# *Tesis de Licenciatura*

# *Software de Multiprocesamiento Paralelo para implementación de "Branch And Cut"*

***Luis Rubén Diez***

*Ciencias de la Computación*
*Departamento de Computación*
*Facultad de Ciencias Exactas y Naturales*
*Universidad de Buenos Aires*

*Director: Dr. Min Chih Lin*
*Codirectora: Lic. Paula Lorena Zabala*

*Diciembre 2004*

# RESUMEN

Esta tesis desarrolla un "framework" para implementar problemas en los cuales el tiempo de procesamiento es muy importante. Se aplican los conceptos de procesamiento paralelo y otras características que favorecen a las resoluciones de estos tipos de problemas.

Existen problemas, como los problemas de optimización, en los cuales el tiempo para obtener una solución deseable puede llegar a ser considerable, para mejorar estos tiempos y el manejo de estas resoluciones, el usuario puede implementar su problema utilizando la librería provista por esta tesis.
El usuario desarrolla su propio programa utilizando el "framework", implementando así un procesamiento paralelo para la resolución de su problema, y además le provee de otras características, como pueden ser: detener el procesamiento, manteniendo la solución parcial hasta el momento, y poder continuar o no cuando lo desee; también puede configurar el grado de paralelismo.

Para la implementación de la librería se realiza un desarrollo para resolución de optimización de problemas lineales enteros mixtos, utilizando algoritmos tipo "Branch and Cut", y para las relajaciones lineales se utilizará la librería CPLEX.

El resultado final de la tesis es una librería para implementación de algoritmos tipo "Branch and Cut" con procesamiento paralelo en las resoluciones de los subproblemas.

# ABSTRACT

This thesis develops a framework to implement problems in which the time of processing is very important. The concepts of the parallel processing and other characteristics that help the resolutions of this kind of problems are applied.

There are problems like the optimization ones, in which the time to get a desired solution can become substantial, to improve these times and the handling of these resolutions, the user can implement his problem using the library supplied by this thesis.

The user develops his own program using the framework implementing a parallel processing for the resolution of his problem, and besides it gives other characteristics such as stopping the processing and being able to continue or not when he desires; he can also configure the grade of parallelism.

To implement the library, a development for the resolution of optimization of mixed integers problems is made using type algorithms "Branch and Cut", and for the linear relaxation will be used the library CXPLEX.

The final result of the thesis is a library to implement the type algorithms "Branch and Cut" with a parallel processing in the resolutions of the subproblems.

# *ÍNDICE GENERAL*


## OBJETIVO ..... 4

## PROCESAMIENTO PARALELO ..... 5

**Introducción ..... 5**
Multi-Procesos ..... 5
Multi-Hilos ..... 5

**Diseño ..... 6**

**Arquitectura ..... 6**
Maestro ..... 6
SubProcesoLp ..... 6
AdmPool ..... 7
AdmSubProcesoLp ..... 7
Interfaz de Usuario ..... 7
Comunicación entre Procesos ..... 7
Memoria Compartida ..... 7
Mensajería ..... 7

**Diseño de Arquitectura ..... 8**

**Persistencia de Datos ..... 9**

## BRANCH AND CUT ..... 10

**Introducción ..... 10**
Optimización Combinatoria ..... 10
Programación Lineal (PL) ..... 10
Programación Entera Mixta ..... 10

**Branch and Bound ..... 11**
Regla de Bifurcación ..... 11
Diagrama de Flujo del Algoritmo *Branch and Bound* ..... 12
La geometría de Bifurcación ..... 13

**Branch And Cut ..... 13**
Heurísticas en *Branch and Cut* ..... 14
Diagrama De Flujo del Algoritmo Branch and Cut ..... 15

## POR QUÉ BRANCH AND CUT ..... 16

**CPLEX ..... 16**

## APLICACIÓN PRÁCTICA ..... 17

# EJEMPLO ..... 18

**Definición ..... 18**

**Formulación Matemática ..... 18**
Variables ..... 19
Formulación ..... 19
Restricciones ..... 19

**Los Datos ..... 20**
Función de Costo ..... 20

**Programa Principal ..... 21**
Entrada ..... 21
Implementación ..... 21
Código Fuente ..... 24

**Ejecución del Ejemplo ..... 25**
Árbol Branch and Cut Generado ..... 26
Árbol *Branch and Cut* - Parcial ..... 26
Árbol Branch and Cut - Final ..... 27
Distribución de Procesamiento ..... 28
Grado de Paralelismo ..... 29
Tiempos de Procesamiento ..... 29
Hardware ..... 30
Configuración ..... 30

**Ejemplo con Cortes ..... 31**
Árbol *Branch and Cut* - Final ..... 31

**Representación de la Solución ..... 32**

# CONCLUSIONES ..... 33

# TRABAJOS FUTUROS ..... 34

# BIBLIOGRAFIA ..... 35

# APÉNDICE ..... 37

# OBJETIVO

El objetivo de esta tesis es construir una librería aplicando el concepto procesamiento paralelo, la misma podrá ser utilizada para desarrollos personalizados, de esta manera se trata de aprovechar los actuales recursos de hardware, de importante crecimiento en las ultimas décadas, para implementar problemas en los cuales el tiempo de procesamiento es muy importante para obtener una solución deseable.

La idea es construir un "framework" que provea al usuario las herramientas necesarias para desarrollar su propio programa, y así poder obtener los resultados adecuados.
La librería contendrá un proceso principal, el cual tendrá el conocimiento general del problema a resolver, procesos de control, y procesos secundarios (*subprocesos*). De estos últimos se ejecutarán en forma paralela la cantidad de instancias que el usuario crea conveniente, en estos estará realmente el poder del procesamiento paralelo.
El proceso principal administra los datos y toda la información de control, levantando diferentes instancias de los *subprocesos*, y definiendo espacios de memoria compartida, la cual se utilizará para el traspaso de información.

Se trata de abstraer al usuario de las dificultades e inconvenientes que se pueden plantear en la programación paralela, ya que todo este código está implementado en la librería. Y deja que el código del usuario sea dedicado sólo al tratamiento del problema a resolver, brindándole para ello el poder del procesamiento paralelo.

