UNIVERSITY OF PADOVA

Civil, Environmental and Architectural department DICEA

Master's degree in Mathematical Engineering

Advanced Solid Mechanics: Second Homework

Andrea Gorgi, Badge 2010658

Academic year 2020/2021

Contents

1 Introduction and creation of test configurations

In the following chapters is presented the work carried out for the project. Part 1 presents 3 possible exact formulations in Cplex plus an additional symmetric adaptation, whereas part 2 illustrates how a genetic algorithm has been customized to solve the drilling problem.
The sets of data used to verify the models have been created using Matlab [4] in two different ways. The first has been to create panels with the holes disposed at random in the domain, but since this is not very representative of the problem (printed circuits have symmetries and usually the holes are disposed in lines or squares), another possibility, with the holes disposed in squares or lines has been explored. In addition I've used three sets of data from the library TSPLIB for the drilling machine problem [6] as final benchmarks. In all the cases has been used an euclidean distance between the nodes as a measure of the cost in time to move from one node to another, assuming that a machine can move in whatever direction in space.

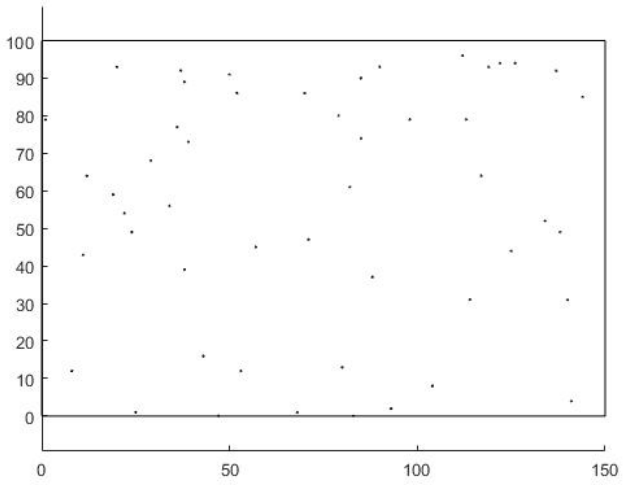

Fig.(1) Example of panel with 50 random holes

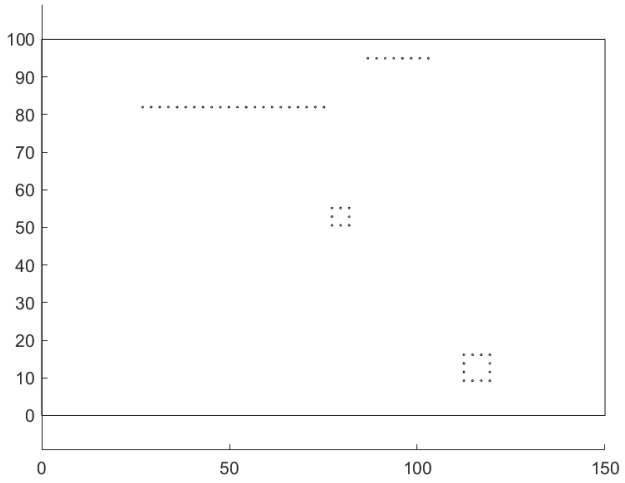

Fig.(2) Example of panel customized with 50 holes

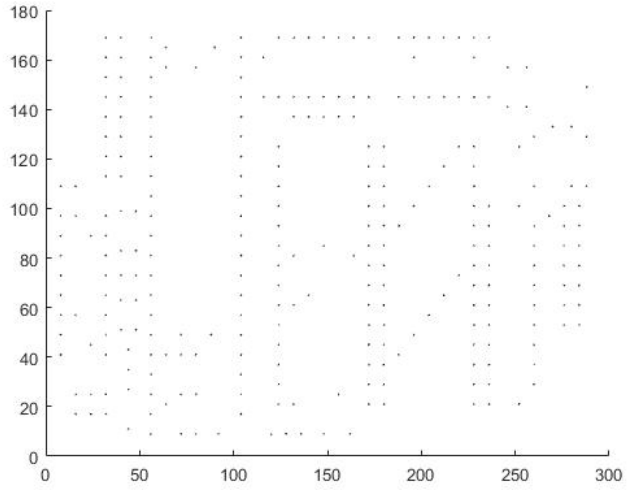

Fig.(3) Data a280.tsp of TSPLIB

2 Part 1: Cplex

2.1 First Model: Miller-Tucker

The first model implemented is the one proposed by Miller ([3]-[2]). Let N be the graph nodes, for the case of the drilling problem it makes sense to assume that all arcs $(i, j), i, j \in N$ are possible (note that even if there were an obstacle between two generic holes and the drill cannot move directly from one node to the other, one could just increment the cost of that arc buy the time needed to avoid the obstacle, or set that cost an extremely high number if it's really impossible to follow that path).
As in the model proposed in the project I referred with c_{ij} the cost (in time) of path $(i, j), \forall i, j \in N$, but the decision variables are slightly different.

- x_{ij}= 1 if arc (i, j) is selected, 0 otherwise, $\forall i, j \in N$;
- U_i = rank of node i in the path, $\forall i \in N$.

The Integer Linear Programming model is then

$$min \sum_{i,j \in N} c_{ij} x_{ij} \tag{1}$$

subject to:

$$\begin{aligned}
&\sum_{i \in N} x_i j = 1 && \forall j \in N \\
&\sum_{j \in N} x_{ij} = 1 && \forall i \in N \\
&U_i + x_{ij} \leq U_j + (n-1)(1 - x_{ij}) && \forall i \in N, \forall j \in N \setminus \{0\}, i \neq j \\
&x_{ij} = 0, 1 && \forall i, j \in N,
\end{aligned} \tag{2}$$

where $n = |N|$. The first two constraints are the same "come-from" and "go-to" constraints of the proposed model, and the third one is a subtour elimination constraints ($(x_{ij} = 1 \implies U_i + 1 = u_j)$ written in a big-M form. This approach, that we will call (MTZ) has $\mathcal{O}(n^2)$ variables and $\mathcal{O}(n^2)$constraints.

2.1.1 Cplex implementation ("Part 1/Miller")

I firstly create a map vector for the indices of x_{ij} variables.

```
//-------------------------------------------------------------------------------------------
/*MAP FOR X VARS: initial memory allocation for map vector*/
        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;
                        }
                }
//-------------------------------------------------------------------------------------------
```

Then we create one x_{ij} variable (updating the map_x vector) and U_i variable at a time inside a for loop

```
//-------------------------------------------------------------------------------------------
// CREATE VARIABLES

        // create x_{ij} variables
        const int x_init = 0; //CPXgetnumcols(env, lp);
        int current_var_position = x_init;
        for (int i = 0; i < N; i++)
        {
                for (int j = 0; j < N; j++)
                {
                        if (i == j) continue;
                        char xtype = 'B';
                        double obj = data.C[i*N + j];
                        double lb = 0.0;
                        double ub = 1.0;
                        snprintf(name, NAME_SIZE, "x_%d_%d", i, j);
                        char* xname = (char*)(&name[0]);
                        CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &xtype, &xname )
                        map_x[i][j] = current_var_position;
            ++current_var_position;
                }
        }
        //-------------------------------------------------------------------------------------
        // create U_i variables [rank of path]
        const int U_init = CPXgetnumcols(env, lp);
        for (int i = 1; i < N; i++)
        {
                char Utype = 'C';
                double obj = 0.0;
                double lb = 0.0;
                double ub = N - 1;
                snprintf(name, NAME_SIZE, "U_%d", i);
                char* Uname = (char*)(&name[0]);
                CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &Utype, &Uname );
        }
```

```
        //------------------------------------------------------------------------------------
```

and add the rows of the linear system relative to the constraints

```
//------------------------------------------------------------------------------------
        // ADD CONSTRAINTS

        // ADD GO-TO CONSTRAINTS [ sum_{j : (i,j) in A}[x_{ij}] = 1 ]
        int nindeces = N - 1;
        for (int i = 0; i < N; i++)
        {
                int curr_index = 0;
                std::vector<int> idx(nindeces);
                std::vector<double> coef(nindeces);

                for (int j = 0; j < N; j++) {
                        if (j == i) continue;
                        idx[curr_index] = map_x[i][j];
                        coef[curr_index] = 1.0;
                        ++curr_index;
                }
                char sense = 'E';
                int matbeg = 0;
                double rhs = 1.0;
                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg, &idx[0]
                &coef[0], 0, 0 );
        }
        //------------------------------------------------------------------------------------
        // ADD COME-FROM CONSTRAINTS [ sum_{i : (i,j) in A}[x_{ij}] = 1 ]
        nindeces = N-1;
        for (int j = 0; j < N; j++)
        {
                int curr_index = 0;
                std::vector<int> idx(nindeces);
                std::vector<double> coef(nindeces);

                for (int i = 0; i < N; i++) {
                        if (j == i) continue;
                        idx[curr_index] = map_x[i][j];
                        coef[curr_index] = 1.0;
                        ++curr_index;
                }
                char sense = 'E';
                int matbeg = 0;
                double rhs = 1.0;
                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg, &idx[0]
                &coef[0], 0, 0 );
        }
        //------------------------------------------------------------------------------------
        // ADD SUBTOUR ELIMINATION BIG-M CONSTRAINT [ U_i - U_j + N x_{ij} <= N - 1]

        for (int i = 0; i < N; i++) {
                for (int j = 1; j < N; j++){
                        if (i == j) continue;
```

```
                        if(i == 0){
                                std::vector<int> idx(2);
                                std::vector<double> coef(2);
                                idx[0]  = U_init + j - 1;
                                idx[1]  = map_x[i][j];
                                coef[0] = -1.0;
                                coef[1] =  N;
                                char sense = 'L';
                                int matbeg = 0;
                                double rhs = N - 1.0;
                                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &se
                                &idx[0], &coef[0], 0, 0 );
                                continue;
                        }
                        std::vector<int> idx(3);
                        std::vector<double> coef(3);
                        idx[0]  = U_init + i - 1;
                        idx[1]  = U_init + j - 1;
                        idx[2]  = map_x[i][j];
                        coef[0] =  1.0;
                        coef[1] = -1.0;
                        coef[2] =  N;
                        char sense = 'L';
                        int matbeg = 0;
                        double rhs = N - 1.0;
                        CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &ma
                        &coef[0], 0, 0 );
                }
        }
        //------------------------------------------------------------------------------------
        // debug
        CHECKED_CPX_CALL( CPXwriteprob, env, lp, "drillingTSP.lp", NULL );
//------------------------------------------------------------------------------------
```

Then the rest of the code simply calles the Cplex solve, asks the user to choose whether print or not the console outputs of Cplex to the screen (flag with values 1 or 0), and prints the results in the terminal, in terms of user and CPU time needed, the final path found, and for $n < 20$ the x_{ij} and U_i variables.

