Investigacion Operativa
Coloreo Particionado de Grafos

28 de noviembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Andrew Ab	???	???

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Resumen: ???

Keywords: ???

Índice

1. Modelo

Dado un grafo $G(V, E)$ con $n = |V|$ vertices y $m = |E|$ aristas, un coloreo de G se define como una asignacion de un color o etiqueta a cada $v \in V$ de forma tal que para todo par de vertices adyacentes $(p, q) \in E$ poseen colores distintos. El clasico problema de *coloreo de grafos* consiste en encontrar un coloreo del grafo que utilize la menor cantidad de colores posibles.

En este trabajo resolveremos una variante de este problema, el *coloreo particionado de grafos*. A partir de un conjunto de vertices V que se encuentra particionado en $V_1, ..., V_k$, el problema consiste en asignar un color $c \in C$ a solo un vertice de cada particion de forma tal que dos vertices adyacentes no reciban el mismo color y minimizando la cantidad de colores utilizados.

Este problema se puede modelar con Programacion Lineal Entera. Para ello, definamos las siguientes variables:

$$x_{pj} = \begin{cases} 1 & \text{si el color } j \text{ es asignado al vertice } p \\ 0 & \text{en caso contrario} \end{cases}$$

$$w_j = \begin{cases} 1 & \text{si } x_{pj} = 1 \text{ para algun vertice } p \\ 0 & \text{en caso contrario} \end{cases}$$

1.1. Funcion objetivo

De esta forma la funcion objetivo del LP consiste en minimizar la cantidad de colores utilizados:

$$min \sum_{j \in C} w_j \tag{1}$$

Notar que $|C|$ esta acotado superiormente por la cantidad de particiones k.

1.2. Restricciones

Los vertices adyacentes no comparten color. Recordar que no necesariamente se le asigna un color a todo vertice.

$$x_{ij} + x_{kj} \leq 1 \quad \forall (i, k) \in E, \ \ \forall j \in C \tag{2}$$

Solo se le asigna un color a un unico vertice de cada particion $p \in P$. Esto implica que cada vertice tiene a lo sumo solo un color.

$$\sum_{i \in V_p} \sum_{j \in C} x_{ij} = 1 \quad \forall p \in P \tag{3}$$

Si un nodo usa color j, $w_j = 1$:

$$x_{ij} \leq w_j \quad \forall i \in V, \forall j \in C \tag{4}$$

Integralidad y positividad de las variables:

$$x_{ij} \in \{0, 1\} \quad \forall i \in V, \forall j \in C \tag{5}$$

$$w_j \in \{0, 1\} \quad \forall j \in C \tag{6}$$

2. Branch & Bound

La implementacion del modelo y del Branch & Bound se encuentran en el apendice.

3. Desigualdades

3.1. Desigualdad de Clique

Sea $j_0 \in \{1, ..., n\}$ y sea K una clique maximal de G. La desigualdad clique estan definida por:

$$\sum_{p \in K} x_{pj_0} \leq w_{j_0} \tag{7}$$

Demostración Para esta demostracion utilizaremos las desigualdades Chvátal-Gomory sobre las restricciones del LP planteado en la seccion 1.2 e induccion. A priori el teorema es bastante intuitivo. Si pinto algun vertice de una clique, no puedo pintar ninguno adyacente del mismo color sin importar la forma en la que particione los vertices del grafo. Sea n el tamanio de la clique maximal.

Casos Base

1. $n = 1$: Si en la clique maximal tengo solo un vertice, no existe arista que contenga este vertice, caso contrario la clique tendria dos elementos. Por lo tanto, este vertice puede estar pintado o no dentro de la particion. Es decir, se cumple la ecuacion que queremos probar.

2. $n = 2$: Si la clique maximal tiene dos elementos, por definicion son conexos. Por la restriccion que indica que los vertices adyacentes no comparten color, aqui hay 2 opciones. La primera opcion es que a ningun vertice se le asigna un color j_0. La otra opcion es que dada la estructura de particiones, se le asigne solo a uno de ellos el color j_0. Por lo tanto la desigualdad para $n = 2$ vale.

