select pairs (or groups) of solutions (parent).
  Recombine parents to generate new solutions (offspring).
  Evaluate the *fitness* of the new solutions
  Replace the population, using the new solutions.
Until (*stopping criterion*)
Return the best generated solution.

Fig. 9 - Genetic Algorithm Pseudocode

The parameters required (and the acronym in the default_parameters file) by the algorithm are:

- populationSize (PS): The size of the population to be taken into consideration.
- maxIteration (MI): The maximum number of iterations that the algorithm must perform.
- maxIterationWithoutImprovement (MIWI): The maximum number of iterations without an improvement on the solution that the algorithm must perform.
- maxTimer (MT): The maximum time (in seconds) for executing the algorithm loop.
- selectedIndividualsNumber (SIN): The number of individuals selected from the population at each iteration.
- probUseOrdCrossover (PUOC): The probability of using ORDERED CROSSOVER in generating new individuals instead of PBX CROSSOVER.
- probMutation (PM): The probability of having mutations within the generation of new individuals.
- probFastOpt (PFLS): The probability of using a Simulated Annealing (fast) algorithm to improve the new individuals generated.
- maxIterationFastLS (MIFLS): The maximum number of Iterations for the Simulated Annealing algorithm (fast).
- avgHammingDistance (AVGHD): The average of different genes among gene sequences within individuals in the population.
- useFinalOpt (UFO): Defines whether to use a Local Search algorithm to improve the final solution found.
- useLinearRanking (ULR): Defines whether you want to use a selection based on Linear Ranking (in case 1) or based on a Tournament Selection (then 0).
- tournamentDimension (TS): the size of the tournament's groups, if useLinearRanking is 0.

however, it is possible to rely on the default parameters making it unnecessary to enter the parameters. Default parameters are selected based on the number of nodes in the instance, ranges of nodes are described in the specified file and if an instance is in a range the upper bound parameters will be selected, i.e.: if we have an instance with 12 nodes, the range it's from 10 to 15, and the parameters for 15 nodes are selected.

Solution Encode

In the algorithm, an individual of the population consists of a sequence of N + 1 genes, where N is the size of the problem, where each gene in position i indicates a node (or hole) to visit at position i in the Hamiltonian cycle, thus defining a tour starting from a node j and returning to node j after passing the remaining n-1 nodes. In addition to the sequence, the individual also contains the total cost of his sequence (tour). This representation is defined by the TSPEncoding class which contains the two attributes defined for the individual.

Fitness Evaluation

To evaluate the fitness of an individual, the evaluate_fitness() function is used, which calculates the cost of the sequence contained in the individual's coding and therefore the cost of the cycle of the nodes. is calculated according to:

$$\sum_{i=0}^{n-1} c[i][i+1]$$

With n=number of genes in the sequence.

Implementation Choices

Generate Population

To generate the population, the size of the population to be generated (and to be kept at each iteration) is first defined, then from this value the sequences are generated in this way:

- 67% of the initial population is generated randomly using the method: generate_random_populations(const int &n, const vector<int> &values), which generates sequences of holes randomly (trying to avoid repetitions).
- The remaining 33% is divided equally:
    - 50% is generated using a Best Insertion Algorithm, which creates the sequences starting each time from 2 different holes.
    - 50% is generated using a Farthest Insertion Algorithm, which creates the sequences starting each time from 2 different holes.

In any case, the generation algorithm tries to verify if a generated sequence has already been inserted, to avoid repetitions within the population already in the first step. The algorithm has been made more efficient through the use of unordered_set to initially handle all sequences of holes, which allows to have a find() method which has a constant time cost (O(1)) instead of the vector find() which requires O(N).

Individuals Selection

In the selection phase of a group of individuals (sequences of holes) among the population, two types of selection are used, and consequently two different algorithms:

- Linear Ranking Selection.
- Tournament Selection.

The use of one or the other is determined by the *useLinearRanking* parameter, if it is 1 then the selection based on the ranking will be used, if it is 0 instead the type of tournament is used. In the case of selection with Linear Ranking, the linear_ranking_selection() function is used which:

1. Starting from the population sorted by increasing cost, define for each individual the probability of being selected using the formula (n-position) / (n*(n-1)/2.0) (with n = number of individuals in population).
2. For each individual then a number is randomly generated and if this number is less than the probability associated with that individual, then it is selected and removed from the participants for selection.
3. If, after going through all the individuals, the number of selected is less than the number of individuals to be selected, it starts again from point 1 with the remaining participants.