2.1.2 Example: "random_panels/size10/dati5.dat"

Result for "random_panels/size10/dati5.dat" problem: One can easily check that there is only one "1" for each row and column of the x matrix and that the U_i are the positions of each node "i" in the final path.

```
Solving the linear drilling TSP
Objval: 665.155

x:
    0    -0    -0     1    -0    -0     0    -0    -0    -0
   -0     0     1    -0    -0    -0    -0    -0    -0    -0
   -0     0     0    -0     0    -0     1    -0    -0    -0
   -0    -0    -0     0     0    -0    -0     1    -0    -0
   -0     1    -0     0     0    -0    -0    -0    -0    -0
   -0     0    -0     0     1     0    -0    -0     0     0
    1    -0     0     0    -0    -0     0    -0    -0    -0
   -0    -0    -0     0     0    -0    -0     0     0     1
   -0    -0    -0     0    -0     1    -0    -0     0    -0
   -0    -0    -0     0    -0    -0    -0    -0     1     0

U:
7
8
1
6
5
9
2
4
3

Optimal solution:
0 --> 3 --> 7 --> 9 --> 8 --> 5 --> 4 --> 1 --> 2 --> 6 --> 0
in 5.28325 seconds (user time)
in 0.21875 seconds (CPU time)
```

Fig.(4) Results of file "random_panels/size10/dati5.dat" with Miller formulation.

2.2 Second Model: Single-Commodity Flow (SCF)

This is the case proposed in the project assignment. Such model differs from the above one in the way used to eliminate subtours, since here we introduce other variables than U_i, the x_{ij} ones (we have here called y_{ij} the binary variables for the selected arcs, to be consistent with the assignment text), that represent the amount of flow from each node i to node $j \neq i$,. This model has $\mathcal{O}(n^2)$ variables and $\mathcal{O}(n)$ constraints, and can be proved to be more efficient than the Miller formulation.

2.2.1 Implementation: "Part 1/SCF

Here we have used two map vectors for the indices of the variables x_{ij} and y_{ij}. I report below only the implementation of the x_{ij} variables and relative constraints, since the y_{ij} variables have the same structure used in Miller's formulation for the x_{ij} ones.

```
//------------------------------------------------------------------------------------
        // create x_{ij} variables
        const int x_init = CPXgetnumcols(env, lp);
        current_var_position = x_init;
        for (int i = 0; i < N; i++)
        {
                for (int j = 1; j < N; j++)
                {
                        if (i == j) continue;
                        char xtype = 'C';
                        double obj = 0.0;
                        double lb = 0.0;
                        double ub = CPX_INFBOUND;
                        snprintf(name, NAME_SIZE, "x_%d_%d", i, j);
                        char* xname = (char*)(&name[0]);
                        CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &xtype, &xname )
                        map_x[i][j] = current_var_position;
            ++current_var_position;
                }
        }
//------------------------------------------------------------------------------------
// ADD CONSTRAINTS

        // ADD UNITARY FLOW CONSTRAINT - SUBTOUR EL. CONSTRAINTS [sum_{i : (i,k) in A}[x_{ik}]
        - sum_{j : (k,j) in A, j != 0} [x_{kj}]= 1 ]
    int nindeces = 2 * (N-1) + 1;
        for (int k = 1; k < N; k++)
        {
                int curr_index = 0;
                std::vector<int> idx(nindeces);
                std::vector<double> coef(nindeces);

                for (int i = 0; i < N; i++) {
                        if (k == i) continue;
                        idx[curr_index] = map_x[i][k];
                        coef[curr_index] = 1.0;
                        ++curr_index;
                }
                for (int j = 1; j < N; j++) {
                        if (k == j) continue;
                        idx[curr_index] = map_x[k][j];
                        coef[curr_index] = -1.0;
                        ++curr_index;
                }
                char sense = 'E';
                int matbeg = 0;
                double rhs = 1.0;
                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg, &i
```

```
                &coef[0], 0, 0 );
        }
    //------------------------------------------------------------------------------
    // ADD BIG-M CONSTRAINT [ x_{ij} <= (|N| - 1) y_{ij}]

        for (int i = 0; i < N; i++) {
                for (int j = 1; j < N; j++){
                        if (i == j) continue;
                        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;
                        coef[1] = -(N - 1.0);
                        char sense = 'L';
                        int matbeg = 0;
                        double rhs = 0.0;
                        CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg,
                        &coef[0], 0, 0 );
                }
        }
//------------------------------------------------------------------------------
```

2.2.2 Example: "random_panels/size10/dati5.dat"

Result for "random_panels/size10/dati5.dat" problem: One can easily check that there is only one "1" for each row and column of the y matrix and that the y_{ij} are the amount of flow from node i to j, starting from n-1 at node 0. Note that the first column of zeros in the x table refers to variables x_{i0}, which were not created in the model, but that we now must be equal to zero, as well as for the x_{ii} variables.

2.3 Third Model: Symmetric adaptation

As written in the first paragraph, I have decided to study the case of euclidean distances as a measure of the time needed to move from one hole to another. In such case it's clear that the cost matrix will result as symmetric, and we could try to exploit some features from it.
For example we notice that it's not really necessary to have two different variables y_{ij} and $y_{ji}, \forall i, j \in N, i \neq j$, since they both refer to the same arc to be selected, and for the objective function it doesn't change if the arc is used in one or the other direction. We are then going to try a formulation with only half of the y_{ij} variables, precisely the lower triangular part of the previous formulation. The loop between all the vertices is guaranteed by the x_{ij} variables, that force the "traveler" to leave one unit of commodity for each city visited.
In this case the "come-from" and "go-to" constraints collapse into one macro constraint that force the arcs selected around a node to be exactly two $\sum_{e \in \delta(v)} y_e = 2 \forall v \in N$, with $\delta(v)$ the set of arcs with v in one of the two vertices. The final formulation for the ILP problem is

$$min \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c_{ij} y_{ij} \tag{3}$$

```
Solving the linear drilling TSP
Objval: 665.155

y:
    0    0    0    1    0    0    0    0    0    0
    0    0    1    0    0    0    0    0    0    0
    0    0    0    0    0    0    1    0    0    0
    0    0    0    0    0    0    0    1    0    0
    0    1    0    0    0    0    0    0    0    0
    0    0    0    0    1    0    0    0    0    0
    1    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    1
    0    0    0    0    0    1    0    0    0    0
    0    0    0    0    0    0    0    0    1    0

x:
    0    0    0    9    0    0    0    0    0    0
    0    0    2    0    0    0    0    0    0    0
    0    0    0    0    0    0    1    0    0    0
    0    0    0    0    0    0    0    8    0    0
    0    3    0    0    0    0    0    0    0    0
    0    0    0    0    4    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    7
    0    0    0    0    0    5    0    0    0    0
    0    0    0    0    0    0    0    0    6    0

Optimal solution:
0 --> 3 --> 7 --> 9 --> 8 --> 5 --> 4 --> 1 --> 2 --> 6 --> 0
in 4.32871 seconds (user time)
in 0.25 seconds (CPU time)
```

Fig.(5) Results of file "random_panels/size10/dati5.dat" with SCF formulation.

subject to:

$$
\begin{aligned}
&\sum_{i\in N} x_{ik} - \sum_{j\in N} x_{kj} = 1 && \forall k \in N \setminus \{0\} \\
&\sum_{e\in\delta(v)} y_e = 2 && \forall v \in N \\
&x_{ij} \le (n-1)y_{ij} && \forall i \in N, \forall j < i \in N \setminus \{0\} \\
&x_{ij} \le (n-1)y_{ji} && \forall i \in N, \forall j > i \in N \setminus \{0\} \\
&x_{ij} \in \mathbf{R}_+ && \forall i,j \in N, i \neq j, j \neq 0 \\
&y_{ij} = 0,1 && \forall i,j \in N,
\end{aligned}
\tag{4}
$$

where we adapted the big-M constraints to the remaining y_{ij} variables. With this formulation with are not forcing anymore the x_{ij} variables to be non-zero only when the arc (i,j) is selected with direction

$i \to j$, but also when the direction is the opposite $j \to i$. Hence we could have solutions with flow in both directions, but it doesn't matter as long as the unitary flow constraint holds, forcing the total exchange between x_{ij} and x_{ji} to be exactly one in the direction of the flow only if arc (i, j) is in the path. For this reason we won't always have a commodity of $(n-1)$ leaving node 0, but two different commodities in two directions, going from 0 to the extremes of the path, like 0 were a node internal of the path and not necessarily the first one. Then the two extremes must have an arc in the path that connects them by the come-from/go-to constraint, since all other nodes already have two arcs selected around them.
With this formulation we have halved the number of integer variables in the formulation.