Para llevar a cabo la implementación, se eligió por sus características, un desarrollo para resolución de optimización de problemas lineales enteros mixtos, utilizando algoritmos tipo "Branch and Cut", y para las relajaciones lineales se utilizará la librería CPLEX.
Luego el procesamiento paralelo existe en cada nodo del árbol "Branch and Cut", es decir los *subprocesos* ya citados, serán los encargados de la resolución de los subproblemas.

Así con este objetivo planteado, el resultado final será una librería para implementación de algoritmos tipo "Branch and Cut" con procesamiento paralelo en las resoluciones de los subproblemas.

# PROCESAMIENTO PARALELO

## INTRODUCCIÓN

De la existencia de problemas que demandan un tiempo de procesamiento muy alto, y del crecimiento considerable del hardware en las últimas décadas, surge la motivación de una implementación de procesamiento paralelo para la resolución de estos problemas.

En un ambiente de procesamiento paralelo, se optimizan los tiempos de procesamiento compartiendo simultáneamente procesador(es), memoria, y otros recursos del sistema operativo.
La implementación de un sistema paralelo se puede hacer en software, hardware o una combinación de ambos. Se pueden utilizar unidades de procesamiento simétricas, es decir, todas capaces de ejecutar la misma tarea con el mismo nivel de eficiencia; o utilizar unidades especializadas en diferentes tareas.

Existen dos maneras de implementar un paralelismo por software:

### MULTI-PROCESOS

El concepto de *proceso* referencia a un proceso del sistema operativo. Todos los sistemas operativos Unix, así como muchos otros sistemas operativos, soportan multi-procesos.
En este esquema cada proceso tiene un espacio de direccionamiento propio y todos comparten el(los) procesador(es). La comunicación y la sincronización entre los procesos pueden ser por memoria compartida o por pasajes de mensajes.

### MULTI-HILOS

Es similar al multiprocesamiento, excepto que un proceso puede estar dividido en varios hilos de ejecución. Todos los hilos comparten el mismo espacio de direccionamiento, esto los diferencia notablemente del esquema multi-procesos.
El pasaje de datos entre los hilos es mucho más eficiente que entre procesos, sin embargo la protección del área de datos es más débil que las de procesos, ya que si un hilo corrompe el área de datos, todos los demás hilos serán afectados.

## DISEÑO

En la tesis se desarrolla un sistema de procesamiento paralelo implementado con multi-procesos. La principal razón por la que se eligió este esquema y no multi-hilos, es la criticidad del área de datos, estableciendo en todo el tiempo de ejecución una máxima protección de los datos procesados. Existen otras razones como la complejidad del código de los procesos, y la escalabilidad de los mismos.
La comunicación y sincronización entre los procesos se implementa con una combinación de memoria compartida y memoria distribuida con mensajería de control.

La librería contiene un proceso principal (***Maestro***), que posee el conocimiento general del problema a resolver, procesos de control (***AdmPool***, ***AdmSubProcesoLp***), y procesos secundarios (***SubProcesoLp***).
De estos últimos subprocesos existen diversas instancias que se ejecutan en forma paralela, implementando así el multiprocesamiento.

## ARQUITECTURA

Se detallan los principales módulos de la arquitectura paralela de la librería. Aquellos que representan un proceso, se ejecutan en forma paralela.
En APÉNDICE, se pueden ver más detalles de arquitectura y procesos.

### MAESTRO

Es el componente principal, contiene el conocimiento general del problema global. Tiene una importante relación con el proceso del usuario, en realidad, en términos de programación este componente es un objeto del programa principal del usuario.
Además de las funcionalidades propias para las resoluciones de los problemas, contiene funcionalidades de control entre otros módulos, mantiene los resultados parciales y final de las soluciones del problema global, y es el módulo responsable de la protección de la información procesada.

### SUBPROCESOLP

Es un proceso que realiza una tarea en particular, su funcionalidad es específica para resolver un subproblema del problema global.
Existen varias instancias de este proceso. La cantidad de instancias se configura por parámetros estáticos, y determina el grado de paralelismo del procesamiento.
Para detalles ver APÉNDICE.

### ADMPOOL

Es un proceso encargado de administrar los datos del problema. En nuestro caso administra las Variables y Restricciones del problema a resolver.
Su mayor interactividad es con los procesos SubProcesoLP que le demandan las Variables y Restricciones del subproblema que deben resolver. Esta comunicación se realiza a través de mensajería. Es un proceso hijo del proceso Maestro.

### ADMSUBPROCESOLP

Es un proceso administrador de los procesos SubProcesoLp. Estos últimos son procesos hijos, y la funcionalidad de este administrador es puramente de control sobre los mismos. Este proceso administrador es hijo del proceso Maestro.

### INTERFAZ DE USUARIO

Es un proceso cuya funcionalidad es la de brindar una interfaz al usuario para facilitar la aplicación de las funcionalidades de la librería.

### COMUNICACIÓN ENTRE PROCESOS

Como ya se adelanto, la comunicación y sincronización entre los módulos es una combinación de memoria compartida y pasaje de mensajes.