In case selection via tournament is used, it is also necessary to define the tournamentSize parameter to define the size of each selection tournament, the applied (using tournament_selection()) steps will be:

1. Initially randomize the positions of the participants (the entire population therefore).
2. Create a group of participants of size n, randomly selecting individuals based on their location.
3. Extract the best individual (the sequence with least cost) among the participants in the group and insert it among the selected individuals.
4. Continue from step 2 if the number of individuals selected is less than required.

Recombine Individuals

In the recombination phase of the selected individuals, to create new ones (new sequences), the selected individuals are grouped in pairs (position i and i+1) and the recombination of their sequences takes place using 2 different algorithms:

- Ordered Crossover
- PBX Crossover

The choice between the two algorithms depends on the initial parameters, based on the probability of choosing to use Ordered Crossover (probUseOrdCrossover), i.e if the probability is 0.7 Ordered Crossover will be used in 70% of cases while PBX Crossover in the remaining 30%, thus also diversifying this phase.

Ordered Crossover is implemented by the ordered_crossover() function which performs an ordered crossover choosing each time 2 different points of exchange between the two sequences (therefore different start and end).

PBX Crossover, on the other hand, is implemented by the PBXCrossover function, which to generate the offsprings randomly defines the positions of the genes present in the sequence of one of the two individuals, selects these positions and then the remaining ones of the offspring sequence are "filled" with the missing genes of the 'other individual, trying not to get repeats within the sequence (hence getting valid tours in this case).

After the crossover processes, the offsprings can undergo mutations and be improved by "Education", these two parts will be better explained in the next paragraphs. Finally, to apply an intensification mechanism, if in the group of newly generated individuals there are more sequences which repeat themselves, other sequences will be generated using only the PBX Crossover in this case.

Mutation

To avoid the random convergence of one or more genes towards the same value, a mutation algorithm has been implemented that randomly modifies the value of some genes, also randomly selected. Mutations can occur in the generation of offsprings, i.e., changes within the generated sequences, the possible mutations can be:

- The swap of two genes within the sequence.
- The reverse of a sub-tour of the sequence.

The probability with which a mutation can occur is dictated by the *probMutation* parameter, while the two types of mutations have respectively 66% probability for the swap and 34% probability for the reverse, to be applied.

Education

To improve the new individuals generated, an algorithm has been implemented that allows you to improve the sequences, trying to replace a sequence with its possible local minimum. During the recombination of the individuals, there is the probability (established by the *probFastOpt* parameter) which allows to perform a fast optimization on the individuals using the 2-opt local search method. The local search implemented is an accelerated version of the original, as a maximum threshold is set for the iterations to be performed (by means of maxIterationFastLS). At each iteration, the algorithm performs a 2-opt between the nodes of the sequence and the two nodes to be exchanged are defined by their positions randomly extracted in the length of the sequence. The algorithm keeps track of the updated cost in case of exchange of the two genes, if the exchange improves the sequence, then the reverse of the subsequence of the offspring genes is performed, otherwise we proceed with the next iteration and the offspring remains unchanged.

Replace Population

In the population replacement phase, it was decided to use an Elitism approach, the idea is initially to maintain the best N (population size) individuals among those already present in the population and the newly generated offsprings. The steps performed to define this substitution are:

1. Create a vector containing all individuals of the current population, plus the newly generated ones.
2. Sort them by increasing cost and choose the first N in the vector.

At this point, to avoid making the population too homogeneous, the hamming distance is used as a parameter to implement a diversification of the population, the operations that are carried out are:

1. The average of the hamming distance among all the solutions entered in the new population is calculated.
2. If the average distance is smaller than the hammingDistanceThreshold parameter, then an attempt is made to replace some solutions present in the population (between 1 and n, avoiding removing the best solution), with the solutions not inserted, the steps are:
    a. for each that has not been selected, an index between 1 and n is randomly generated to identify an individual within the population.
    b. If the hamming distance between the two sequences is greater than the required threshold, then the sequence present in the population is replaced with the one to which it was compared.

In the end the new population is generated.

Stopping Criteria

The stopping criteria are specified like the other parameters before starting the algorithm and consist of:

- Maximum iterations: the maximum number of iterations that the algorithm must perform.
- Maximum iterations without an improvement: the maximum number of consecutive iterations that the algorithm performs without experiencing an improvement in the solution.
- Time limit: the maximum time in which the algorithm must be executed (there is no maximum time in the default parameters).

These criteria allow you to obtain valid solutions in less time / with fewer executions.