3. $n = 3$: Este es el caso mas interesante en el que utilizamos la desigualdad de Chvátal-Gomory. Si la clique tiene 3 vertices, hay tres desigualdades que se deben cumplir:

   - $x_{1j_0} + x_{2j_0} \leq 1$
   - $x_{2j_0} + x_{3j_0} \leq 1$
   - $x_{1j_0} + x_{3j_0} \leq 1$

   Multiplicando todas estas desigualdades por $1/3$ y sumando entonces:
   $1/3(x_{1j_0} + x_{2j_0}) + 1/3(x_{2j_0} + x_{3j_0}) + 1/3(x_{2j_0} + x_{3j_0}) \leq 3/2$
   Como x_{ij} toma valores enteros, entonces: $1/3(x_{1j_0} + x_{2j_0}) + 1/3(x_{2j_0} + x_{3j_0}) + 1/3(x_{2j_0} + x_{3j_0}) \leq 1$
   Simplificando: $x_{1j_0} + x_{2j_0} + x_{3j_0} \leq 1$.
   Utilizando la definicion de w_j entonces: $x_{1j_0} + x_{2j_0} + x_{3j_0} \leq w_{j_0}$
   Por lo tanto la desigualdad vale para $n = 3$.

Paso Inductivo: $P(n-1) \implies P(n)$
Como vale la hipotesis inductiva, sabemos que:

$$\sum_{p \in K-n} x_{pj_0} \leq w_{j_0}$$

Al agregar un vertice a la clique, agregamos $n - 1$ aristas:
$x_{1j_0} + x_{nj_0} \leq 1$, $x_{2j_0} + x_{nj_0} \leq 1$,..., $x_{(n-1)j_0} + x_{nj_0} \leq 1$
Utilizando esto, podemos ver que:

$$x_{nj_0} + \sum_{p \in K-n} x_{pj_0} \leq w_{j_0}$$

Esto es claramente equivalente a lo que queremos demostrar y se puede justificar a partir de dos casos:

- Si al vertice x_{nj_0} se le asigna un color, por las restricciones de las aristas que agregamos al resto de los vertices de la clique no se le puede asignar el color j_0.

- Si al vertice x_{nj_0} no se le asigna un color o se le asigna un color diferente a j_0, por hipotesis inductiva sabemos que lo que queremos probar vale. $\square$

3.2. Desigualdad de Aujero Impar

Sea $j_0 \in \{1, ..., n\}$ y sea $C_{2k+1} = v_1, ..., v_{2k+1}$, $k \geq 2$, un aujero de longitud impar. La desigualdad esta definida por:

$$\sum_{p \in C_{2k+1}} x_{pj_0} \leq k w_{j0} \tag{8}$$

Demostración Por teoremas de coloreo (que se prueban en general por induccion), sabemos que el numero cromatico $\chi(C) = 3$. En el peor de los casos, cada vertice del aujero estara en una particion diferente. Aqui nuevamente tenemos dos casos:

- Si no se asigna el color j_0 a algun vertice del aujero, la desigualdad vale.
- Si se asigna el color j_0, en el peor de los casos el mismo sera utilizado por a lo sumo $(|C| - 1)/2$ vertices. Como $|C| = 2k + 1$, $(2k + 1 - 1)/2 = k$. Por lo tanto vale la desigualdad. □

3.3. Planos de Corte

Luego de relajar el PLEM, los algoritmos de separacion buscan acotar el espacio de busqueda para que se parezca mas a la capsula convexa. Existen algoritmos de separacion exactos y heuristicos. Los algoritmos heuristicos, luego de resolver la relajacion del problema entero y encontrar una solucion optima x^*, retornan una o mas desigualdades de la clase violadas por alguna familia de desigualdades. Por ser un algoritmo heuristico, es posible que exista una desigualdad de la clase violada aunque el procedimiento no sea capaz de encontrarla. Si se encuentra una desigualdad que es violada por la solucion optima de la relajacion, se agrega esta nueva restriccion y se vuelve a resolver el programa lineal. Este procedimiento se conoce como algoritmo de plano de corte. Si una solucion optima al problema existe, este tipo de algoritmo no necesariamente la encuentra. Por ejemplo, las heuristicas que encuentran desigualdades validas pueden fallar y el algoritmo no puede continuar.