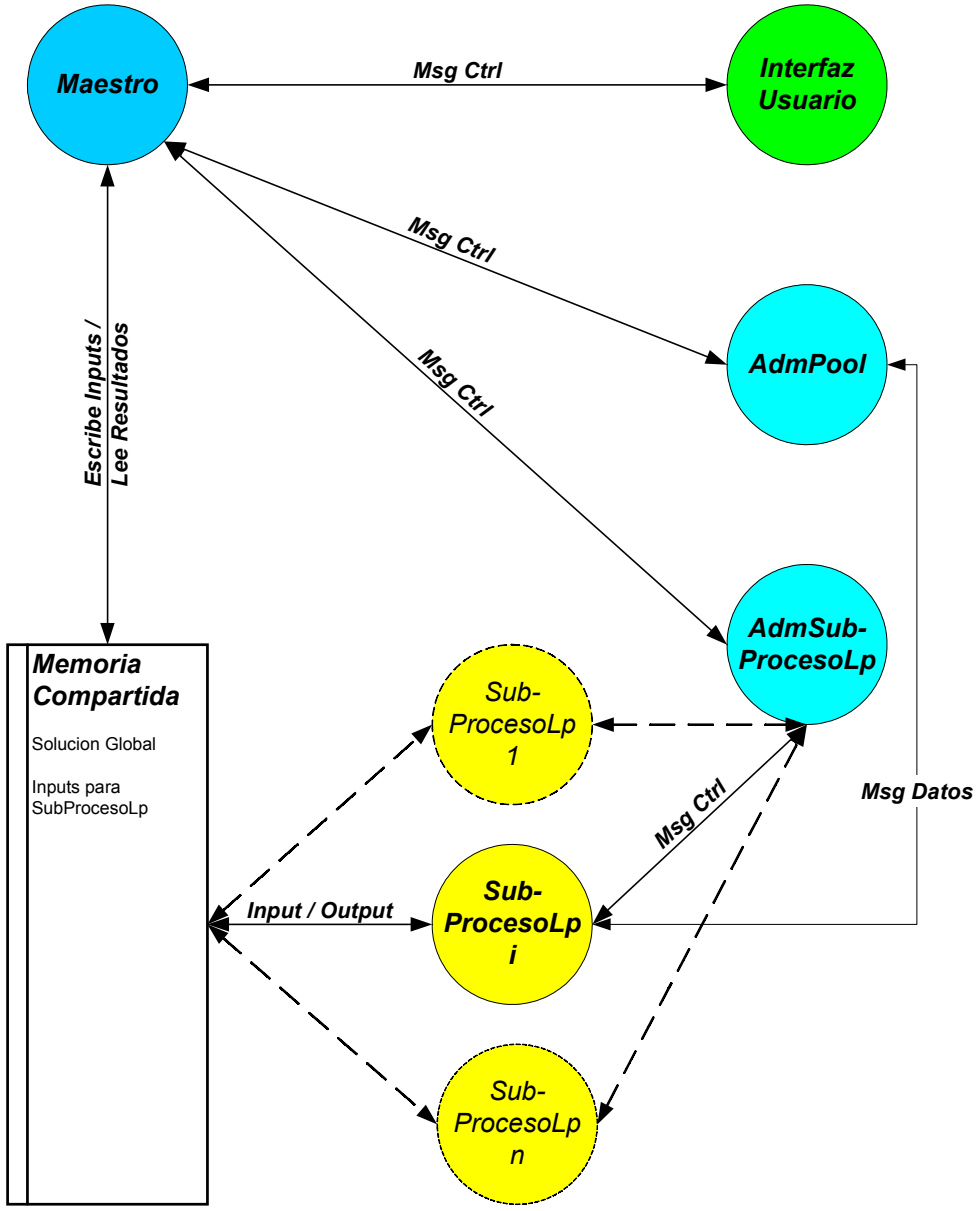
#### Memoria Compartida

Para una comunicación más eficiente entre el proceso Maestro y las distintas instancias de los SubProcesoLP se establece un área de memoria compartida entre ellos.
Una parte de la memoria compartida se utiliza para la información de entrada de los subproblemas a resolver, y está asignada por el proceso Maestro, de manera tal que se distribuya entre todos los SubProcesoLp.
Y otra parte es compartida por el módulo Maestro y todos los SubProcesoLp, en la cual existe la información de la solución global del problema.
Para el acceso a la memoria compartida cada proceso utiliza semáforos (implementado con IPC), solucionando de esta manera el problema de la concurrencia.

#### Mensajería

Entre todos los procesos de la librería existe una comunicación de mensajes, implementada con IPC, esta comunicación es tanto de control como de datos, dependiendo entre que módulos se establezca.
Para más detalles del pasaje de mensajes entre los procesos, ver APÉNDICE.

## DISEÑO DE ARQUITECTURA



## PERSISTENCIA DE DATOS

Una característica muy importante de la librería es su administración de los datos procesados, y control de ejecución.

El módulo Maestro realiza en forma automática ***puntos de control*** guardando en forma persistente imágenes del procesamiento actual, es decir toda la información procesada, pendiente de procesar, y datos de control.
De esta manera, ante una caída inesperada del procesamiento por algún evento externo, se puede retomar la ejecución desde el último punto de control realizado.
Esto es muy importante cuando se quiere procesar grandes instancias de problemas que pueden demorar un tiempo considerable.

Mediante el módulo Interfaz de Usuario se pueden enviar órdenes de control al módulo Maestro, de forma tal que se puede detener (y continuar) el procesamiento cuando el usuario lo desea, pudiendo acceder a la información procesada hasta dicho momento.
Es posible mantener información de varias instancias procesadas de forma parcial (interrumpidas), y el usuario puede retomar o no la ejecución de cualquiera de ellas.
También con la Interfaz es posible realizar y/o recuperar backups de diferentes instancias.
Para mayores detalles de las órdenes de control del módulo Interfaz de Usuario, ver APÉNDICE.

Cabe mencionar en este capítulo el acceso a los datos de los resultados. El **árbol *branch and cut*** generado por el algoritmo, se encuentra en un archivo en formato XML, pudiendo el usuario acceder en cualquier instante al mismo. Este archivo se va generando de manera parcial a medida que se obtiene los resultados.

# BRANCH AND CUT

## INTRODUCCIÓN

La programación matemática, originada como una disciplina científica, se concierne en la búsqueda de decisiones óptimas bajo restricciones.
Desde su invención en 1958, hasta hoy, los algoritmos *branch-and-cut-and-price* pertenecen a las técnicas más exitosas para la resolución de problemas de programación lineal entera mixta y de optimización combinatoria.

## OPTIMIZACIÓN COMBINATORIA

Con una función lineal objetivo, la tarea consiste en seleccionar de una familia de subconjuntos $\mathcal{F} \subseteq 2^E$ de un conjunto finito E en $F \in \mathcal{F}$, que maximiza (minimiza) una función lineal objetivo: $\sum e \in F$, ce para coeficientes $c \in R$, $e \in E$.
Matemáticamente esto es trivial, porque una optimización del conjunto F puede obtenerse con una enumeración finita, sin embargo tal estrategia es claramente insatisfecha por un computo práctico.

## PROGRAMACIÓN LINEAL (PL)

El padre de la programación lineal es George B. Dantzig, que propuso el siguiente modelo, y construyó el famoso algoritmo *simplex* para la resolución.