Final Optimization

The SA algorithm was created in order to optimize the final solution in case the parameter is set, while the rest of the algorithm is implemented as a normal SA, with the variant that the initial temperature is set to (numGenes-1) * 2, while the coolingRate at 1 - (0.01 * (100 / (numGenes-1))), the SA is implemented so that it accepts solutions that do not improve on the initial solution, but will always eventually return the best accepted solution.

Parameter Calibration

Genetic algorithms, especially this one, depend a lot on the entered parameters, in my case the parameters are those defined at the beginning of the genetic algorithm section. The calibration of these parameters has not been done on all the instances used in cplex, but only on some instances defined in the test results that can be viewed in the section Results, this is because it would take much more time to test each instance, especially the more substantial ones and adjust each parameter accordingly. From the tests carried out, the calibration of the parameters is defined by the default_parameters.txt file (Parameter values for instances with 300 or more nodes have not been tested to speed up the process, so defined values are guesses for optimal values), in which the instances are divided by range of nodes and for each range suitable parameters are defined for each type of TSP problem. The explanations on all the parameters have not been included because some vary based on the instances used and the user's desire to try certain configurations, I will only focus on a small group of parameters.

Population Size

The size for the initial population for each instance is almost always increasing as the number of nodes increases, however the values are not too large so as not to burden the execution of the algorithm too much, but also because a larger population size initial does not seem to bring much improvement as too many instances could be considered which would not lead to an optimal or very good solution.

Maximum Iterations

For the maximum number of iterations, an increasing value was defined as the number of nodes (or holes) grew, trying in any case not to reach exaggerated figures to balance a good evolution of the population initial and computational costs.

Time Limit

No maximum time has been set in the calibration, first to check without restrictions how long each instance took to give a solution with these parameters, and because I think that the maximum time depends on the user's needs and therefore it is not possible to define a default value valid for each user. The execution time, then very often it is less than the execution of the same instances with Cplex.

Probability of Mutations

In defining the mutation probability for the calibration, I have tried to keep it low in almost all instances since a high mutation probability increases the probability of explore more solutions, but it slows down the convergence to the global optimum, however it was not defined too low either because otherwise there was the risk of obtaining a premature convergence (local optimum).

Future Implementations

Some possible improvements and/or future implementations may concern:

- Parallelize population generation: The generation of random sequences, "best_insertion_random" and "farthest_insertion_random" can be done in parallel to improve performance. However, we must consider how to manage the duplication of sequences in this case.
- Maybe try to decrease the parameters to insert and consequently leave the choice of parameters to the algorithm.
- Perform more Tests regarding the calibration of the parameters especially for instances with more than 50 nodes.
- Increase escalation and diversification steps, perhaps by providing different types of algorithms for final optimization based on the number of nodes or inserting other types of selection and/or crossovers in the process.

Test

Test Format

Test files generally describe data for instances of the Traveling Salesman Problem (TSP). In this case, however, they have been adapted thinking of the reference problem, i.e. for the creation of holes in the boards.

Each file is formatted as follows:

- The first line, "NAME : dataset_10_04", gives the name of the data file.
- The second line, "NODES : 10", indicates contains the number of holes to be made on the board.
- The third line, "TYPE : SYMMETRIC", specifies that the distance between any two holes is the same in both directions. In other words, if the distance from hole A to hole B is X, the distance from hole B to hole A is also X.

- The remaining lines give the distance between pairs of holes. Each line has three fields: "NODE1", "NODE2", and "TIME". "NODE1" and "NODE2" are the indices of the two holes being connected, and "TIME" is the distance between them. For example, the line "0 1 37.9471" indicates that the distance between hole 0 and hole 1 is 37.9471 seconds.

these last lines will then give the formatting to the matrix of the distances of the nodes, an example (with 10 holes) is the following:

	0	1	2	3	4	5	6	7	8	9
0	0	37.9471	84.2345	15.1234	97.9876	58.6543	13.4789	78.1234	50.9876	9.6543
1	37.9471	0	38.0123	84.3456	15.2345	58.7654	13.5678	78.2345	51.0987	9.7654
2	84.2345	38.0123	0	38.1234	84.4567	15.3456	58.8765	13.6789	51.2098	9.8765
3	15.1234	84.3456	38.1234	0	38.2345	84.5678	15.4567	58.9876	51.321	9.9876
4	97.9876	15.2345	84.4567	38.2345	0	38.3456	84.6789	15.5678	59.0987	10.0987
5	58.6543	58.7654	15.3456	84.5678	38.3456	0	38.4567	84.789	15.6789	59.2098
6	13.4789	13.5678	58.8765	15.4567	84.6789	38.4567	0	38.5678	84.8901	15.789
7	78.1234	78.2345	13.6789	58.9876	15.5678	84.789	38.5678	0	38.6789	84.9012
8	50.9876	51.0987	51.2098	51.321	59.0987	15.6789	84.8901	38.6789	0	38.789
9	9.6543	9.7654	9.8765	9.9876	10.0987	59.2098	15.789	84.9012	38.789	0