2.3.1 Implementation: Part 1/Symm_SCF

As in the above case I create two map vectors for the indices of variables x_{ij} and y_{ij}, but I create only half of the y_{ij} variables

```
//------------------------------------------------------------------------------
        // CREATE VARIABLES

        // create y_{e} variables for all e in E
        const int y_init = 0;
        int current_var_position = y_init;
        for (int i = 0; i < N; i++)
        {
                for (int j = 0; j < i; j++)
                {
                        //if (i == j) continue;
                        char ytype = 'B';
                        double obj = data.C[i*N + j];
                        double lb = 0.0;
                        double ub = 1.0;
                        snprintf(name, NAME_SIZE, "y_%d_%d", i, j);
                        char* yname = (char*)(&name[0]);
                        CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &ytype, &yname );
                        map_y[i][j] = current_var_position;
            ++current_var_position;
                }
        }
//------------------------------------------------------------------------------
```

The subtour-elimination constraint is the same of above, and the others are only slightly different

```
//------------------------------------------------------------------------------
        // ADD GO-TO/FROM-YO CONSTRAINTS [ sum_{e in delta(v)}[y_{e}] = 2  forall v in N]
        nindeces = N-1;
        for (int v = 0; v < N; v++)
        {
                int curr_index = 0;
                std::vector<int> idx(nindeces);
                std::vector<double> coef(nindeces);

                for (int j = 0; j < v; j++) {
```

```
                        //if (j == i) continue;
                        idx[curr_index] = map_y[v][j];
                        coef[curr_index] = 1.0;
                        ++curr_index;
                }
                for (int i = v + 1; i < N; i++) {
                        //if (j == i) continue;
                        idx[curr_index] = map_y[i][v];
                        coef[curr_index] = 1.0;
                        ++curr_index;
                }
                char sense = 'E';
                int matbeg = 0;
                double rhs = 2.0;
                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg, &i
                &coef[0], 0, 0 );
        }
        //------------------------------------------------------------------------------
        // ADD BIG-M CONSTRAINT [ x_{ij} <= (|N| - 1) y_{ij}]

        for (int i = 0; i < N; i++) {
                for (int j = 1; j < N; j++){
                        if (i == j) continue;
                        std::vector<int> idx(2);
                        std::vector<double> coef(2);
                        idx[0]  = map_x[i][j];
                        if (i > j) idx[1]  = map_y[i][j];
                        else idx[1]  = map_y[j][i];
                        coef[0] = 1.0;
                        coef[1] = -(N - 1.0);
                        char sense = 'L';
                        int matbeg = 0;
                        double rhs = 0.0;
                        CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &ma
                        &coef[0], 0, 0 );
                }
        }
//------------------------------------------------------------------------------
```

2.3.2 Example: "random_panels/size10/dati5.dat"

Result for "random_panels/size10/dati5.dat" problem:
One can check (6) the behaviour of the flow looking at the y and x variables. For example the "3" and "6" in the first line of x tells us that there are 3 units of flow leaving to node 3 (one can check that path (0,3) is selected) and 6 leaving to node 6. The two lines with all zeros are the extremes of the path (nodes 8 and 9) and it's easy to see that $y_{98} = 1$ so that also the last arc is selected and counted in the objective function. The final path is printed always using 0 as the first node, thus reordering the cycle found.

```
Solving the linear drilling TSP
Objval: 665.155

y:

   -0
   -0    1
    1   -0   -0
   -0    1   -0   -0
   -0   -0   -0   -0    1
    1    0    1   -0   -0   -0
    0   -0   -0    1    0   -0   -0
   -0   -0   -0   -0   -0    1   -0   -0
   -0   -0   -0   -0   -0    0   -0    1    1

x:
    0    0    0    3    0    0    6    0    0    0
    0    0    0    0    3    0    0    0    0    0
    0    4    0    0    0    0    4    0    0    0
    0    0    0    0    0    0    0    2    0    0
    0    0    0    0    0    2    0    0    0    0
    0    0    0    0    0    0    0    0    1    0
    0    0    9    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    1
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0

Optimal solution:
0 --> 3 --> 7 --> 9 --> 8 --> 5 --> 4 --> 1 --> 2 --> 6 --> 0
in 1.92992 seconds (user time)
in 0.15625 seconds (CPU time)
```

Fig.(6) Results of file "random_panels/size10/dati5.dat" with Symmetric SCF formulation.

2.4 Fourth Model: Multi-Commodity Flow (MCF)

In this formulation, summarized in ([2]), we imagine that the salesman carries $n-1$ commodities, one unit for each customer. Let the y_{ij} variables be as in (SCF) formulation. For all $i, j, k \in N, i \neq j, j \neq 0$ let us define the additional continuous variables f_{ij}^k, representing the amount of the k_{th} commodity (if any) passing directly from node i to node j. The ILP formulation has then the same objective function of (SCF) and the same constraints on the y_{ij} variables, with the extra cosntraints

$$\begin{aligned}
&0 \leq f_{ij}^k \leq y_{ij} \quad \forall i, j, k \in N, k \neq 0, i \neq j \\
&\sum_{j \in N, j \neq 0} f_{1j}^k = 1 && \forall k \in N \setminus \{0\} \\
&\sum_{i \in N, i \neq k} f_{ik}^k = 1 && \forall k \in N \setminus \{0\} \\
&\sum_{i \in N, i \neq j} f_{ij}^k - \sum_{i \in N \setminus \{0\}, i \neq j} f_{ij}^k = 0 && \forall j, k \in N \setminus \{0\}, j \neq k
\end{aligned} \tag{5}$$

In order, the new constraints state that a commodity cannot flow along an edge unless that edge belongs to the tour, that each commodity leaves the depot and arrives at its destination, and the last ensures that, when a commodity arrives at a node that is not its final destination, then it also leaves that node. This formulation has $\mathcal{O}(n^3)$variables and $\mathcal{O}(n^3)$ constraints.

2.4.1 Implementation: "Part 1/MCF"

I report here only the creation of f_{ij}^k variables and relative constraints, since the rest of the code is equal to the SCF case.

```
//-------------------------------------------------------------------------------
        /*MAP FOR f VARS: initial memory allocation for map vector*/
        map_f.resize(N);
        for ( int i = 0 ; i < N ; ++i ) {
                map_f[i].resize(N);
                for ( int j = 0 ; j < N ; ++j ) {
                        map_f[i][j].resize(N);
                        for (int k = 0; k < N; ++k) {
                                map_f[i][j][k] = -1;
                        }
                }
         }
//-------------------------------------------------------------------------------
// create f_{ij}^k variables
        const int f_init = CPXgetnumcols(env, lp);
        current_var_position = f_init;
        for (int k = 1; k < N; k++)
        {
                for (int i = 0; i < N; i++)
                {
                        if (i == k) continue;
                        for (int j = 1; j < N; j++)
                        {
                        if (i == j) continue;
                                char xtype = 'C';
                                double obj = 0.0;
                                double lb = 0.0;
                                double ub = CPX_INFBOUND;
                                snprintf(name, NAME_SIZE, "f_%d_%d(%d)", i, j, k);
                                char* fname = (char*)(&name[0]);
                                CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &xtype,
                                map_f[i][j][k] = current_var_position;
                                ++current_var_position;
                        }
                }
        }
//-------------------------------------------------------------------------------
// COMMODITY LEAVES DEPOT CONSTRAINT [sum_{J in N\{0}}[f_{0 j}^k]= 1 ]
        //nindeces = N-1;
        for (int k = 1; k < N; k++)
        {
                int curr_index = 0;
                std::vector<int> idx(nindeces);
                std::vector<double> coef(nindeces);
                int i = 0;
                for (int j = 1; j < N; j++) {
                        idx[curr_index] = map_f[i][j][k];
                        coef[curr_index] = 1.0;
                        ++curr_index;
```

```
                }
                char sense = 'E';
                int matbeg = 0;
                double rhs = 1.0;
                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg, &idx[0]
                &coef[0], 0, 0 );
        }
//------------------------------------------------------------------------------
// COMMODITY REACHES DESTINATION CONSTRAINT [sum_{i in N}[f_{i k}^k]= 1 ]
        nindeces = N;
        for (int k = 1; k < N; k++)
        {
                int curr_index = 0;
                std::vector<int> idx(nindeces);
                std::vector<double> coef(nindeces);
                for (int i = 0; i < N; i++) {
                        if (i == k) continue;
                        idx[curr_index] = map_f[i][k][k];
                        coef[curr_index] = 1.0;
                        ++curr_index;
                }
                char sense = 'E';
                int matbeg = 0;
                double rhs = 1.0;
                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg, &idx[0]
                &coef[0], 0, 0 );
        }
//------------------------------------------------------------------------------
// COMMODITY LEAVES NODE DIFFERENT FROM ITS FINAL DESTINATION CONSTRAINT [sum_{i in N}[f_{i j}^k]
-[sum_{i in N \{0}}[f_{j i}^k= 0 ]
        nindeces = 2*N-1;
        for (int k = 1; k < N; k++)
        {
                for (int j = 1; j < N; j++){
                        if (j == k) continue;
                        int curr_index = 0;
                        std::vector<int> idx(nindeces);
                        std::vector<double> coef(nindeces);
                        for (int i = 0; i < N; i++) {
                                if (i == j || i == k) continue;
                                idx[curr_index] = map_f[i][j][k];
                                coef[curr_index] = 1.0;
                                ++curr_index;
                        }
                        for (int i = 1; i < N; i++) {
                                if (i == j) continue;
                                idx[curr_index] = map_f[j][i][k];
                                coef[curr_index] = -1.0;
                                ++curr_index;
                        }
                        char sense = 'E';
                        int matbeg = 0;
                        double rhs = 0.0;
                        CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg,
                        &coef[0], 0, 0 );
                }
```

```
        }
//------------------------------------------------------------------------------
// ADD CONSTRAINT [ f_{ij}^k <= y_{ij}]
        for (int k = 1; k < N; k++){
                for (int i = 0; i < N; i++) {
                        if (i == k) continue;
                        for (int j = 1; j < N; j++){
                                if (i == j) continue;
                                std::vector<int> idx(2);
                                std::vector<double> coef(2);
                                idx[0]  = map_f[i][j][k];
                                idx[1]  = map_y[i][j];
                                coef[0] = 1.0;
                                coef[1] = -1.0;
                                char sense = 'L';
                                int matbeg = 0;
                                double rhs = 0.0;
                                CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &se
                        }
                }
        }
//------------------------------------------------------------------------------
```

2.4.2 Example: "random_panels/size10/dati5.dat"

Result for "random_panels/size10/dati5.dat" problem: It's also possible to use a symmetric version of