*maximizar* $c^T x$
*sujeto a:* $Ax \leq b, x \geq 0$ *donde:* $c \in R^n, A \in R^{mxn}, b \in R^m$

## PROGRAMACIÓN ENTERA MIXTA

Para la integración de algunos problemas reales se requiere de variables de decisión o variables que representan cantidades que sólo pueden tomar valores enteros. Así se plantea un modelo de programación lineal entera, que es un modelo en el cual algunas variables requieren valores enteros.
Los algoritmos originales (Ralph Gomory, 1958) introducen el concepto de ***planos de cortes***. Este método es teóricamente eficiente, arribando a soluciones óptimas; pero para resolver los problemas reales se volvía impracticable por los tiempos computacionales.
En 1960, A.H.Land y A.G.Doig propusieron el método ***Branch and Cut***, y comenzó a ser utilizado en los códigos computacionales comerciales provenientes de las mayorías de las compañías de computación.
El método inicialmente se uso para la resolución de problemas de programación lineal, y en los 70 para los de programación entera mixta.

## BRANCH AND BOUND

*Branch and bound* es el método más usado para resolver problemas de programación lineal entera mixta. Es la clase general de la cual descienden los algoritmos *branch-and-cut-and-price*.
El algoritmo usa la estrategia de dividir y conquistar para particionar el espacio de soluciones en subproblemas y luego optimizar individualmente cada uno.

Consideramos la instancia:
$S$ es el conjunto de soluciones de un determinado problema, sea $c \in R^S$ , el vector de costos asociados con los miembros de $S$. Queremos determinar el elemento de $S$ con menor costo, y obtenemos un $\hat{s} \in S$, como una "buena" solución.

Usando *branch and bound*, en la fase de procesamiento se relaja la integrabilidad de las variables del problema produciendo un *límite inferior* en el valor de la solución óptima:
- Si la solución de la relajación es un elemento de $S$ o tiene costo igual a $\hat{s}$, entonces decimos que la nueva solución es **óptima**, y finaliza el algoritmo.
- En otro caso, identificamos n subconjuntos de $S$: $S_1, \ldots, S_n$, *tales que* $\cup S^n_{i=1} = S$, cada uno de estos subconjuntos son llamados *subproblemas*, y se agregan a la lista de candidatos a subproblemas (que necesitan ser procesados).
  A este paso se lo denomina **bifurcación** (branching).

Para continuar con el algoritmo, se selecciona uno de los subproblemas candidatos y se procesa. Existen cuatro posibles resultados:
- Si encontramos una solución factible **mejor que $\hat{s}$**, reemplazamos $\hat{s}$ por la nueva y continuamos.
- Puede ocurrir que el subproblema no tenga solución, en este caso se desecha el subproblema (nodo), haciendo **poda**.
- Otro caso se compara el límite inferior con el límite superior global, si este es mayor o igual, entonces también se hace **poda** por límite.
- Y finalmente si no se puede podar, se hace una nueva **bifurcación**, generando nuevos hijos de este subproblemas y agregándolos a la lista de subproblemas candidatos activos (o sea, esperando a ser procesados).

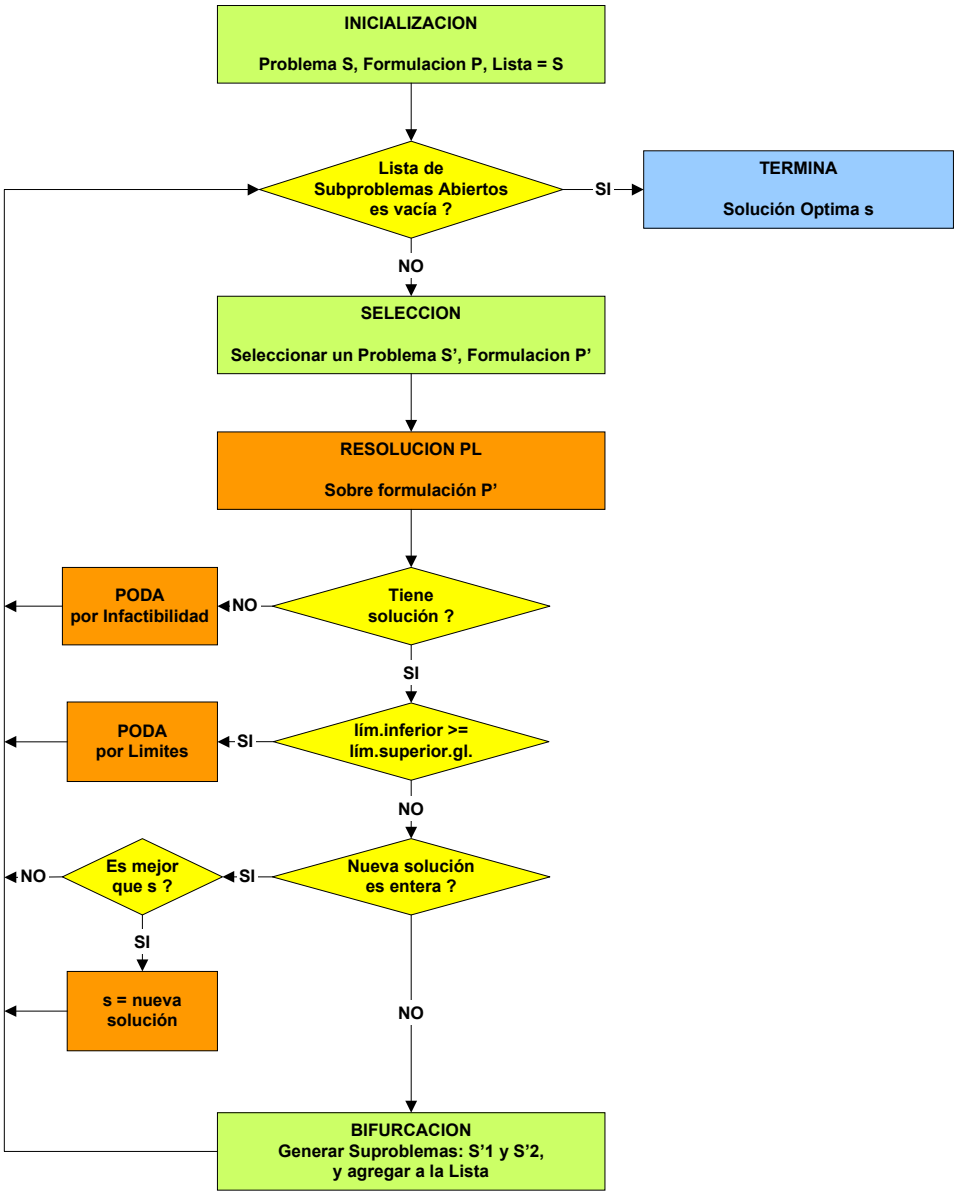
Se continua de esta manera hasta que la lista de subproblemas candidatos activos este vacía, y en este punto la solución actual será la ***solución óptima***.