Generation

The test files used during the analysis of the two implementations for the TSP problem were generated in the following ways:

- for files with 3 to 5 nodes: they were created manually as they are easier to write.
- the 10-node files: they were generated using an online software called **ChatGPT**, which is an AI-based chatbot prototype developed by **OpenAI** that specializes in dialogue [1]. I wanted to use it to test its ability to create test files for algorithms or mathematical models and because it made it easier for me to create files.
- For files from 25 to 300 nodes: I made a small script in **Kotlin** that allows me to generate such files, these scripts will be attached to the report if needed.
- Some instances (in the “special” folder) are taken from [2], and converted to my format, using the script in python in Google Collab [3] in Fig. 10.

```python
import math

def euclidean_distance(x1, y1, x2, y2):
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def read_tsp_file(file_path):
    with open(file_path) as f:
        lines = f.readlines()

    node_coord_section_index = lines.index("NODE_COORD_SECTION\n") + 1
    node_coords = []
    for line in lines[node_coord_section_index:]:
        if line == "EOF\n":
            break
        x, y = map(int, line.strip().split()[1:])
        node_coords.append((x, y))

    num_nodes = len(node_coords)
    distance_matrix = [[0] * num_nodes for _ in range(num_nodes)]
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            x1, y1 = node_coords[i]
            x2, y2 = node_coords[j]
            distance = euclidean_distance(x1, y1, x2, y2)
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance

    return distance_matrix

def write_distance_matrix_to_file(file_name, num_nodes, distance_matrix):
    with open(file_name, 'w') as file:
        file.write("NAME : " + file_name + "\n")
        file.write("NODES : " + str(num_nodes) + "\n")
        file.write("TYPE : SYMMETRIC\n")
        file.write("NODE1 NODE2 TIME\n")

        for i in range(num_nodes):
            for j in range(i+1, num_nodes):
                file.write(str(i) + " " + str(j) + " " + str(distance_matrix[i][j]) + "\n")



distance_matrix = read_tsp_file("st70.tsp")
write_distance_matrix_to_file("st70.dat", 70, distance_matrix)
```

Fig. 10 - Script to convert .tsp to my .dat file

Results

In this session the results obtained by running the algorithms on the datasets within the project are defined. The two algorithms were compared, evaluating the difference in execution times between the algorithm in Cplex which defines an optimal solution and the genetic algorithm which instead very often resists only a good (feasible) solution.

Note: for the genetic algorithm (GA) the tests were carried out starting from instances of 10 nodes.

Legend:

- **File Name**: name of the file.
- **N**: number of nodes.
- **Solution**: solution for the data file.
- **Time Home**: execution time in seconds at home.
- **Time Lab**: execution time in seconds at laboratory.
- **Repetitions**: number of repetitions for the instance.
- **N° opt. sol. found**: Number of time that the algorithm found the optimal solution.
- **AVG Difference**: the AVG difference from the optimal solution and the one provided by the algorithm.
- **AVG Time**: AVG time of execution.
- **Min val.**: minimum value obtained from the repetitions.
- **Max val.**: maximum value obtained from the repetitions.
- **% Success**: percentual of success in finding the optimal solution.

Table 1 - Cplex Algorithm Data

File Name	N	Solution	Time(s) Home	Time(s) Lab	Repetitions
dataset_3_01	3	5	0.00248	0.009388220	10000