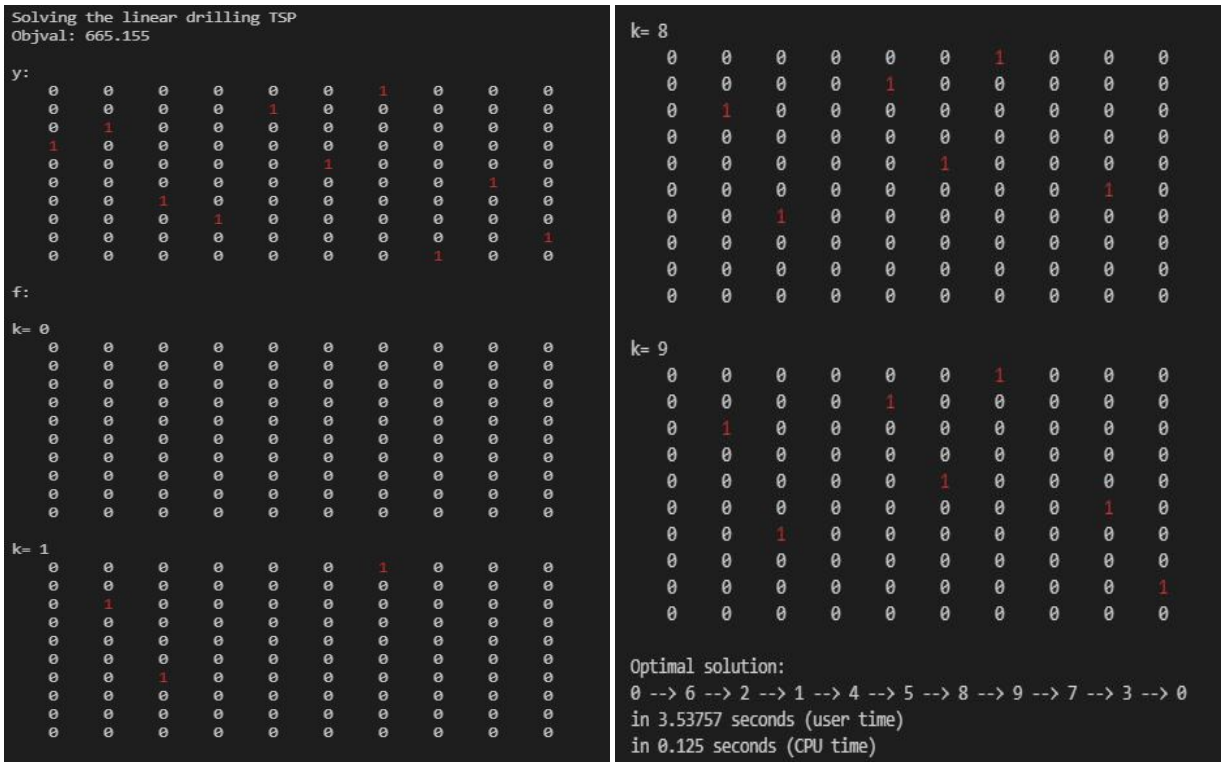

Fig.(7) Results of file "random_panels/size10/dati5.dat" with MCF formulation.

this formulation on the $y_{ij}variables$, exactly as we did for the (SCF) case (look at "Symm_MCF" folder).

2.5 Results and comparison

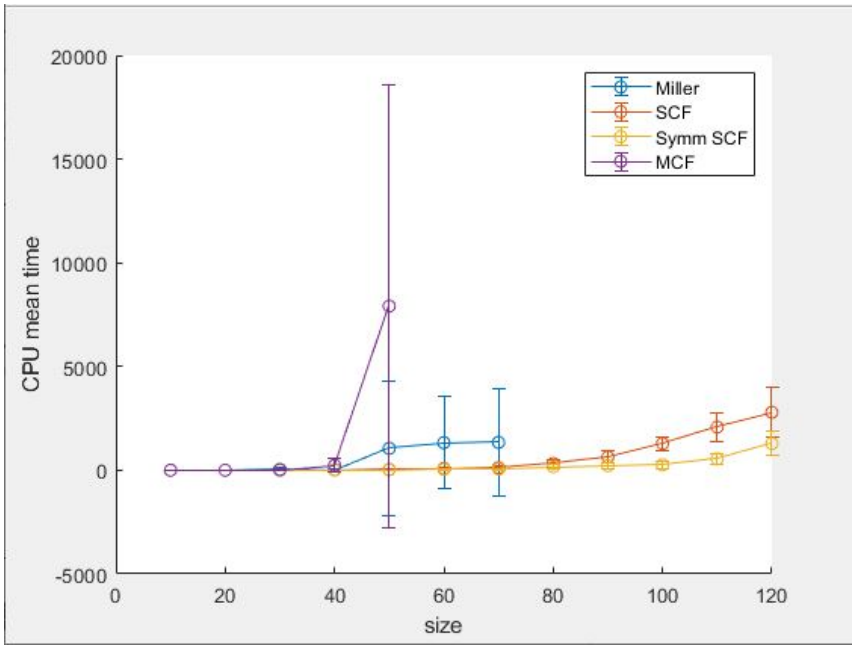

Fig.(8) Size Vs mean CPU time for random holes panels with error bars.

RANDOM_PANELS		FORMULATIONS			
SIZE	(over 15 configurations)	Miller	SCF	SCF_symm	MCF
10	Mean (s) +/- STD (s)	0.31+/-0.1	0.22+/-0.10	0.12+/-0.076	0.04+/-0.15
20	Mean (s) +/- STD (s)	1.25+/-1.42	1.3+/-0.50	0.33+/-0.33	2.81+/-2.14
30	Mean (s) +/- STD (s)	6.01+/-70.74	6.31+/-3.89	1.98+/-1.48	23.069+/-16.75
40	Mean (s) +/- STD (s)	23.96+/-18.28	11.34+/-4.72	5.23+/-2.41	235.43+/-312.01
50	Mean (s) +/- STD (s)	1055.1+/-+/-3246.	40.82+/-17.37	13.13+/-6.63	7912.36+/-10673.0
60	Mean (s) +/- STD (s)	1327.7+/-2237.49	83.89+/-56.12	37.91+/-32.58	>1e4
70	Mean (s) +/- STD (s)	1341.6+/-2614.29	122.16+/-86.06	57.29+/-39.95	>1e4
80	Mean (s) +/- STD (s)	>2e4	321.71452+/-180.62	168.79+/-109.80	>1e4
90	Mean (s) +/- STD (s)	>2e4	632.62+/-308.68	220.20+-145.48	>1e4
100	Mean (s) +/- STD (s)	>2e4	1289.73330.97	257.04+/-197.94	>1e4
110	Mean (s) +/- STD (s)	>2e4	2080+/-700.66	551.04+/-267.90	>1e4
120	Mean (s) +/- STD (s)	>2e4	2790.51+/-1182.94	1288.59+/-594.62	>1e4
			FORMULATIONS		
10 RUNS OF SIZE 50		Miller	SCF	SCF_symm	MCF
In 1 s (user time) ~ 5s (CPU)	Number of exact sols	0/10	0/10	0/10 + 2 not feasible	0/10
	average relative error(%)	(158+/-117.8) %	(218.1+/-177.4) %	(158+/-117.8) %	(372+/-31.2) %
In 10 s (user time) ~ 45s (CPU)	Number of exact sols	4/10	10/10	10/10	0/10
	average relative error(%)	(1.48+/-2.24) %	0%	0%	(372+/-31.2) %
in 40 s (user time) ~ 155s (CPU)	Number of exact sols	9/10	10/10	10/10	0/10
	average relative error(%)	(0.02+/-0.06) %	0%	0%	(372+/-31.2) %

Fig.(9) Table of results for random holes panels.

The results present an increase in CPU time for the computations of a customized panel with respect to a random one, for all methods except the MCF. In the excel table, we see how the MCF results doesn't change in 40 seconds of running times for 50 holes panels, and it's probably due to the fact that such scheme requires a longer time to create the problem, given the much bigger number of constraints and variables, and thus in this small time is not able to start efficiently the procedure, giving as output what is probably the first computed feasible solution.

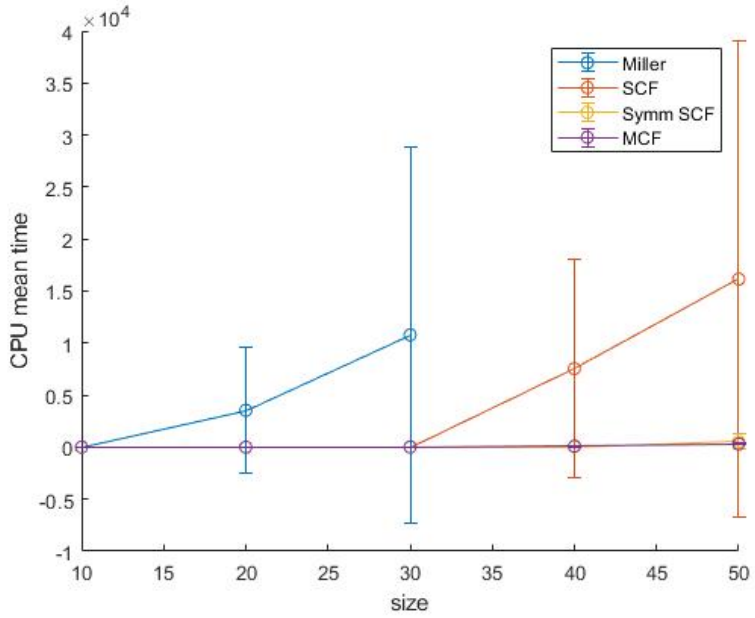

Fig.(10) Size Vs mean CPU time for custom holes panels with error bars.

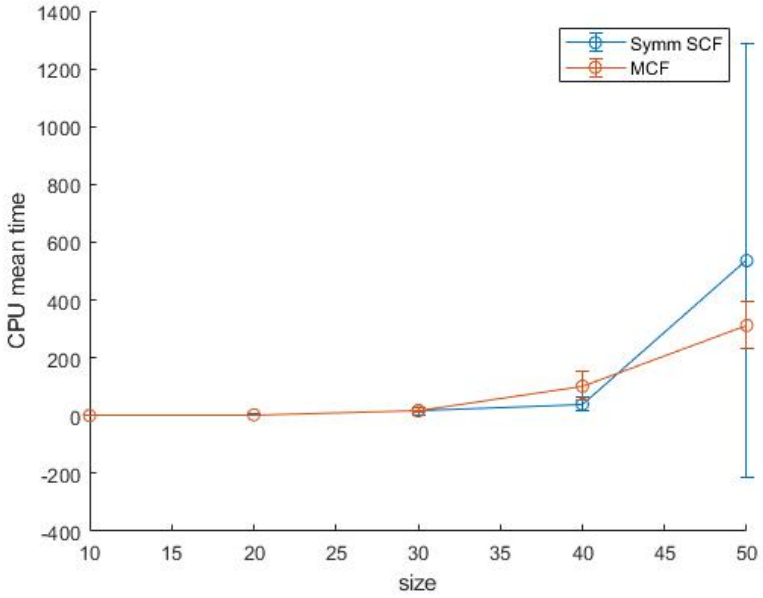

Fig.(12) Symmetric SCF Vs MCF for custom panels.