3.3.1. Heuristica de Separacion para Clique

3.3.2. Heuristica de Separacion para Aujero Impar

4. Cut & Branch

5. Experimentacion

6. Conclusion

7. Apéndice A: Código

7.1. coloring.cpp

```cpp
#include <ilcplex/ilocplex.h>
#include <ilcplex/cplex.h>

#include <stdlib.h>

#include <string>
#include <vector>

#define TOL 1e-05

ILOSTLBEGIN // macro to define namespace

struct edge {
    int from;
    int to;

    edge(int a, int b) {
        from = a;
        to = b;
    }
};

int getVertexIndex(int id, int color, int partition_size);
int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size);
int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, vector<edge>& edges, int edge_size, int partition_size);
int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector<int> >& partitions, int partition_size);
int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size);
int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int partition_size);
int setBranchAndBoundConfig(CPXENVptr& env);

// colors array!
const char* colors[] = {"Blue", "Red", "Green", "Yellow", "Grey", "Green", "Pink", "AliceBlue","AntiqueWhite","Aqua","Aquamarine","Azure","Beige",
"Bisque","Black","BlanchedAlmond","BlueViolet","Brown","BurlyWood","CadetBlue","Chartreuse","Chocolate","Coral","CornflowerBlue",
"Cornsilk","Crimson","Cyan","DarkBlue","DarkCyan","DarkGoldenRod","DarkGray","DarkGrey","DarkGreen","DarkKhaki","DarkMagenta","DarkOliveGreen",
"Darkorange","DarkOrchid","DarkRed","DarkSalmon","DarkSeaGreen","DarkSlateBlue","DarkSlateGray","DarkSlateGrey","DarkTurquoise",
"DarkViolet","DeepPink","DeepSkyBlue","DimGray","DimGrey","DodgerBlue","FireBrick","FloralWhite","ForestGreen","Fuchsia",
"Gainsboro","GhostWhite","Gold","GoldenRod","Gray","GreenYellow","HoneyDew","HotPink","IndianRed","Indigo",
"Ivory","Khaki","Lavender","LavenderBlush","LawnGreen","LemonChiffon","LightBlue","LightCoral","LightCyan","LightGoldenRodYellow",
"LightGray","LightGrey","LightGreen","LightPink","LightSalmon","LightSeaGreen","LightSkyBlue","LightSlateGray","LightSlateGrey",
```

```
"LightSteelBlue","LightYellow","Lime","LimeGreen","Linen","Magenta","Maroon","MediumAquaMarine","MediumBlue","MediumOrchid",
"MediumPurple","MediumSeaGreen","MediumSlateBlue","MediumSpringGreen","MediumTurquoise","MediumVioletRed","MidnightBlue",
"MintCream","MistyRose","Moccasin","NavajoWhite","Navy","OldLace","Olive","OliveDrab","Orange","OrangeRed","Orchid",
"PaleGoldenRod","PaleGreen","PaleTurquoise","PaleVioletRed","PapayaWhip","PeachPuff","Peru","Plum","PowderBlue",
"Purple","RosyBrown","RoyalBlue","SaddleBrown","Salmon","SandyBrown","SeaGreen","SeaShell","Sienna","Silver","SkyBlue",
"SlateBlue","SlateGray","SlateGrey","Snow","SpringGreen","SteelBlue","Tan","Teal","Thistle","Tomato","Turquoise","Violet",
"Wheat","White","WhiteSmoke","YellowGreen"};

int main(int argc, char **argv) {

    if (argc != 2) {
        printf("Usage: %s inputFile\n", argv[0]);
        exit(1);
    }

    /* read graph input file
     * format: http://mat.gsia.cmu.edu/COLOR/instances.html
     * graph representation chosen in order to load the LP easily.
     * - vector of edges
     * - vector of partitions
     */
    FILE* fp = fopen(argv[1], "r");

    if (fp == NULL) {
        printf("Invalid input file. \n");
        exit(1);
    }

    char buf[100];
    int vertex_size, edge_size;

    vector<edge> edges;

    while (fgets(buf, sizeof(buf), fp) != NULL) {
        if (buf[0] == 'c') continue;
        else if (buf[0] == 'p') {
            sscanf(&buf[7], "%d %d", &vertex_size, &edge_size);
            // printf("vertex_size: %d, edge_size: %d \n", vertex_size, edge_size);
            // printf("Adding edges! \n");
        }
        else if (buf[0] == 'e') {
            int from, to;
            sscanf(&buf[2], "%d %d", &from, &to);
            // printf("Edge: (%d,%d) \n", from, to);
            edges.push_back(edge(from, to));
        }
    }

    // set random seed
    srand(time(NULL));

    // asign every vertex to a partition
    int partition_size = rand() % vertex_size;
```

```
    vector<vector<int> > partitions(partition_size, vector<int>());

    // warning: this procedure doesn't guarantee every partition will have an element.
    for (int i = 1; i <= vertex_size; ++i) {
        int assign_partition = rand() % partition_size;
        partitions[assign_partition].push_back(i);
    }

    // update partition_size
    for (std::vector<vector<int> >::iterator it = partitions.begin(); it != partitions.end(); ++it) {
        if (it->size() == 0) --partition_size;
    }

    // start loading LP using CPLEX