dataset_3_02	3	12	0.00305	0.009632940	10000
dataset_3_03	3	7.4	0.00298	0.009604180	10000
dataset_3_04	3	293.39	0.00295	0.009435380	10000
dataset_5_01	5	5	0.00299	0.010183300	1000
dataset_5_02	5	208.12312	0.02625	0.021202700	1000
dataset_5_03	5	242.66662	0.02918	0.021881400	1000
dataset_5_04	5	218.66664	0.02192	0.018247900	1000
dataset_10_01	10	221.6666	0.03393	0.026749700	1000
dataset_10_02	10	132	0.01759	0.022702600	1000
dataset_10_03	10	135.3623	0.03647	0.024632400	1000
dataset_10_04	10	162.5399	0.04598	0.036254200	1000
dataset_25_01	25	708.0482	0.27552	0.271872000	1000
dataset_25_02	25	707.4864	0.15949	0.158908000	1000
dataset_25_03	25	618.7828	0.23613	0.290056000	1000
dataset_25_04	25	265.2828	0.70045	0.480417000	1000
dataset_50_01	50	280.8588	1.95692	1.439590000	100
dataset_50_02	50	434.7995	1.63348	0.993495000	100
dataset_50_03	50	445.2178	1.34009	0.803261000	100
dataset_50_04	50	592.8718	1.65238	1.904990000	100
dataset_100_01	100	735.1676	32.76090	87.111600000	10
dataset_100_02	100	598.5325	18.91520	44.743300000	10
dataset_100_03	100	730.0877	28.57790	78.010900000	10
dataset_100_04	100	685.4287	20.86430	96.398600000	10
dataset_250_01	250	881.1187	6375.18000	7758.170000000	1
dataset_250_02	250	789.3718	3799.36000	4211.320000000	1

In the Table 1 is possible to see the data analyzed for the algorithm in Cplex, in which the execution times are also shown (both at "home" and in the laboratory) for each instance. Times as expected increase as the number of nodes in the instance increases, eventually reaching unacceptable times for instances with 250 nodes. Particularly in laboratory tests times are worse.

Table 2 - Genetic Algorithm Data at Home

File Name	N	N° Opt. Sol. Found	AVG difference	Avg Time (s)	Min val.	Max val.	% Success	Repetitions
dataset_10_01	10	697	4.079	0.00214	221.667	288	0.697	1000
dataset_10_02	10	591	2.524	0.00246	132	180	0.591	1000
dataset_10_03	10	568	2.82384	0.00249	135.362	184.834	0.568	1000
dataset_10_04	10	979	0.597008	0.00283	162.54	198.182	0.979	1000
dataset_25_01	25	80	63.8391	0.22786	708.048	906.34	0.08	1000
dataset_25_02	25	0	90.9298	0.23170	707.679	899.348	0	100
dataset_25_03	25	0	73.9661	0.22178	622.835	814.797	0	100
dataset_25_04	25	4	19.0789	0.22576	265.283	323.644	0.04	100
dataset_50_01	50	0	78.9153	1.47357	307.308	424.015	0	100
dataset_50_02	50	0	150.352	1.47604	485.697	668.09	0	100
dataset_50_03	50	0	165.556	1.52424	530.667	746.013	0	100
dataset_50_04	50	0	297.94	1.53016	760.245	1103	0	100
dataset_100_01	100	0	454.428	22.06417	1077.97	1278.66	0	10
dataset_100_02	100	0	422.052	20.92818	876.379	1148.9	0	10

dataset_100_03	100	0	393.473	23.01230	966.221	1249.72	0	10
dataset_100_04	100	0	427.19	21.75510	1027.32	1220.59	0	10
dataset_250_01	250	0	793.3	259.64958	1576.29	1755.12	0	10
dataset_250_02	250	0	852.119	255.63556	1596.35	1723.31	0	5

On the other hand, the Table 2 shows the times for the genetic algorithm, the times are in line with those of the Cplex algorithm, with a marked improvement in instances with 250 nodes.

Table 3 - Genetic Algorithm Data at Lab

File Name	N	N° Opt. Sol. Found	AVG difference	Avg Time (s)	Min val.	Max val.	% Success	Repetitions
dataset_10_01	10	769	2.52533	0.000601136	221.667	275.333	0.769	1000
dataset_10_02	10	711	1.396	0.000593945	132	180	0.711	1000
dataset_10_03	10	726	1.26327	0.000593161	135.362	162.011	0.726	1000
dataset_10_04	10	998	0.071284	0.00059148	162.54	198.182	0.998	1000
dataset_25_01	25	111	54.3849	0.045544839	708.048	901.425	0.111	1000
dataset_25_02	25	39	73.2442	0.046745481	707.486	956.764	0.039	1000
dataset_25_03	25	14	65.5397	0.043799326	618.783	831.845	0.014	1000
dataset_25_04	25	30	17.3328	0.040381265	265.283	329.605	0.03	1000
dataset_50_01	50	0	59.6887	0.781084704	302.161	407.078	0	100
dataset_50_02	50	0	118.799	0.837608094	486.576	635.262	0	100
dataset_50_03	50	0	137.348	0.826226193	508.587	690.64	0	100
dataset_50_04	50	0	218.262	0.871495871	715.123	980.112	0	100
dataset_100_01	100	0	384.656	10.6993706	1024.83	1264.31	0	10
dataset_100_02	100	0	385.058	10.9131321	880.599	1065.74	0	10
dataset_100_03	100	0	414.992	10.8560356	1070.16	1199.78	0	10
dataset_100_04	100	0	377.238	11.1033916	1006.34	1153.92	0	10
dataset_250_01	250	0	822.602	200.194159	1603.37	1819.32	0	10
dataset_250_02	250	0	794.231	226.472363	1449.4	1686.69	0	5