3 Part 2: Genetic Algorithm using local search

I report here the main features of the implemented Genetic algorithm. Three different classes have been used to create the model: "TSPSolution" contains the key features of each single solution (like the path, the fitness, the function that prints it to the screen, etc.); "Population" has the methods and parameters of the population (like the solutions inside the current family, the best current solution in the population, the methods to add and remove individuals from the population, etc.) and "TSPSolver" has the main functions to solve the problem.

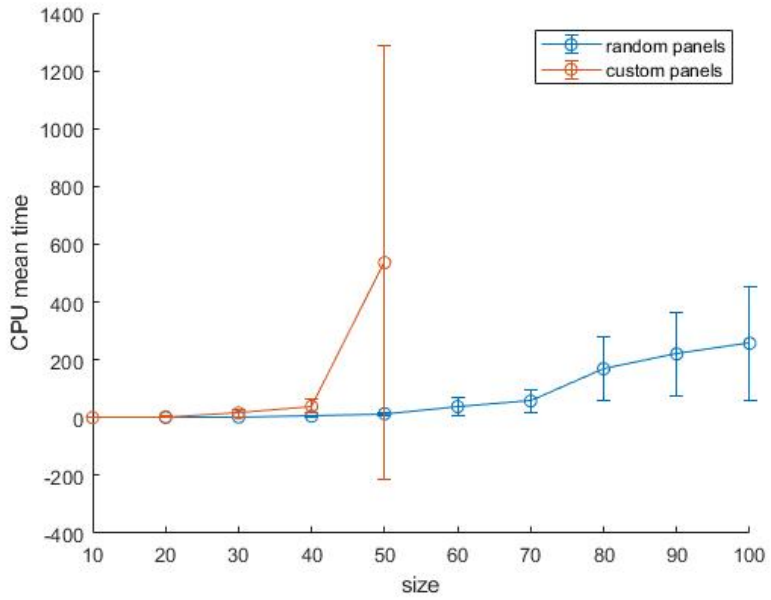

Fig.(13) Symmetric SCF, comparison between custom and random panels

CUSTOM_PANELS		FORMULATIONS			
SIZE	(over 10 configurations)	Miller	SCF	SCF_symm	MCF
10	Mean (s) +/- STD (s)	0.125+/-0.03	0.23+/-0.03	0.14+/-0.05	0.053571+/-0.02
20	Mean (s) +/- STD (s)	3539.2+/-965.38	3.96+/-3.10	2.73+/-1.97	2.5335+/-0.46
30	Mean (s) +/- STD (s)	10790.96+/-2881.05	16.34+/-9.50	15.92+/-12.00	18.109+/-7.01
40	Mean (s) +/- STD (s)	>1e4	7551+/-1044.5	39.27+/-23.55	101.77+/-49.86
50	Mean (s) +/- STD (s)	>1e4	16155.49+/-2288.9	536.18+/-750.46	312+/-81.62
			FORMULATIONS		
10 RUNS OF SIZE 50		Miller	SCF	SCF_symm	MCF
In 1 s (user time) ~ 5s (CPU)	Number of exact sols	0/10	0/10	0/10 + 2 not feasible	0/10
	average relative error(%)	(113.0+/-117.8) %	(7.66+/-8.4) %	(224.7+/-384.6) %	(18.4+/-17.8) %
In 10 s (user time) ~ 45s (CPU)	Number of exact sols	0/10	0/10	1/10	0/10
	average relative error(%)	(12.45+/-8.1) %	(0.74+/-1.7)%	(0.66+/-1.2)%	(18.4+/-17.8) %
in 40 s (user time) ~ 200s (CPU)	Number of exact sols	0/10	4/10	7/10	0/10
	average relative error(%)	(4.3+/-4.2) %	(0.04+/-0.09)%	1.00E-05	(18.4+/-17.8) %

Fig.(14) Table of results for custom holes panels.

Encoding
The solutions are encoded using a path representation (genes are the holes in order of visit).

Initial Population
The initial population is generated at random.

Selection
I choose to use a mating pool selection, with the individuals in the pool selected according to a linear ranking scheme. Below the implemented code.

```
//-----------------------------------------------------------------------------------------------------------------
std::vector<TSPSolution> TSPSolver:: SelectParents(POPULATION& Pop, const int& N_p) {
    std::vector<double> probabilities;
    std::vector<TSPSolution> parents;
    std::vector<int> ParIdx;
    parents.reserve(N_p);
    probabilities.resize(N_p);
```

```
    ParIdx.reserve(N_p);   // Indeces of the already selected parents
    for (int i_p = 0; i_p < N_p ; i_p++) {
      if (i_p == 0) LinRankProb(Pop.population, probabilities);
      else LinRankProb(Pop.population, probabilities, ParIdx);
      double rnd = ((double) rand() / (RAND_MAX));
      double corr_term = 0;
      for (int i = 0; i < Pop.N; i++) {
        if(Isin(ParIdx, i)) continue;
        if (rnd <= (probabilities[i] + corr_term)){
          parents.push_back(Pop.population[i]);
          parents[i_p].fitness = Pop.population[i].fitness;
          ParIdx.push_back(i);
          break;
        }
        corr_term += probabilities[i];
      }
    }
    return parents;
  };
```

ParIdx is a vector containing the indices in the population of the already selected parents.

Recombination
Two parents are selected with a uniform probability distribution between the mating pool for each recombination. Then two possibilities have been explored for the recombination, each one arising from specific operators which maintains feasibility.
The first one is the **Partially Mapped Crossover** (PMX) with two cuts whose indices are randomly chosen with a uniform probability.

```
//-------------------------------------------------------------------------------------------------------------
void TSPSolver :: RecombinationPMX(const std::vector<TSPSolution>& parents,
std::vector<TSPSolution>& childs, int& flag) {
  int N_child = 2;
  int N_genes = parents[0].sequence.size();
  if (flag == 1) N_child = 1;
  // chose randomly the cuts
  int cut1 = rand() % (N_genes - 2) + 1;
  int cut2 = rand() % (N_genes - 3) + 1;
  if (cut2 == cut1) cut2 = N_genes - 1;
  else if (cut2 < cut1) {
    int cut_check = cut1;
    cut1 = cut2;
    cut2 = cut_check;
  }
  for (int i_child = 0; i_child < N_child; i_child++){
    TSPSolution currChild;
    currChild.sequence.resize(parents[0].sequence.size());
    std::vector<int> MissingSpots;
    int oppParent = (2 - i_child - 1) % 2;
    for (int i_gene = cut1; i_gene < cut2; i_gene++) {
      currChild.sequence[i_gene] = parents[oppParent].sequence[i_gene];
    }
    for (int i_gene = 1; i_gene < cut1; i_gene++){
      if (!Isin(currChild.sequence, parents[i_child].sequence[i_gene])) {
```

```
        currChild.sequence[i_gene] = parents[i_child].sequence[i_gene];
      }  else MissingSpots.push_back(i_gene);
    }
    for (int i_gene = cut2; i_gene < N_genes - 1; i_gene++){
      if (!Isin(currChild.sequence, parents[i_child].sequence[i_gene])) {
        currChild.sequence[i_gene] = parents[i_child].sequence[i_gene];
        }  else MissingSpots.push_back(i_gene);
    }
    for (uint i_void = 0; i_void < MissingSpots.size(); i_void++) {
      for (int j_gene = 1; j_gene < N_genes - 1; j_gene++) {
            if (!Isin(currChild.sequence, parents[oppParent].sequence[j_gene])){
              currChild.sequence[MissingSpots[i_void]] = parents[oppParent].sequence[j_gene];
              break;
            }
      }
    }
    childs.push_back(currChild);
  }
}
//-----------------------------------------------------------------------------
```

The other method proposed is a **revised Cycle Crossover**, that we can call (CX2), proposed in the article [1]. The normal (CX) method has the benefit to create offspring in such a way that each bit with its position comes from one of the parents, but has the drawback to ends up with offsprings identical to their parents in some cases. The (CX2) extends the normal method trying to maintain the differentiation in the offsprings. For more information look at the main article (`https://www.hindawi.com/journals/cin/2017/7430125/`).

```
//-----------------------------------------------------------------------------
void TSPSolver :: RecombinationCX2(const std::vector<TSPSolution>& parents,
std::vector<TSPSolution>& childs, int& flag) {
  int N_genes = parents[0].sequence.size();
  TSPSolution Child0;
  TSPSolution Child1;
  Child0.sequence.resize(parents[0].sequence.size());
  if (flag != 1) {
    Child1.sequence.resize(parents[0].sequence.size());
  }
  int index  = 1;
  int i_gene = 1;
  int gene_add = parents[1].sequence[index];
  while (i_gene < N_genes - 1) {
    if (Isin(Child0.sequence, gene_add)){
      ++index;
      gene_add = parents[1].sequence[index];
      continue;
    }
    Child0.sequence[i_gene] = gene_add;
    int jump1 = parents[1].sequence[findIndex(parents[0].sequence, gene_add)];
    int jump2 = parents[1].sequence[findIndex(parents[0].sequence, jump1)];
    if (flag != 1) Child1.sequence[i_gene] = jump2;
    gene_add  = parents[1].sequence[findIndex(parents[0].sequence, jump2)];
    ++i_gene;
  }
```

```
  childs.push_back(Child0);
  if (flag != 1) {
    childs.push_back(Child1);
  }
}
//-----------------------------------------------------------------------------
```

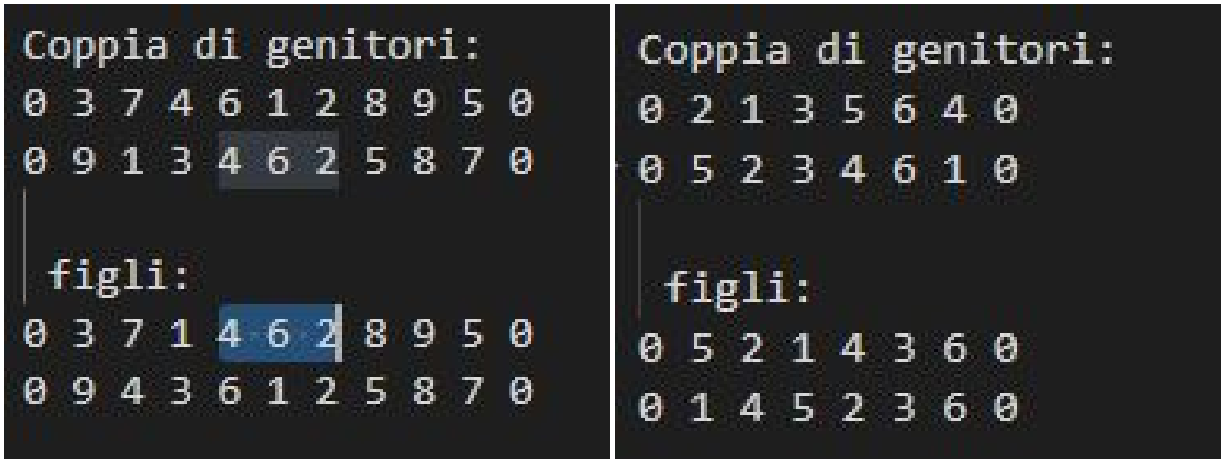

Fig.(15) (PMX) crossover (left), (CX2) crossover (right).