## REGLA DE BIFURCACIÓN

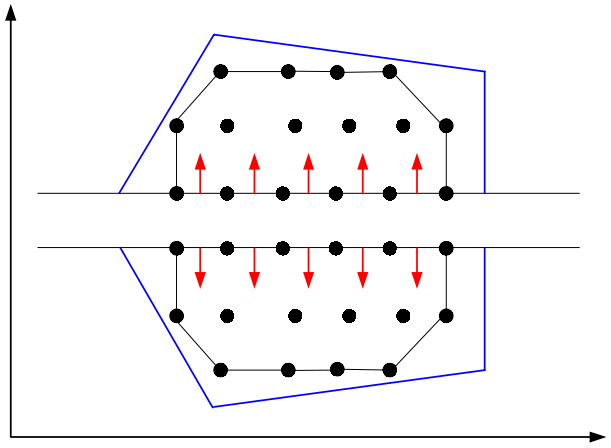
La manera más común de bifurcación es la siguiente:
- Seleccionar la variable entera i, tal que el valor $x_i$ es fraccional en la solución de la relajación lineal.
- Crear dos subproblemas con los subconjuntos de $S$: $S_1$ y $S_2$ tales que:
  $S_1 = S \cap \{ x: x_i <= \lfloor x_i \rfloor \}$
  $S_2 = S \cap \{ x: x_i >= \lceil x_i \rceil \}$

Esta es sólo una manera de hacer la bifurcación, libremente se puede bifurcar de la manera más conveniente para la resolución del problema.

## DIAGRAMA DE FLUJO DEL ALGORITMO *BRANCH AND BOUND*



## LA GEOMETRÍA DE BIFURCACIÓN



## BRANCH AND CUT

La mayoría de los ejemplos de aplicaciones prácticas se tornan NP-Hard. No obstante una combinación de programación entera, lleva a una metodología algoritmica que ha resuelto optimizaciones de casos del mundo real.
Cualquier $\mathcal{F} \subseteq E$ esta representado por el vector $X^F \in \{0,1\}^E$ tal que $X^F_e = 1 \Leftrightarrow e \in F$.
En general no es fácil definir el problema real como un problema de programación lineal entera. Teóricamente y en términos del esfuerzo de implementaron, hacer un desarrollo de las técnicas de programación lineal entera utilizable en una computación eficaz, implica normalmente una larga y compleja tarea.

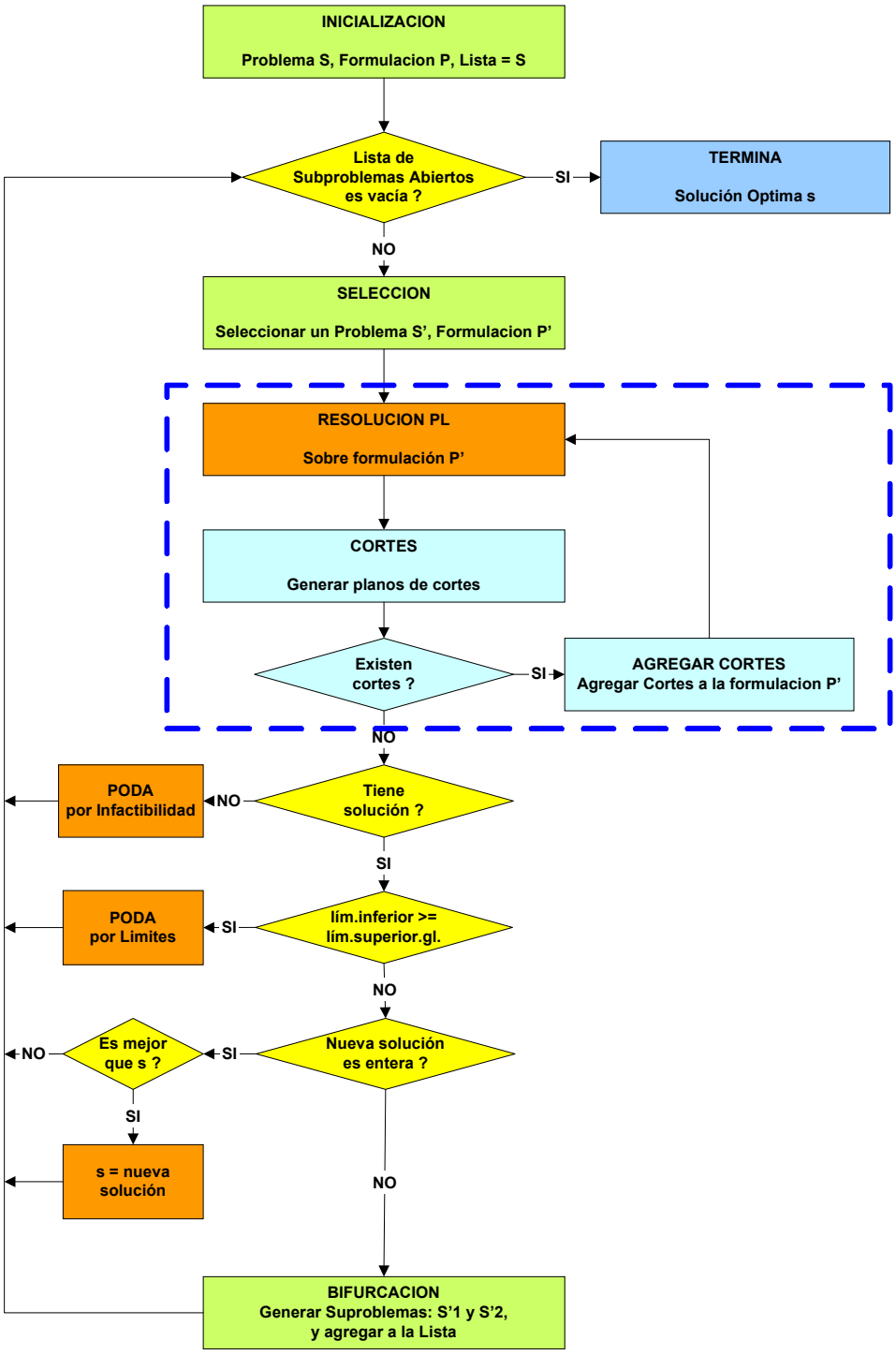
*Branch and cut* es una variante de *branch and bound*, donde las relajaciones de PL son magnificadas por desigualdades válidas globales. Las desigualdades válidas son generadas dinámicamente utilizando métodos de separación (planos de cortes). Se trata de mejorar los límites incorporando desigualdades válidas.
Las estructuras de cortes, generadas por un algoritmo de separación, son válidas por definición.
Las ventajas de las desigualdades válidas globales son que pueden ser utilizadas más tarde en otro nodo del árbol de búsqueda. Para compartir esta información, se utilizan *pools de cortes* para almacenar cortes y proveerlos eficientemente para ser utilizados más tarde.