Furthermore, the tests for the genetic algorithm were also carried out in the Torre Archimede laboratory, and initially execution times were better than those obtained at home, for this reason the calibration parameters were changed making them more "aggressive" so to obtain better solutions and times as shown in the Table 3.

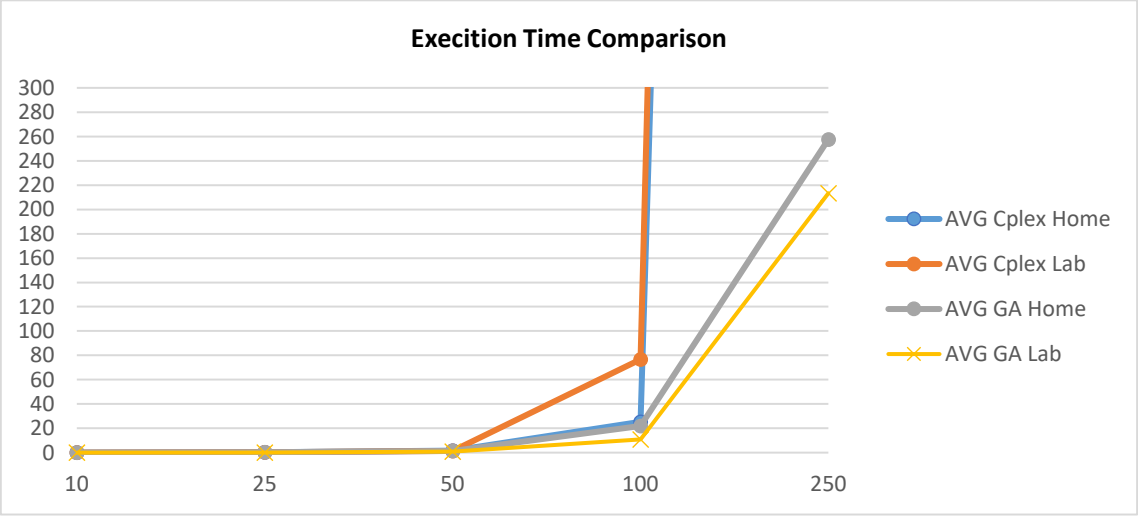

Fig. 11 - AVG time execution for the algorithms

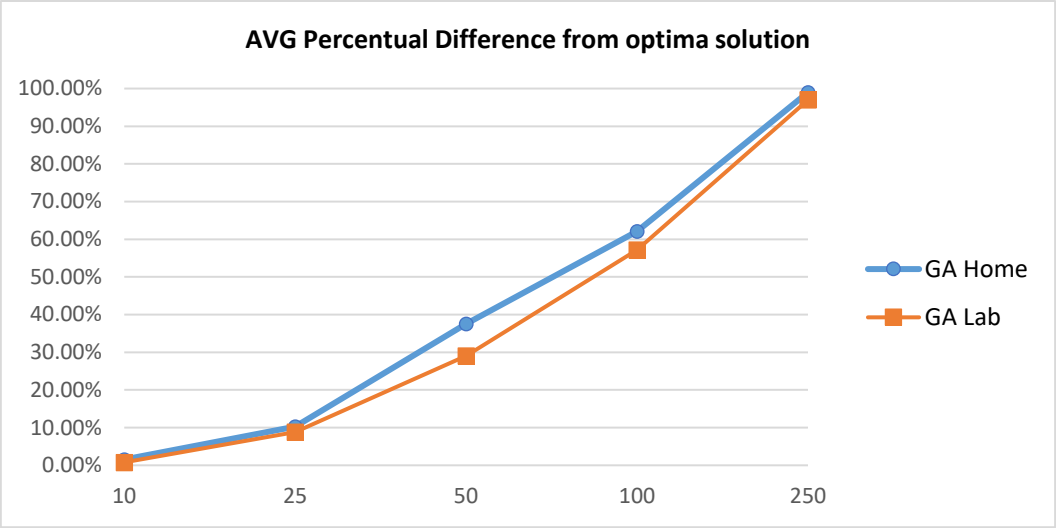

Fig. 12 - Percentual of different from the optimal solution for GA

Finally, analyzing the graphs in the figures, the execution times of the two algorithms are almost identical up to 50 nodes (GA is slightly better), after which the genetic algorithm improves. However, seeing the data in the tables and analyzing how many times the optimal solution is found (or approaches) using the genetic algorithm, I feel like saying that:

