3 Set up Cplex & Model construction

CPLEX requires a number of internal data structures in order to execute properly. These data structures must be initialized before any call to the CPLEX Callable Library.
The first call to the CPLEX Callable Library is always to the routine CPXopenCPLEX.This routine returns a pointer to the CPLEX environment. At first, the CPLEX environment was initialized through Callable Library with

```
CPXENVptr env = CPXopenCPLEX(&error);
```

After setting up our development environment we have to instantiate the problem object calling the routine CPXcreateprob. CPLEX allows us to create more than one problem object, although typical applications will use only one. Each problem object is referenced by a pointer returned by CPXcreateprob and represents one specific problem instance. So we give the statement:

```
CPXLPptr lp = CPXcreateprob(env, &error, "TSP");
```

The problem object must be destroyed at the end of the algorithm calling CPXfreeprob:

```
CPXfreeprob(env, &lp);
```

After all calls to the Callable Library are complete, the environment is released by the routine:

```
CPXcloseCPLEX(&env);
```

This routine specifies to CPLEX that all calls to the Callable Library are complete, any memory allocated by CPLEX is returned to the operating system, and the use of CPLEX is ended for this run.

3.1 Function xpos

CPLEX provides several options for entering problem data and supports the industry-standard MPS (Mathematical Programming System) file format as well as CPLEX LP format, a row-oriented format. The optimizer software stores variables in a vector so before inserting them we need a function to link the variable coefficient inside CPLEX with its indexes in order to print variables for an eventually debug. Do this and create a function called *xpos*; see Figure 2.

Because of our model doesn't use variables x_{ii} and duplicated variables with index $j \leq i$, we don't need a complete matrix of variables. The index of every variable inside CPLEX is

$$i * n + j - \frac{(i+1)*(i+2)}{2}$$

where n represents the number of nodes of the graph.
So the code is below

```
int xpos(int i, int j, instance *inst) {
    if (i > j) return xpos(j, i, inst);
   return i * inst->nnodes + j - ((i + 1)*(i + 2) / 2);
}
```

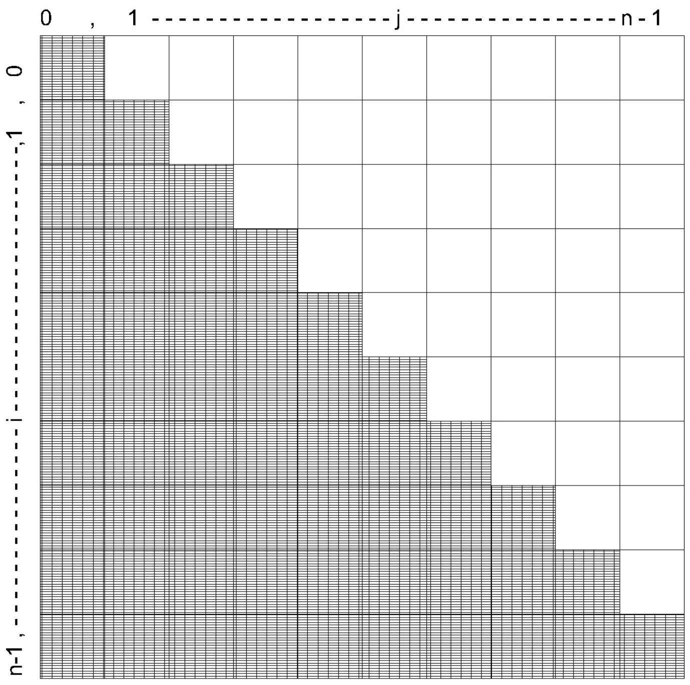
0 , 1 ----------------j---------------n-1
0 , 1 -------------i--------------- n-1 ,

Figure 2: Structure of variables inside the optimizer

3.2 Creation of the model

The problem object instantiated by CPXcreateprob represents an empty problem that contains no data. It has zero constraints, zero variables, and an empty constraint matrix. This empty problem object must be populated with data. This step can be carried out in several ways.

- The problem object can be populated by assembling arrays of data and then calling appropriate CPX functions to copy the data into the problem object.
- Alternatively, one can populate the problem object by sequences of calls to the routines CPXnewcols, CPXnewrows, CPXaddcols, CPXaddrows, and CPXchgcoeflist.
- If the data already exist in a file using MPS format or LP format, you can use CPXreadcopyprob to read the file and copy the data into the problem object.

3.3 Insertion of the variables

We first called CPXnewcols to specify the column-based problem data, such as the objective, bounds, and variables names. To insert the model in CPLEX we do:

```
 double lb = 0.0;
char binary = 'B';
char integer = 'I';

 char **cname = (char **)calloc(1, sizeof(char *));
cname[0] = (char *)calloc(100, sizeof(char));
```

```
for (int i = 0; i < inst->nnodes; i++)
{
    for (int j = i + 1; j < inst->nnodes; j++)
    {
        double obj = dist(i, j, inst);
        sprintf(cname[0], "x(%d,%d)", i + 1, j + 1);
        double ub = 1.0;
        if (CPXnewcols(env, lp, 1, &obj, &lb, &ub, &binary, cname)) print_error(" wrong
            CPXnewcols on x var.s");
        if (CPXgetnumcols(env, lp) - 1 != xpos(i, j, inst)) print_error(" wrong position for x
            var.s");
    }
}
```

We use CPXnewcols to insert, at first, the variables and then we check if our function **xpos** work right with the command CPXgetnumcols. The mathematical formulation is the following

$$\sum_{\substack{i,j\\i<j}} c_{i,j} x_{i,j}$$
$$0 \le x_{ij} \le 1 \qquad \forall i, j\ , \quad i < j$$

3.4 Insertion of constraints

To insert a new constraint we make use of CPXnewrows with:

```
for (int h = 0; h < inst->nnodes; h++)
{
    int lastrow = CPXgetnumrows(env, lp);
    double rhs = 1.0;
    char sense = 'E';
    sprintf(cname[0], "deg(%d)", h + 1);
    if (CPXnewrows(env, lp, 1, &rhs, &sense, NULL, cname)) print_error(" wrong
        CPXnewrows");
    for (int i = 0; i < inst->nnodes; i++)
    {
        if (i == h) continue;
        if (CPXchgcoef(env, lp, lastrow, xpos(i, h, inst), 1.0)) print_error(" wrong
            CPXchgcoef");
    }
}
```

The mathematical formulation related to the constraints is the following

$$\sum_{e\in\delta(h)} x_e = 1 \qquad \forall\, h \in N$$

Rhs specifies the number on the right side of equation while the variable **sense** indicates the type of expression (in this case it represents an equation).
We change all coefficient values in every row to create the single constraint with the instruction CPXchgcoef. The vector **cname** is used in particular to print a short description to all elements in the lp file where we can see all variables and all constraints inserted in the optimizer model.

3.5 Debugging of a model

You can write the problem to a CPLEX LP file (named model.lp) by calling the routine CPXwriteprob. This file can be examined to detect whether any errors occurred in the routines creating the problem. In our case used this command to print our model in a text file and to check possible problems in the formulation. If we have the optimal solution available, we can manually write it in the lp file after Subject To:

```
x(4,6)  = 1
x(6,15) = 1
.
.
.
.
.
x(23,4) = 1
```

Then, to detect possible bugs, we have two cases:

1. If the solution cost does't match the expected one, then the way in which we compute costs is incorrect.
2. If CPLEX says that the problem is impossible, then we were wrong in building the model.