Generational Replacement
Starting from a Best Individuals rule, a variable Diversity threshold method has been implemented, based on Hamming distances between the solutions ([5]). With this second method all the solutions with hamming distance less than the (variable) threshold D (initially computed as the half of the maximum distance between the individuals of the initial population) are penalized by a factor $bestFitness/2$, bestFitness fitness of the current best solution. The value of D it's aimed to be initially high, to guarantee a bigger exploration of the search space, and decreases in time or in the number of non improving iterations. When the value of D is small the algorithm is almost the same of the Best Individuals rule.

```
//-------------------------------------------------------------------------------------------
 void POPULATION:: Replacement(int currNonImpr, int maxNonImpr, std::ofstream& outFile){
     std::vector<int> NewPop;
     std::vector<int> CurrPop(population.size());
     for (uint i = 0; i < population.size(); i++) CurrPop[i] = i;
     int N_genes = population[0].sequence.size();
     NewPop.reserve(N);
     NewPop.push_back(BestIdx);
     FindBestIndividual();
     CurrPop.erase(CurrPop.begin() + BestIdx);
     int currSize = NewPop.size();
     int selIdx = BestIdx;
     int eraseIdx;

     double D = D_init - D_init * maxNonImpr/ ( maxNonImpr + currNonImpr * currNonImpr);
     while (currSize < N) {
       double incumbent = tsp.infinite;
         for (uint i_ind = 0; i_ind < CurrPop.size(); i_ind++) {
             int idx = CurrPop[i_ind];
             //compute Hamming distance to the closest neighbour
             int DCN = N_genes;
             for (uint new_ind = 0; new_ind < NewPop.size(); new_ind++) {
```

```
            int DCN_new = HammingDist(population[NewPop[new_ind]].sequence, population[idx].sequenc
            if (DCN_new < DCN) DCN = DCN_new;
          }

          double currFit;
          if ( DCN < D ) currFit = population[idx].fitness + BestValue/10; //damage
          else currFit = population[idx].fitness;
          if (currFit < incumbent){
            incumbent = currFit;
            selIdx = idx;
            eraseIdx = i_ind;
          }
        }
        NewPop.push_back(selIdx);
        CurrPop.erase(CurrPop.begin() + eraseIdx);
        currSize = NewPop.size();
    }
    // population = NewPop
    for (int i = population.size(); i >= 0; i--){
      if (Isin(CurrPop, i)) population.erase(population.begin() + i);
    }
   }
//---------------------------------------------------------------------------
```

Mutations
For each gene of each solution, a probability of

```
double max_prob = mu + 1e-4; // gene maximum mutation probability
    double min_prob = 1e-4; // gene maximum mutation probability
    double curr_prob = max_prob / (sqrt(K)) + min_prob;
```

with K number of the current iteration, and mu user parameter, has been used to compute the possibility of a "swap move" between that gene and a random other one in the sequence.

Integrated Local Search
I used a First Improvement Local Search scheme to improve the 50% of the new individuals. Plus, at the end of the genetic algorithm, a Best Search Local Search is used to obtain the best possible result.

Stopping criteria
The user must provide all the three

- Time limit;
- Maximum number of non improving iterations;
- Maximum number of iterations,

or use the default values.

3.1 Automatic calibration of the parameters

The provided algorithm has mainly seven parameters:

- N_{POP} number of individuals in the population ;
- $N_{Parents}$ number of individuals in the mating pool;
- N_{Child} number of childs generated in each iteration;
- μ gene's mutation probability;
- T_{max} maximum (CPU) running time;
- NII maximum number of subsequent non improving iterations;
- itmax maximum number of iterations.

A first improvement local search scheme has been used to calibrate those parameters in order to obtain good values, if not the optimal ones, for each size of the problem. Actually, since the method has been tested to be very fast, and the total number of iterations is much less restrictive than the maximum number of non improving iterations, I calibrated only the NII parameter. The initial values and possible increments are

```
//-----------------------------------------
int N_pop      = 10;
int N_parents  = 2;
int N_childs   = 2;
double mu      = 0.0;
int Tmax       = 100;
int MaxNonImpr = 50;
int MaxIt      = 10000;
//-----------------------------------------
int delta_N    = 10;
int delta_N_p  = 2;
int delta_N_c  = 2;
double delta_mu= 0.01;
int delta_NII  = 20;
//-----------------------------------------
```

In order to make more effective the first improvement local search scheme, we test firstly the benefit of increasing the more likely parameters, N_{pop}, $N_{Parents}$ and N_{Childs}, then mu, and hence NII. Then we computed the solution of the genetic algorithm for ten files of the random holes panels, taking the mean of the relative errors ($|trueFitness - computedFitness|/trueFitness$ over ten computations for each of these files, and the the mean of the means. Hence the code tries to increase the first parameter and compute again the mean error. If the mean is smaller than the incumbent one, we update the parameters and restart the procedure, otherwise we increase the following parameter. The algorithm stops when no parameter can be increased alone to improve the solution. Then, the same procedure is applied over the customized panels.
The computations show different sub-optimal values for the random and customized panels. This is reasonable since the local search scheme implemented works better if the nodes are closed to each other in a systematic way than the case of completely random positions. If the exact solutions are not available, the same procedure to calibrate the parameters could be used minimizing the fitness values.

```
RESULT FILE

N = 10, N_p = 2, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 50

 mean_rel_err = 0.0114385

N = 20, N_p = 2, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 50

 mean_rel_err = 0.0192645

N = 10, N_p = 4, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 50

 mean_rel_err = 0.0108534

N = 20, N_p = 4, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 50

 mean_rel_err = 0.0217775

N = 10, N_p = 6, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 50

 mean_rel_err = 0.0121684

N = 10, N_p = 4, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 70

 mean_rel_err = 0.00865713

N = 20, N_p = 4, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 70

 mean_rel_err = 0.0122718

N = 10, N_p = 6, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 70

 mean_rel_err = 0.00954603

N = 10, N_p = 4, N_c = 2, mu = 0, T_max = 100, MaxNonImpr = 90

```

Fig.(16) Example of calibration output file.

3.2 Results

I tested the (PMX) and the (CX2) based Genetic algorithms over the same sets of data used for the exact methods. Actually part of this sets have been used to train the algorithm, so it would be better to employ different panels configuration, but I didn't had the time to solve other problems with the exact methods to evaluate the precision of this heuristic method. For each panel configuration the meta-heuristic algorithm developed repeats the computations ten times and takes the means of the results.
The stringent stop criterion, as said above, has resulted to be the number of consequent non improving iterations, therefore we can say that the algorithm, therefore we can see from these graphs and tables that both methods converge extremely fast to a local optimum. For both random and custom configurations, the PMX approach works better than the CX2 one in terms of mean relative errors, and even if for the

RESULT FILE

MEAN REL ERR	STD	MAX	MIN	TIME (AV)(s)	IT (AV)
0.0103363	0.00146782	0.0110702	0.00740067	0.121875	102
0.0112441	0.00405822	0.0139008	0.00504507	0.110937	98
0.00287936	0.00494063	0.012749	0.000257274	0.153125	140
0.00317123	0.00349639	0.00831932	1.42527e-06	0.221875	202
0.0128506	0.00908423	0.0227866	0.00378015	0.142187	126
0.00851996	0.000884137	0.0098705	0.00794116	0.16875	155
0.0114007	0.000211088	0.0116118	0.0111896	0.173437	169
0.000335024	0.000333031	0.000668055	1.99271e-06	0.18125	177
0.000766151	0.000530837	0.00126389	0.000137942	0.207813	211
0.00718425	0.000531045	0.00799544	0.0068366	0.115625	109

Fig.(17) Example of results output

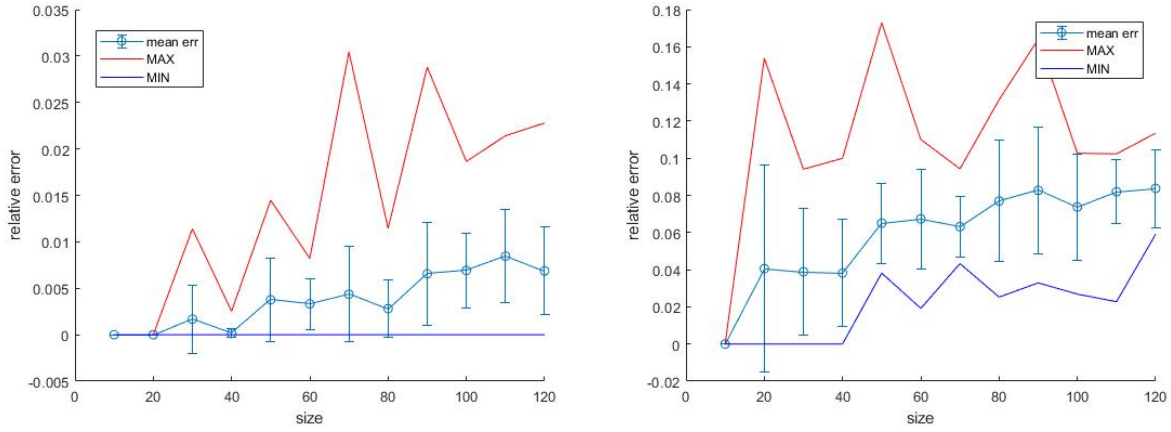

Fig.(18) Max/min/mean relative errors with PMX (left) and CX2(right) recombination methods with random panels data.

random panels the CX2 recombination leads to faster schemes (less CPU time and iterations), so that one could imagine to solve the problem introducing more differentiation in the scheme, for the customized holes panels the situation is reversed, and the PMX approach results be more efficient even in those categories.
In both cases, the methods perform way better for customized panels than for the random ones, probably due to the regular configuration of the holes which makes local search more effective.