En resumen, un algoritmo básico de *branch and cut*, es un algoritmo del *branch and bound*, en el cual los límites son soluciones de las PL-relajaciones que son iterativamente reforzadas por planos de cortes, en cada nodo del árbol. Se utilizan las herramientas computacionales de ***bifurcar***, y algoritmos de ***límites*** junto con las herramientas teóricas de ***poliedros***.

## HEURÍSTICAS EN *BRANCH AND CUT*

Es un método utilizado para obtener "buenas" soluciones factibles y de esta manera obtener buenas cotas. El papel de las heurísticas es acelerar el método de búsqueda de soluciones factibles.
Existen diferentes tipos, las ***heurísticas iniciales***, usadas para obtener la solución inicial y límites para el comienzo del *branch and cut*; y las ***heurísticas primales*** para obtener límites en los nodos del árbol de búsqueda. En general, estas heurísticas utilizan la información dada por la solución de la relajación lineal.
Cabe aclarar que las heurísticas no resuelven el problema, ayudan a encontrar "buenas" soluciones.

## DIAGRAMA DE FLUJO DEL ALGORITMO BRANCH AND CUT




El recuadro remarcado en el diagrama indica los nuevos componentes agregados al algoritmo básico de *branch and bound.*

# POR QUÉ BRANCH AND CUT

Los problemas combinatorios son conceptualmente fáciles de modelar como problemas enteros, pero su resolución en la practica implica mucho tiempo de procesamiento.
Desde los principios de la optimización como reconocimiento en el campo matemático, los investigadores han sido intrigados y obstaculizados por la dificultad para resolver la mayoría de las más interesantes clases de problemas discretos de optimización. Aún cuando los problemas de optimización sean conceptualmente fáciles de modelar como programas enteros, el desafío de resolverlos en la práctica lleva mucho tiempo.
Existen sistemas que son fuertes para una solución de problemas generales de optimización enteros, pero no son convenientes para una solución de problemas de optimización de gran escala.

En las últimas décadas hubo grandes progresos en cuanto a la habilidad para resolver problemas de optimización discreta de gran escala.
Una de las principales razones del progreso, es el importante crecimiento del hardware, tanto en velocidad de procesamiento como en memorias. Favorecido por esto último, el uso del procesamiento paralelo se impone para el desarrollo de los problemas de optimización.

Por éstas razones descriptas, se eligió el método de *branch and cut* para ser implementado con la librería PBC, y así poder resolver de una manera más eficiente los problemas "difíciles", empleando los recursos del procesamiento paralelo utilizados para la implementación. Además, y quizás la razón más importante es la técnica de dividir y conquistar usada por el algoritmo, que es muy apropiada para una implementación de procesamiento paralelo.

Otro importante motivo es el tiempo de proceso para la resolución de las relajaciones lineales, que es uno de los cuellos de botella del algoritmo *branch and cut*. Con esta implementación se puede paralelizar en cada nodo el procesamiento de estas relajaciones, obteniendo resultados muy satisfactorios.

## CPLEX

CPLEX es una herramienta para resolver los problemas de PL. Ofrece librerías para resolver, modificar e interpretar resultados de PL.
Para resolver un problema de PL utiliza el método ***simplex*** que satisface todos los requerimientos del algoritmo *branch and cut*. Si bien el algoritmo *simplex* no es de orden polinomial, en la práctica se obtienen resultados muy satisfactorios, aún mejores que otros algoritmos polinomiales existentes.

# APLICACIÓN PRÁCTICA

El usuario desarrollara su propio programa utilizando esta librería para la optimización de su problema específico.
Inicialmente se debe formular el problema lineal entero y construir el "mapping" a las variables, y luego utilizando las rutinas adecuadas, se introduce un conjunto de restricciones que pueden ser agregadas por generación de cortes.

La tarea principal del usuario, es representar el problema original del mundo real como un problema de programación lineal entera, y del mapeo las variables y restricciones. No se debe ocupar de la implementación de la resolución de las relajaciones lineales y de la incorporación de los cortes en cada nodo, para ello sólo debe utilizar las rutinas provistas. Tampoco se debe ocupar del mantenimiento del árbol de búsqueda y el procesamiento en paralelo de estos subproblemas.

El objetivo final del programa del usuario es obtener la optimización de su problema original.

# EJEMPLO

Se crea un ejemplo de uso de la librería, para ello se eligió resolver una instancia del conocido problema del Viajante de Comercio. Se desarrolla el código fuente del usuario, y se vincula con la librería. Luego se realiza una ejecución con un juego de datos reales, y se hace el seguimiento de la misma.

El ***Problema del Viajante de Comercio*** (***TSP****: Traveling Salesman Problem*) es uno de los problemas que centra mucha atención de matemáticos y computadores científicos, especialmente por su fácil descripción y su dificultad para resolverlo. Pertenece a la clase de problemas de optimización computacional para la versión de problemas de decisión, conocidos como "NP-complete". Hasta hoy nadie ha encontrado un algoritmo polinomial para resolverlo, sin embargo, debido a su gran cantidad de aplicaciones, se desarrollaron numerosos algoritmos heurísticos y/o diferentes técnicas de optimización para resolver instancias de gran escala.