- For instances with 10 nodes, it is preferable to use the genetic algorithm as it finds the optimal solution with a high percentage and the execution time is slightly less than Cplex.
- For instances with 25 nodes, the data depends a lot on the type of instance, but I would recommend using the genetic algorithm, perhaps modifying the default parameters used.
- For instances with 50 to 100 nodes I would recommend using Cplex If you need an optimal solution, instead the GA in case you want shorter execution times with a good solution anyway.
- Finally, for instances with 250 nodes and more, I would evaluate more the parameters to use for the genetic algorithm, because the execution times at the moment are really low compared to Cplex, but the average distance from the optimal solution is almost double compared to the solution, however, this can be improved in my opinion by increasing the execution time in favor of a greater number of iterations and strengthening of the parameters.

Finally, I would like to say that the data is based on parameter calibrations which may take longer than expected to be optimal.

Test Calibration

This section shows the default parameters defined, and additional files used for parameter tests.

NODES	PS	MI	MIWI	MT	SIN	PUOC	PM	PFLS	MIFLS	AVGHD	UFO	ULR	TS
5	3	10	9	/	2	1	0.1	0.1	1	1	0	1	0
10	5	100	50	/	2	0.8	0.2	0.7	8	3	0	1	0
15	6	150	100	/	3	0.8	0.2	0.7	9	3	0	1	0
25	10	500	250	/	6	0.7	0.2	0.3	46	5	0	1	0
50	22	1000	650	/	13	0.65	0.22	0.4	60	8	1	0	4
75	25	1300	900	/	15	0.7	0.22	0.33	70	9	1	0	8
100	40	2000	1200	/	25	0.7	0.2	0.3	80	10	1	0	11
250	50	3000	1800	/	28	0.6	0.2	0.3	140	18	0	0	12
300	45	2800	1300	/	22	0.5	0.25	0.32	250	30	0	0	10
500	54	3200	1400	/	27	0.5	0.25	0.31	300	40	0	0	11
1000	70	4000	2000	/	35	0.5	0.25	0.3	400	50	0	0	13

The files used for calibration are those placed in the dataset\calibration folder. The values refer to the tests carried out in the laboratory, any values for home testing can be found in the old_parameters.txt file. It also indicates that for instances with 300 or more nodes the parameters have not been tested but are only values to have default parameters for those file types. The table with the data relating to the calibration test files is shown below.

FileName	N	Opt. Solution	N° Opt. Sol. Found	AVG difference	Avg Time (s)	Min val.	Max val.	% Success	Repetitions
dataset_10_01	10	221.6666	769	2.52533	0.000601136	221.6666	275.333	0.769	1000
dataset_25_01	25	708.0482	111	54.3849	0.045544839	708.048	901.425	0.111	1000
dataset_50_02	50	434.7995	0	118.799	0.837608094	486.576	635.262	0	100
dataset_100_02	100	598.5325	0	385.058	10.9131321	880.599	1065.74	0	10
dataset_250_01	250	881.1187	0	822.602	200.194159	1603.37	1819.32	0	10
p01_d	15	291	997	0.048	0.002632858	291	307	0.997	1000
att48	48	33523	0	760.29	0.557699393	33614	35238	0	100
st70	70	675	0	21.7188	1.90162003	677.11	726.483	0	100

Fig. 13 - Test on Calibration

Instruction

How to run the code

Cplex Algorithm

To use the algorithm in Cplex it is necessary to perform the **Make** operation with the appropriate makefile contained in the Cplex folder (There is an additional Makefile_home which was used for home testing). To use the program defined with Cplex, just insert the following line of code:

```
.\main .\dataset\dataset_250_02.dat
```

making sure to insert ".\main" and a test file with the format previously defined (the files can be found in dataset folder).

Note: Rarely the execution of the code can lead to exposing an error, in this case it is only necessary to launch the program again as it is an error due to the name of the variables.