    int status;
    CPXENVptr env; // pointer to enviroment
    CPXLPptr lp;   // pointer to the lp.

    env = CPXopenCPLEX(&status); // create enviroment

    if (env == NULL) {
        printf("Error creating enviroment.\n");
        exit(1);
    }

    // create LP
    lp = CPXcreateprob(env, &status, "Instance of partitioned graph coloring.");

    if (lp == NULL) {
        printf("Error creating the LP.\n");
        exit(1);
    }

    loadObjectiveFunction(env, lp, vertex_size, partition_size);
    loadAdyacencyColorRestriction(env, lp, edges, edge_size, partition_size);
    loadSingleColorInPartitionRestriction(env, lp, partitions, partition_size);
    loadAdyacencyColorRestriction(env, lp, vertex_size, partition_size);

    // write LP formulation to file, great to debug.
    status = CPXwriteprob(env, lp, "graph.lp", NULL);

    if (status) {
        printf("Problem writing LP problem to file.");
        exit(1);
    }

    setBranchAndBoundConfig(env);
    solveLP(env, lp, edge_size, vertex_size, partition_size);

    return 0;
}

int getVertexIndex(int id, int color, int partition_size) {
    return partition_size + ((id-1)*partition_size) + (color-1);
}
```

```
int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size) {

    // load objective function
    int n = partition_size + (vertex_size*partition_size);
    double *objfun  = new double[n];
    char     *ctype = new char[n];
    char **colnames = new char*[n];

    for (int i = 0; i < partition_size; ++i) {
        objfun[i] = 1;
        ctype[i]    = CPX_BINARY;
        colnames[i] = new char[10];
        sprintf(colnames[i], "w_%d", (i+1));
    }

    for (int id = 1; id <= vertex_size; ++id) {
        for (int color = 1; color <= partition_size; ++color) {
            int index = getVertexIndex(id, color, partition_size);
            objfun[index]   = 0;
            ctype[index]    = CPX_BINARY;
            colnames[index] = new char[10];
            sprintf(colnames[index], "x_%d%d", id, color);
        }
    }

    int status = CPXnewcols(env, lp, n, objfun, NULL, NULL, ctype, colnames);

    if (status) {
        printf("Problem adding variables with CPXnewcols.\n");
        exit(1);
    }

    // free memory
    for (int i = 0; i < n; ++i) {
        delete[] colnames[i];
    }

    delete[] objfun;
    delete[] ctype;
    delete[] colnames;

    return 0;
}

int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, vector<edge>& edges, int edge_size, int partition_size) {

    // load first restriction
    int ccnt = 0;                             // new columns being added.
    int rcnt = edge_size * partition_size; // new rows being added.
    int nzcnt = rcnt*2;                       // nonzero constraint coefficients being added.

    double *rhs = new double[rcnt];          // independent term in restrictions.
    char *sense = new char[rcnt];            // sense of restriction inequality.

    int *matbeg = new int[rcnt];             // array position where each restriction starts in matind and matval.
```

```
    int *matind = new int[rcnt*2];          // index of variables != 0 in restriction (each var has an index defined above)
    double *matval  = new double[rcnt*2];   // value corresponding to index in restriction.
    char **rownames = new char*[rcnt];      // row labels.

    int i = 0;
    for (std::vector<edge>::iterator it = edges.begin(); it != edges.end(); ++it) {
        int from = it->from;
        int to   = it->to;
        for (int color = 1; color <= partition_size; ++color) {
            matbeg[i] = i*2;

            matind[i*2]   = getVertexIndex(from, color, partition_size);
            matind[i*2+1] = getVertexIndex(to  , color, partition_size);

            matval[i*2]   = 1;
            matval[i*2+1] = 1;

            rhs[i] = 1;
            sense[i] = 'L';
            rownames[i] = new char[40];
            sprintf(rownames[i], "%s", colors[color-1]);

            ++i;
        }
    }

    // debug flag
    // status = CPXsetintparam(env, CPX_PARAM_DATACHECK, CPX_ON);

    // add restriction
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind, matval, NULL, rownames);

    if (status) {
        printf("Problem adding restriction with CPXaddrows.\n");
        exit(1);
    }

    // free memory
    for (int i = 0; i < rcnt; ++i) {
        delete[] rownames[i];
    }

    delete[] rhs;
    delete[] sense;
    delete[] matbeg;
    delete[] matind;
    delete[] matval;
    delete[] rownames;

    return 0;
}


int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector<int> >& partitions, int partition_size) {