## DEFINICIÓN

El problema se puede definir simplemente como: "Un viajante de comercio desea visitar exactamente una vez cada una de *n* ciudades, partiendo de una de ellas, y retornando a la misma. Para viajar de una ciudad a otra existe un costo, luego se desea saber cual es el circuito menos costoso que el viajante puede tomar." **[16]**

## FORMULACIÓN MATEMÁTICA

Como primer paso para poder resolver el problema, se debe definir una "buena" formulación matemática del mismo. En este caso la estructura matemática es una grafo completo, donde cada nodo, representa una ciudad, y las aristas representan la conectividad entre ciudades. Cada eje tiene asociada una distancia (o costo).
La longitud de un viaje (incluyendo varias ciudades), es la suma de las distancias asociadas a las aristas que forman el camino. Dependiendo si las aristas son bidireccionales, se puede distinguir entre un TSP simétrico o asimétrico. Una formulación de programación lineal entera del problema para un TSP asimétrico es:

## VARIABLES

Para cada arista del grafo (i,j) se definen las variables enteras $x_{ij}$, que toman los valores:

$x_{ij}$ = ***1**, si la arista (i,j) pertenece al circuito solución.*
= ***0**, cc.*

El valor del coeficientes $c_{ij}$ de estas variables en la función objetivo es el costo de la arista que representa dicha variable.
También para cada arista se define la variable entera $y_{ij}$ que representan el orden de las aristas en el circuito elegido. Estas variables no forman parte de la función objetivo, sólo interviene en las restricciones del problema.

$y_{ij}$ = ***0**, si la arista (i,j) no pertenece al circuito solución.*
= ***k**, orden de la arista (i,j) en el circuito,* $1 \leq k \leq n$

## FORMULACIÓN

$$\textbf{minimizar } \sum_i^n \sum_j^n c_{ij}\, x_{ij}$$

t.q.:

$$\sum_i^n x_{ij} = 1 \quad \forall i \qquad \text{(R1)}$$
$$\sum_j^n x_{ij} = 1 \quad \forall i \qquad \text{(R2)}$$

$$\sum_j^n y_{pj} = 1 \quad \forall j \text{, p: nodo inicial.} \qquad \text{(R3)}$$
$$\sum_i^n y_{ip} = n \quad \forall i \text{, p: nodo inicial.} \qquad \text{(R4)}$$

$$\sum_j^n y_{kj} - \sum_i^n y_{ik} = 1 \quad \forall k: 2 \leq k \leq n \qquad \text{(R5)}$$

$$y_{ij} \leq r_{ij}\, x_{ij} \quad \forall i,j \qquad \text{(R6)}$$
$$\text{t.q.: } r_{ij} = 1 \text{ si } i=1$$
$$= n \text{ si } j=1$$
$$= n-1 \text{ cc}$$

## RESTRICCIONES

Existen diversas maneras de definir las restricciones del problema, esto impacta en la cantidad de restricciones, o en el crecimiento de las mismas durante la resolución del problema. Se ha implementado una manera en la cual el número de restricciones es polinomial con relación a la cantidad de variables.

- Las restricciones (R1) y (R2), garantizan que a cada nodo, llega y sale sólo una arista.

- Para las restricciones (R3) y (R4), debemos elegir un nodo inicial ***p*** que será el origen del camino, luego la suma de las variable ***y*** es: ***1*** para las que representan las aristas iniciales (que parten de ***p***), y ***n*** para las variables y que representan las aristas finales (que llegan al ***p***).
- La restricción (R5) garantiza la secuencialidad de los valores de las variables ***y***.
- Y por último todas las restricciones (R6) relacionan los valores de ***y*** a las variables ***x***.

La cantidad de restricciones para cada problema dependerá polinomialmente de la cantidad de variables, o cantidad de ciudades($n$) del problema real TSP, de la siguiente manera:

**#variables** $= n^2 * 2$
**#restricciones** $= n + n + 1 + 1 + (n - 1) + ( n^2 / 2 )$

## LOS DATOS

Los datos fueron recuperados de datos reales del *National Imagery and Mapping Agency* **[19]** que ha creado una base de datos con la locación geográfica de Argentina. Como primer ejemplo tomamos una sub-muestra de la muestra real para tener un mejor seguimiento del procesamiento.
Cada ciudad es representada por su latitud y longitud, el costo de viaje entre dos ciudades es especificada por una aproximación de la mayor distancia circular en la Tierra (tomando la Tierra como una esfera).

## FUNCIÓN DE COSTO

Se define una función de costo que calcula la distancia entre dos pares de coordenadas geográficas. Los valores de latitud y longitud son valores expresados en radianes.

```
calculoDistancia( lat_i, long_i, lat_j, long_j,)
{
   if( lat_i = lat_j  and  long_i = long_j )
         distancia = 0;

   else if( (sin(lat_i) * sin(lat_j) + cos(lat_i) * cos(lat_j) * cos(long_i - long_j) )  >  1 )
         distancia = 3963.1 * acos(1);

   else
         distancia = 3963.1 * acos( sin(lat_i) * sin(lat_j) + cos(lat_i) * cos(lat_j) * cos(long_i - long_j));

   return distancia;
}
```

## PROGRAMA PRINCIPAL

Se desarrolló un programa para la resolución del problema TSP, este programa no es parte de la librería PBC, sino es un ejemplo del uso de misma para la resolución de un problema.

### ENTRADA

Los datos de entrada del programa son los datos de las ciudades que se desean recorrer en el problema TSP, y se encuentran en un archivo plano con registros de cantidad de campos fijos, y campos variables con delimitador, con el siguiente formato:

| Nombre | Descripción | Tipo |
|---|---|---|
| Identificador | Identificador único de la ciudad. | Entero |
| Nombre | Nombre de la ciudad | String (sin blancos) |
| Latitud | Latitud de la coordenada geográfica | Real (99.9999999) |
| Longitud | Longitud de la coordenada geográfica | Real (99.9999999) |

El nombre de los campos es puramente descriptivo, no debe aparecer en el archivo, los campos son de tamaño variable, delimitados por espacio, y el delimitador de registros es el fin de línea, es decir cada línea del archivo corresponde a un registro.
Como ejemplo se muestran algunos registros:

*1 Bariloche -41.15 -71.3*
*2 Buenos_Aires -34.6 -58.45*
*3 El_Calafate -50.3333333 -72.3*
*...*

### IMPLEMENTACIÓN

La implementación del programa ejemplo, básicamente contiene dos etapas. La primera es la transformación del problema del mundo real al formato e interpretación requerido por la librería; y luego en una segunda etapa se realiza la optimización del problema.
Para implementar la optimización, se desarrolla una clase ***MaestroUsr***, descendiente de la clase *Maestro*, y también se desarrolla la clase ***SubproblemaUsr***, heredada de la clase *Subproblema*. Para ambas clases, se redefinen algunos métodos heredados, como son:

***MaestroUsr::outputParcial()***
Simplemente escribe en el log del proceso maestro los resultados de cada subproblema, en el mismo tiempo que se van generando.