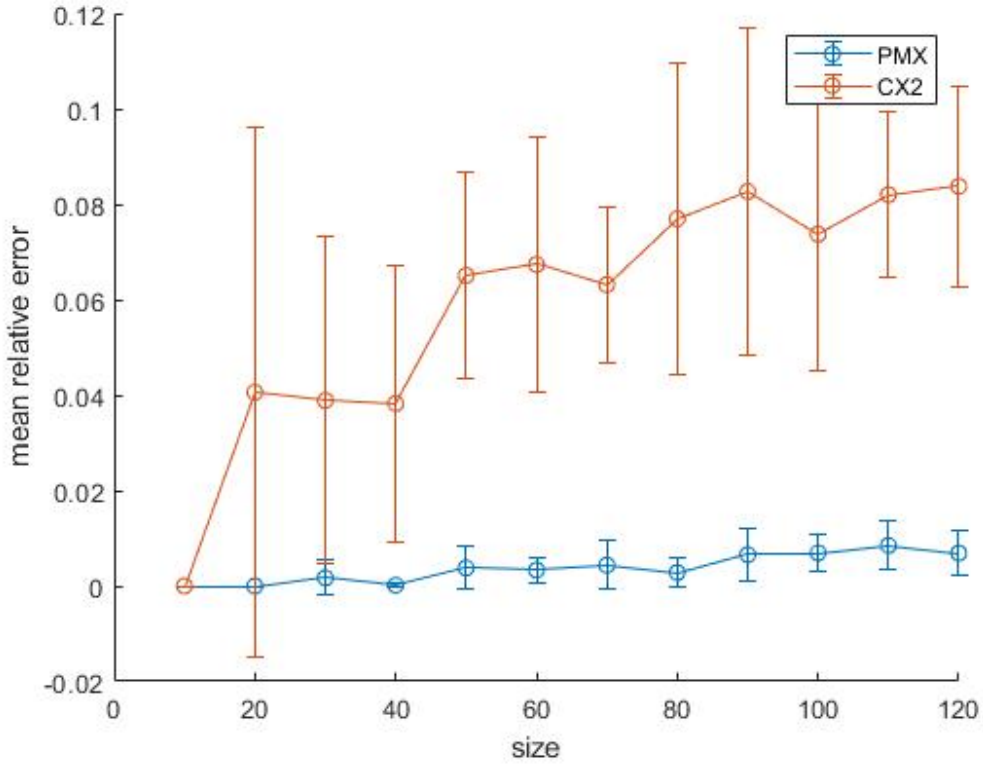

Fig.(19) Relative errors comparison with random panels.

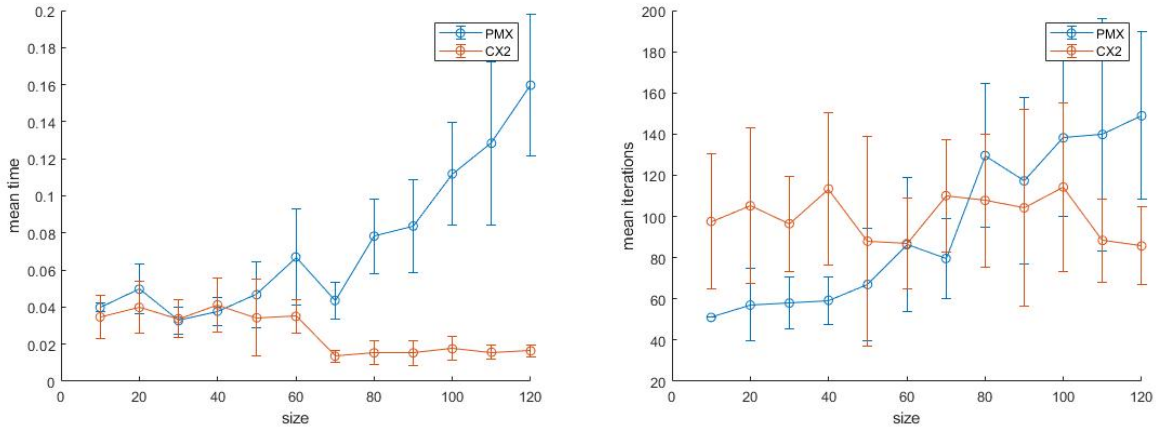

Fig.(20) Comparison between PMX and CX2 formulation in terms of CPU time (left) and iterations (right) with random panels data.

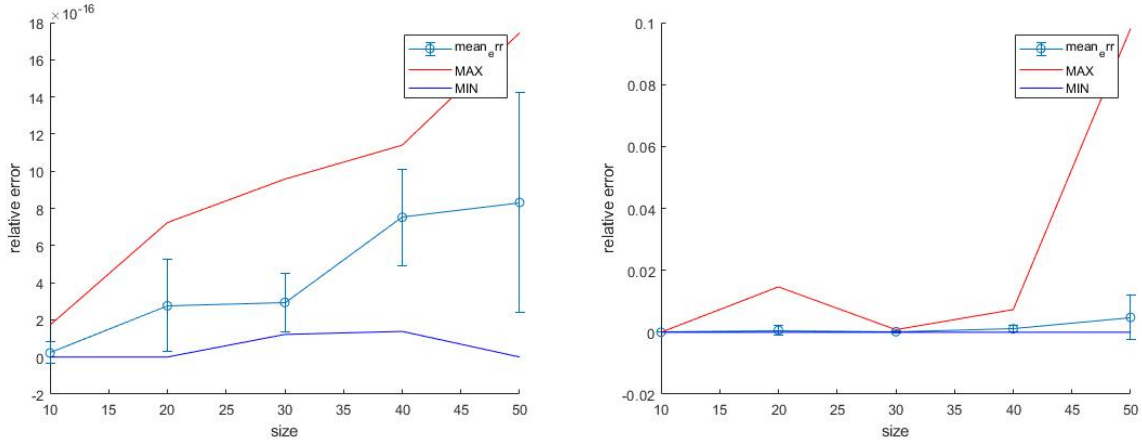

Fig.(21) Max/min/mean relative errors with PMX (left) and CX2(right) recombination methods with custom panels.

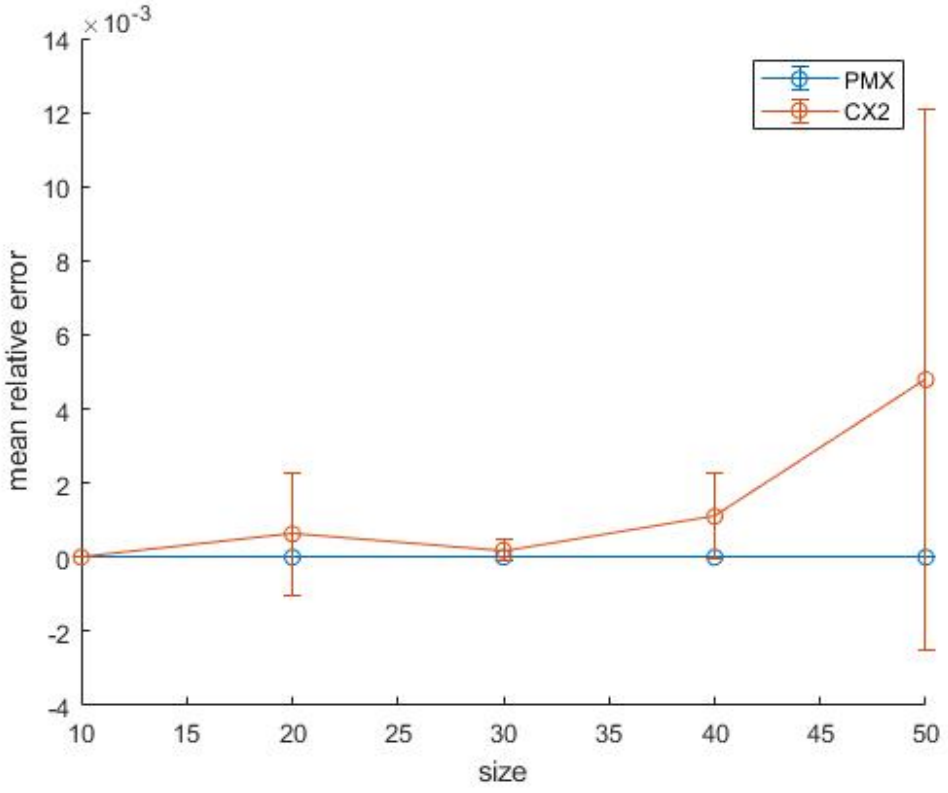

Fig.(22) Relative errors comparison with custom panels.