```

```
    // load second restriction
    int p = 1;
    for (std::vector<vector<int> >::iterator it = partitions.begin(); it != partitions.end(); ++it) {

        int size = it->size();                 // current partition size.
        if (size == 0) continue;               // skip empty partitions.

        int ccnt = 0;                          // new columns being added.
        int rcnt = 1;                          // new rows being added.
        int nzcnt = size*partition_size;       // nonzero constraint coefficients being added.

        double *rhs = new double[rcnt];        // independent term in restrictions.
        char *sense = new char[rcnt];          // sense of restriction inequality.

        int *matbeg = new int[rcnt];           // array position where each restriction starts in matind and matval.
        int *matind = new int[nzcnt];          // index of variables != 0 in restriction (each var has an index defined above)
        double *matval  = new double[nzcnt];   // value corresponding to index in restriction.
        char **rownames = new char*[rcnt];     // row labels.

        matbeg[0] = 0;
        sense[0]  = 'E';
        rhs[0]    = 1;
        rownames[0] = new char[40];
        sprintf(rownames[0], "partition_%d", p);

        int i = 0;
        for (std::vector<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
            for (int color = 1; color <= partition_size; ++color) {
                matind[i] = getVertexIndex(*it2, color, partition_size);
                matval[i] = 1;
                ++i;
            }
        }

        // add restriction
        int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind, matval, NULL, rownames);

        if (status) {
            printf("Problem adding restriction with CPXaddrows.\n");
            exit(1);
        }

        // free memory
        delete[] rownames[0];
        delete[] rhs;
        delete[] sense;
        delete[] matbeg;
        delete[] matind;
        delete[] matval;
        delete[] rownames;

        ++p;
    }
```

```
    return 0;
}

int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size) {

    // load third restriction
    int ccnt = 0;                               // new columns being added.
    int rcnt = vertex_size * partition_size;    // new rows being added.
    int nzcnt = rcnt*2;                         // nonzero constraint coefficients being added.

    double *rhs = new double[rcnt];             // independent term in restrictions.
    char *sense = new char[rcnt];               // sense of restriction inequality.

    int *matbeg = new int[rcnt];                // array position where each restriction starts in matind and matval.
    int *matind = new int[rcnt*2];              // index of variables != 0 in restriction (each var has an index defined above)
    double *matval  = new double[rcnt*2];       // value corresponding to index in restriction.
    char **rownames = new char*[rcnt];          // row labels.

    int i = 0;
    for (int v = 1; v <= vertex_size; ++v) {
        for (int color = 1; color <= partition_size; ++color) {
            matbeg[i] = i*2;

            matind[i*2]   = getVertexIndex(v, color, partition_size);
            matind[i*2+1] = color-1;

            matval[i*2]   = 1;
            matval[i*2+1] = -1;

            rhs[i] = 0;
            sense[i] = 'L';
            rownames[i] = new char[40];
            sprintf(rownames[i], "color_res");

            ++i;
        }
    }

    // add restriction