Genetic Algorithm

To use the Genetic Algoritm it is necessary to perform the **Make** operation with the appropriate makefile contained in the Meta-Heuristic folder. Initially, if you are unable to use the program, just try to recall the main through the following command:

.\main and help will appear as in Fig. 14.

```
PS C:\Users\marco\Desktop\Narde\Università\Computer Science\Methods And Models For Combinatorial Optimization\2_parte_progetto\Refactoring_2> .\main
HI! it looks like you want to use the program, here are some suggestions:
Multiple types of usage:

1) Usage: ./main [OPTION]
Options:
  -h, --help            Display this help message

2) Usage: ./main [filename]
Allows you to enter the file to be tested, with the parameters chosen directly from the program
Example: ./main .\dataset\dataset_10_01.dat   -> Run the programm on the file dataset_10_01.dat (Check documentation for the file format)

3) Usage: ./main filename.dat [populationSize] [maxIteration] [maxIterationWithoutImprovement] [maxTimer] [selectedIndividualsNumber] [probUseOrdCrossover] [probMutation] [probFastOpt] [maxIterationFastLS] [avgHammingDistance] [useFinalOpt] [useLinearRanking] [tourna
mentDimension]
- populationSize: The size of the population to be taken into consideration (INTEGER)
- maxIteration: The maximum number of iterations that the algorithm must perform (INTEGER)
- maxIterationWithoutImprovement: The maximum number of iterations without an improvement on the solution that the algorithm must perform (INTEGER)
- maxTimer: The maximum time (in seconds) for executing the algorithm loop (does not take into account a possible final optimization!), if you don't want to take or the time put '/' (INTEGER)
- selectedIndividualsNumber: The number of individuals selected from the population at each iteration (INTEGER)
- probUseOrdCrossover: The probability of using ORDERED CROSSOVER in generating new individuals instead of PBX CROSSOVER (DECIMAL between 0-1) EX: if is 0.7 -> prob(ORDCROS.) = 0.7 and prob(PBXCROS.) = 0.3
- probMutation: The probability of having mutations within the generation of new individuals (DECIMAL between 0-1)
- probFastOpt: The probability of using a Simulated Annealing (fast) algorithm to improve the new individuals generated (DECIMAL between 0-1)
- maxIterationFastLS: The maximum number of Iterations for the Simulated Annealing algorithm (fast) in case its probability is > 0 (INTEGER)
- avgHammingDistance: The average of different genes among gene sequences within individuals in the population (INTEGER should be between 1 and the number of genes (nodes) for each sequence) EX (hamming distance): ([0 1 2 3 0] , [0 2 1 3 0]) has 2 different genes
- useFinalOpt: Defines whether to use a Local Search algorithm to improve the final solution found (0/1)
- useLinearRanking: Defines whether you want to use a selection based on Linear Ranking (in case 1) or based on a Tournament Selectio (then 0) (0/1)
- tournamentDimension: If you decide to use Tournament Selection (then useLinearRanking = 0) then you need to define a value for the size of the subsets for the selection, otherwise put '/' (INTEGER)
```

Fig. 14 - Help screen

This screen will also be displayed if main is invoked with the --help (or -h) option as described in the help.

Then to run the genetic algorithm for the TSP you have the following options:

- Recall only the main file and then the dataset on which you want to test the algorithm as:

```
.\main .\dataset\calibration\st70.dat
```

  in this way the program will be executed using the default parameters contained in the **default_parameters.txt** file.

- Use the same call as before, but this time specifying all the parameter values that are listed by the help, then the parameters defined in Algorithm Implemented.

Conclusion

In conclusion it can be said that the Cplex algorithm is currently excellent for calculating optimal solutions for instances up to 100 nodes, after which the times start to worsen exponentially, in these cases it is therefore better to use the genetic algorithm with better parameters tested. In addition to the execution time, it is then necessary to decide whether we can sacrifice time to obtain an optimal response or if instead it is better to obtain a good response in less time, from which it follows that the Cplex algorithm allows us to obtain in 99% of the cases the excellent solution but with not always optimal times, while the generic algorithm especially for some instances is an excellent alternative if we have limited resources. However, analyzing the data collected in the laboratory for the generic algorithm, the execution time is much lower than that calculated using cplex and for this reason the parameters can be further intensified to sacrifice some execution time but allowing get better, if not optimal results as in instances with nodes less than 25. A final note concerns the dataset used, almost all instances with the dataset_##_##.dat prefix are randomly generated instances and therefore do not correspond to real data for the problem in question, this could have influenced the tests of the algorithms that therefore they are to be evaluated carefully with many more real instances.

References

[1] https://openai.com/blog/chatgpt/
[2] https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html
[3] https://colab.research.google.com