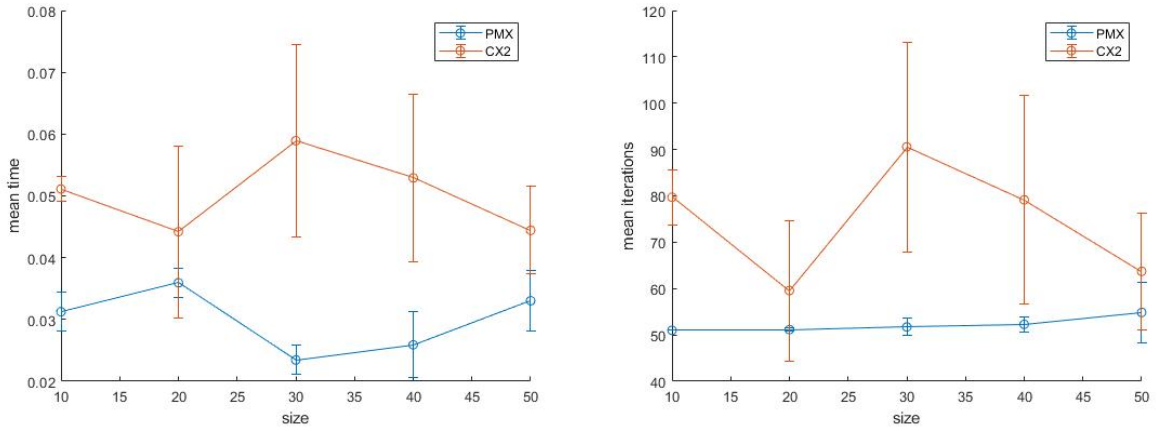

Fig.(23) Comparison between PMX and CX2 formulation in terms of CPU time (left) and iterations (right) with custom panels data.

RANDOM_PANELS		PMX				
SIZE	(over 10 configurations, each runned 10 times)	relative error (%)	Max error (%)	Min error (%)	time (CPU)	iterations
10	Mean (s) +/- STD (s)	7.45e-15+/-7.91e-15	1.64E-14	0.00E+00	3.98e-02+/-2.47e-03	51.00+/-0.00
20	Mean (s) +/- STD (s)	7.99e-15+/-6.06e-15	1.41E-14	0.00E+00	4.97e-02+/-1.34e-02	57.00+/-17.58
30	Mean (s) +/- STD (s)	1.68e-01+/-3.65e-01	1.14E+00	9.55E-05	3.27e-02+/-7.20e-03	58.00+/-12.45
40	Mean (s) +/- STD (s)	1.53e-02+/-4.78e-02	2.52E-01	1.95E-14	3.78e-02+/-7.65e-03	59.30+/-11.59
50	Mean (s) +/- STD (s)	3.76e-01+/-4.52e-01	1.45E+00	1.31E-04	4.67e-02+/-1.77e-02	67.00+/-27.28
60	Mean (s) +/- STD (s)	3.33e-01+/-2.75e-01	8.21E-01	6.29E-05	6.70e-02+/-2.59e-02	86.30+/-32.39
70	Mean (s) +/- STD (s)	4.36e-01+/-5.15e-01	3.04E+00	1.22E-04	4.36e-02+/-9.78e-03	79.60+/-19.39
80	Mean (s) +/- STD (s)	2.80e-01+/-3.12e-01	1.14E+00	5.73E-05	7.83e-02+/-2.01e-02	129.60+/-34.93
90	Mean (s) +/- STD (s)	6.56e-01+/-5.52e-01	2.88E+00	1.08E-04	8.38e-02+/-2.50e-02	117.30+/-40.34
100	Mean (s) +/- STD (s)	6.92e-01+/-4.05e-01	1.87E+00	5.14E-05	1.12e-01+/-2.75e-02	138.20+/-38.23
110	Mean (s) +/- STD (s)	8.46e-01+/-5.03e-01	2.14E+00	2.15E-04	1.28e-01+/-4.41e-02	139.80+/-56.38
120	Mean (s) +/- STD (s)	6.87e-01+/-4.71e-01	2.28E+00	1.43E-04	1.60e-01+/-3.80e-02	148.90+/-40.68

		CX2				
SIZE	(over 10 configurations, each runned 10 times)	relative error (%)	Max error (%)	Min error (%)	time (CPU)	iterations
10	Mean (s) +/- STD (s)	5.96e-15+/-7.70e-15	1.57E-14	0.00E+00	3.47e-02+/-1.17e-02	97.60+/-32.99
20	Mean (s) +/- STD (s)	4.06e+00+/-5.57e+00	1.54E+01	0.00E+00	3.98e-02+/-1.41e-02	105.20+/-37.61
30	Mean (s) +/- STD (s)	3.89e+00+/-3.42e+00	9.41E+00	3.68E-04	3.38e-02+/-1.03e-02	96.50+/-23.14
40	Mean (s) +/- STD (s)	3.82e+00+/-2.90e+00	1.00E+01	3.46E-04	4.13e-02+/-1.45e-02	113.40+/-36.93
50	Mean (s) +/- STD (s)	6.51e+00+/-2.15e+00	1.73E+01	3.82E+00	3.42e-02+/-2.08e-02	88.00+/-50.71
60	Mean (s) +/- STD (s)	6.74e+00+/-2.69e+00	1.10E+01	1.92E+00	3.50e-02+/-9.03e-03	86.80+/-22.01
70	Mean (s) +/- STD (s)	6.31e+00+/-1.63e+00	9.43E+00	4.33E+00	1.36e-02+/-3.30e-03	110.10+/-27.14
80	Mean (s) +/- STD (s)	7.69e+00+/-3.26e+00	1.31E+01	2.53E+00	1.53e-02+/-6.54e-03	107.80+/-32.26
90	Mean (s) +/- STD (s)	8.27e+00+/-3.42e+00	1.64E+01	3.29E+00	1.53e-02+/-6.78e-03	104.20+/-47.67
100	Mean (s) +/- STD (s)	7.37e+00+/-2.84e+00	1.03E+01	2.68E+00	1.75e-02+/-6.41e-03	114.30+/-41.04
110	Mean (s) +/- STD (s)	8.20e+00+/-1.72e+00	1.02E+01	2.28E+00	1.56e-02+/-3.90e-03	88.40+/-20.12
120	Mean (s) +/- STD (s)	8.38e+00+/-2.10e+00	1.13E+01	5.91E+00	1.64e-02+/-3.31e-03	85.70+/-18.83

Fig.(24) Results for random panels.

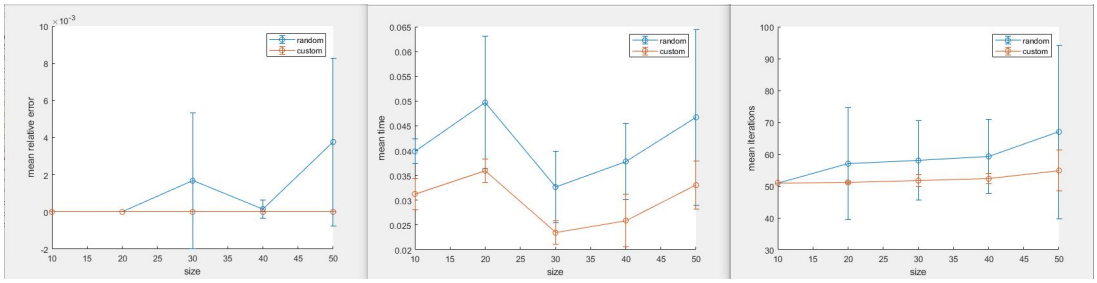

Fig.(25) Custom versus random panels with PMX recombination.

3.3 Some ideas for possible improvements

Since the running times have been proved to be quite low for both formulations, even for problems of large size (see below "a280.tsp" or "pcb442.tsp"), it meay be worth to introduce more diversification in the method and explore more regions of the search space.
Another good improvement to this algorithm would be to choose an appropriate set of initial solutions in the population, avoiding to use completely random initial populations. For example we could use a local search scheme over some initial solutions, trying to use the hamming distance to penalize the objective function in case of too similar sequences. In this way we would obtain both good initial individuals to recombine, and check the diversity.
Another extremely interesting possibility uses a k-opt optimization over the holes to visit, creating firstly smaller optimal sequences over good neighborhood of the centroids, and then connecting the sequences with some random swap move. This approach would probably be particularly effective for out problem, in which most of the holes are near to each other in square or line formation. [7].

4 Conclusion and final considerations

Finally, we test the the SCF-symm, GA+PMX and GA+CX2 with three problems of the library TSPLIB [6], "d198.tsp", "a280.tsp" and "pcb442.tsp". As a first thing we train the GA algorithms over ten customized panels of size 200, 280 and 440 to calibrate the parameters over the fitness. Below the results It seem safe to say that for large sets of data (number of holes > 150) the best solution for a quick answer

	SCF_symm		GA+PMX		GA+CX2	
	(100 s)	(max 10000 s)	(mean over 10 runs +/- std)	min/max	(mean over 10 runs +/- std)	min/max
relative error (%)	no solution	12%	1.11+/-0.64 % (0.25 s - 297 iter)	0.71 %/3 %	5.14+/-1e-5 % (0.06 s - 112 iter)	5.14 %/5.14 %
relative error (%)	no solution	solved (7680s)	3.82+/-1.35 % (0.74 s - 278 iter)	1.56 %/5.63%	10.59+/-0.9 % (0.07 s - 65 iter)	7.9 %/10.9 %
relative error (%)	no solution	15%	1.67+/-0.38 % (16.1 s - 349 iter)	1.17 %/2.6 %	10.44+/-0.45 % (0.18 s - 71 iter)	10.92 %/ 9.7 %

Fig.(26) Comparison between the formulations over three problems of TSPLIB.

would be the GA+PMX approach. We see that even the worst cases have still small relative errors from the true solution, so it seems sufficiently reliable, even if one should test the results on this large sizes on more sets of data and not just one. If instead the company has more time and could wait for an exact solution the SCF-symm method has shown to be the faster one, but it may still require days or weeks of CPU time for very large sets of data, since its CPU cost grows exponentially with the size of the problem. Moreover in most cases the SCF-symm algorithm cannot provide any feasible solutions in short running times.

Bibliography

[1] M. Nauman Sajid Abid Hussain Yousaf Shad Muhammad. *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator*. `https://www.hindawi.com/journals/cin/2017/7430125/`. 2017.

[2] Dirk Oliver Theis Adam N. Letchford Saeideh D. Nasiri. "Compact formulations of the Steiner Traveling Salesman Problem and related problems". In: (2013).

[3] Graves S. Gavish B. "The travelling salesman problem and related problems". In: (1978).

[4] MATLAB. *version R2021b*. Natick, Massachusetts: The MathWorks Inc., 2021.

[5] Aguirre Arturo Segura Carlos Rionda Botello Salvador. "A Novel Diversity-based Evolutionary Algorithm for the Traveling Salesman Problem, p.492". In: (2017).

[6] *TSPLIB*. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`.

[7] Deyun Zhou Yang Liu. *An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP*. `https://www.hindawi.com/journals/mpe/2015/212794/`. 2015.