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind, matval, NULL, rownames);

    if (status) {
        printf("Problem adding restriction with CPXaddrows.\n");
        exit(1);
    }

    // free memory
    for (int i = 0; i < rcnt; ++i) {
        delete[] rownames[i];
    }

    delete[] rhs;
```

```
    delete[] sense;
    delete[] matbeg;
    delete[] matind;
    delete[] matval;
    delete[] rownames;

    return 0;
}

int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int partition_size) {

    int n = partition_size + (vertex_size*partition_size); // amount of total variables

    // calculate runtime
    double inittime, endtime;
    int status = CPXgettime(env, &inittime);

    // solve LP
    status = CPXmipopt(env, lp);

    status = CPXgettime(env, &endtime);

    if (status) {
        printf("Optimization failed.\n");
        exit(1);
    }

    // check solution state
    int solstat;
    char statstring[510];
    CPXCHARptr p;
    solstat = CPXgetstat(env, lp);
    p = CPXgetstatstring(env, solstat, statstring);
    string statstr(statstring);
    if (solstat != CPXMIP_OPTIMAL && solstat != CPXMIP_OPTIMAL_TOL &&
        solstat != CPXMIP_NODE_LIM_FEAS && solstat != CPXMIP_TIME_LIM_FEAS) {
        exit(1);
    }

    double objval;
    status = CPXgetobjval(env, lp, &objval);

    if (status) {
        printf("Problem obtaining optimal solution.\n");
        exit(1);
    }

    // get values of all solutions
    double *sol = new double[n];
    status = CPXgetx(env, lp, sol, 0, n - 1);

    if (status) {
        printf("Problem obtaining the solution of the LP.\n");
        exit(1);
    }

    // write solutions to current window
```

```
    cout << endl << "Optimization result: " << statstring << endl;
    cout << "Runtime: " << (endtime - inittime) << endl;
    printf("Graph: vertex_size: %d, edge_size: %d, partition_size: %d\n", vertex_size, edge_size, partition_size);
    cout << "Colors used: " << objval << endl;
    for (int color = 1; color <= partition_size; ++color) {
        if (sol[color-1] == 1) {
            cout << "w_" << color << " = " << sol[color-1] << " (" << colors[color-1] << ")" << endl;
        }
    }

    for (int id = 1; id <= vertex_size; ++id) {
        for (int color = 1; color <= partition_size; ++color) {
            int index = getVertexIndex(id, color, partition_size);
            if (sol[index] == 1) {
                cout << "x_" << id << " = " << colors[color-1] << endl;
            }
        }
    }

    delete[] sol;

    return 0;
}

int setBranchAndBoundConfig(CPXENVptr& env) {

    // CPLEX config
    // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.2.0/ilog.odms.cplex.help/Content/Optimization/Documentation/CPLEX/_pubskel/CPLEX916.html

    // maximize objective function
    // CPXchgobjsen(env, lp, CPX_MAX);

    // enable/disable screen output
    CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);

    // set excecution limit
    CPXsetdblparam(env, CPX_PARAM_TILIM, 3600);

    // disable presolve
    // CPXsetintparam(env, CPX_PARAM_PREIND, CPX_OFF);

    // enable traditional branch and bound
    CPXsetintparam(env, CPX_PARAM_MIPSEARCH, CPX_MIPSEARCH_TRADITIONAL);

    // use only one thread for experimentation
    CPXsetintparam(env, CPX_PARAM_THREADS, 1);

    // do not add cutting planes
    CPXsetintparam(env, CPX_PARAM_EACHCUTLIM, CPX_OFF);

    // disable gomory fractional cuts
    CPXsetintparam(env, CPX_PARAM_FRACCUTS, -1);

    // measure time in CPU time
    CPXsetintparam(env, CPX_PARAM_CLOCKTYPE, CPX_ON);

```

```
    return 0;
}
```