***MaestroUsr::output()***
Realiza la escritura en el log del proceso maestro de la solución global encontrada.

***SubproblemaUsr::generarReglasBranch()***
Implementa una bifurcación por la primer variable no entera. Es decir del resultado de la relajación lineal, toma la primer variable ***x*** no entera, y genera dos nuevas variables de la siguiente manera:
Si la variable no entera es ***x***, con un valor no entero ***v***, las dos nuevas variables son:
***x '*** : que es una copia de ***x***, con la modificación del limite superior = $\lfloor \mathbf{v} \rfloor$
***x''*** : que es una copia de ***x***, con la modificación del limite inferior = $\lceil \mathbf{v} \rceil$
Luego genera las restricciones correspondientes a las nuevas variables, y se crea así la regla de bifurcación.

***SubproblemaUsr::generarHijos()***
Genera dos nuevos subproblemas a partir de la bifurcación realizada. Básicamente toma las nuevas reglas generadas con el método detallado anteriormente y completa las estructuras *InfoProblema* con los datos (Variables y Restricciones) para los dos nuevos subproblemas, y actualiza los Pools de Variables y Restricciones. Luego la librería se encargara de generar propiamente las instancias de los dos nuevos subproblemas.

***SubproblemaUsr::preOptimizacionLp()***
Escribe en el log del suproblema que corresponda el detalle del subproblema a optimizar.

***SubproblemaUsr::output()***
Realiza la escritura en el log del subproceso correspondiente los resultados de la optimización del subproblema procesado.

### *main()*

Se muestra el detalle del código principal del programa de ejemplo.

```
int main(int argc, char** argv)
{
	MaestroUsr * m = NULL;

	try
	{
		//
		// Cargar problema TSP
		//
		TSPArg = new TSP( archivoTSP );
		TSPArg->cargarDatos( archivoTSP );

		//
		// Recupera los datos ya cargados, para asi mapearlos a los  pools de Variables y Restricciones
		//
		mapearTSP( TSPArg );


		//
		// Crea objeto MaestroUsr
		//
		m = new MaestroUsr("TestArg", false, false, MINIMIZAR);

		//
		// Carga las Variables y Restricciones en la libreria.
		//
		m->inicializarPool(poolVariable, poolRestriccion);

		//
		// Optimiza el problema global.
		//
		m->optimizar();

		//
		// Muestra la Solucion
		//
		m->output();

		//
		// Muestra los solucion
		//
		mostrarSolucionTSP( m->getSolucion() );

		delete(m);

		return 0;
	}
	catch(Excepcion& e)
	{
		log(0, "%s\n", e.mostrarError() );
		return -1;
	}
	catch(...)
	{
		log(0, "Excepcion desconocida.\n");
		return -1;
	}
}
```

## CÓDIGO FUENTE

Los archivos fuentes con el código del ejemplo se encuentran en el directorio de ejemplo. Para compilarlos seguir los pasos ya explicados en el Manual de Usuario (APÉNDICE) en configuración del Makefile.

*mainTSP.cpp*:
Programa principal, carga el problema real, realiza la formulación e optimiza.

*maestro_usr.cpp, maestro_usr.h*
Implementan la clase MaestroUsr, que hereda de la clase Maestro.

*subp_usr.cpp, subp_usr.h*
Implementan la clase SubproblemaUsr, que hereda de la clase Subproblema.

*TSPproblema.cpp, TSPproblema.h*:
Clase TSP, que implementa la formulación del problema.

*TSPvar.cpp TSPvar.h* :
Clase VarTSP, representa las variables del problema TSP, es utilizada por la clase TSP.

*TSPcorte.cpp TSPcorte.h* :
Clase CorteTSP, representa los cortes en el problema TSP.

*CVRPSEPxxx.cpp CVRPSEPxxx.h* :
Varios archivos fuentes que implementan la generación de cortes. Son utilizados por la Clase CorteTSP. Estos fuentes son provistos por CVRPSEP **[20]**.

*arg8.tsp*
Archivo de ejemplo con los datos de entrada de un problema TSP.

*PBC_arbol_AAMMDD_HHMMSS_9999.xml*
Archivo de resultados, contiene el árbol del resultado (en formato XML) de la optimización del problema TSP de data *arg8.tsp*.

## EJECUCIÓN DEL EJEMPLO

Tomando como una primera ejecución con nuestro ejemplo, el archivo de datos *arg8.tsp* que contiene 8 ciudades turísticas de la Argentina, podemos realizar un seguimiento de los primeros pasos de la optimización del problema.
Para una correcta interpretación de los resultados, cabe aclarar las siguientes consideraciones:

- Si aplicamos las fórmulas ya detalladas para las variables y restricciones de nuestra representación TSP, llegamos a #Variables 128 = y #Restricciones = 57
- Los cantidad de SubProcesos (para procesar los SubProblema) de la librería se configuró en 4.
- El tipo de búsqueda en el árbol *Branch and Cut*, se configuro en DFS, esto es utilizado por el objeto Maestro para ir generando el árbol.

Los datos de entrada son:

| ***Id*** | ***Ciudad*** | ***Lat*** | ***Long*** |
|---|---|---|---|
| *1* | *Bariloche* | *-41.15* | *-71.3* |
| *2* | *Buenos_Aires* | *-34.6* | *-58.45* |
| *3* | *El_Calafate* | *-50.3333333* | *-72.3* |
| *4* | *Iguazu* | *-25.5666667* | *-54.4333333* |
| *5* | *Jujuy* | *-24.1833333* | *-65.3* |
| *6* | *Mendoza* | *-32.8833333* | *-68.8166667* |
| *7* | *Puerto_Madryn* | *-42.7666667* | *-65.05* |
| *8* | *Ushuaia* | *-54.8* | *-68.3* |

Para poder identificar a que ciudades corresponde una determinada variable, se genera el identificador de la variable ***x*** de la siguiente manera:

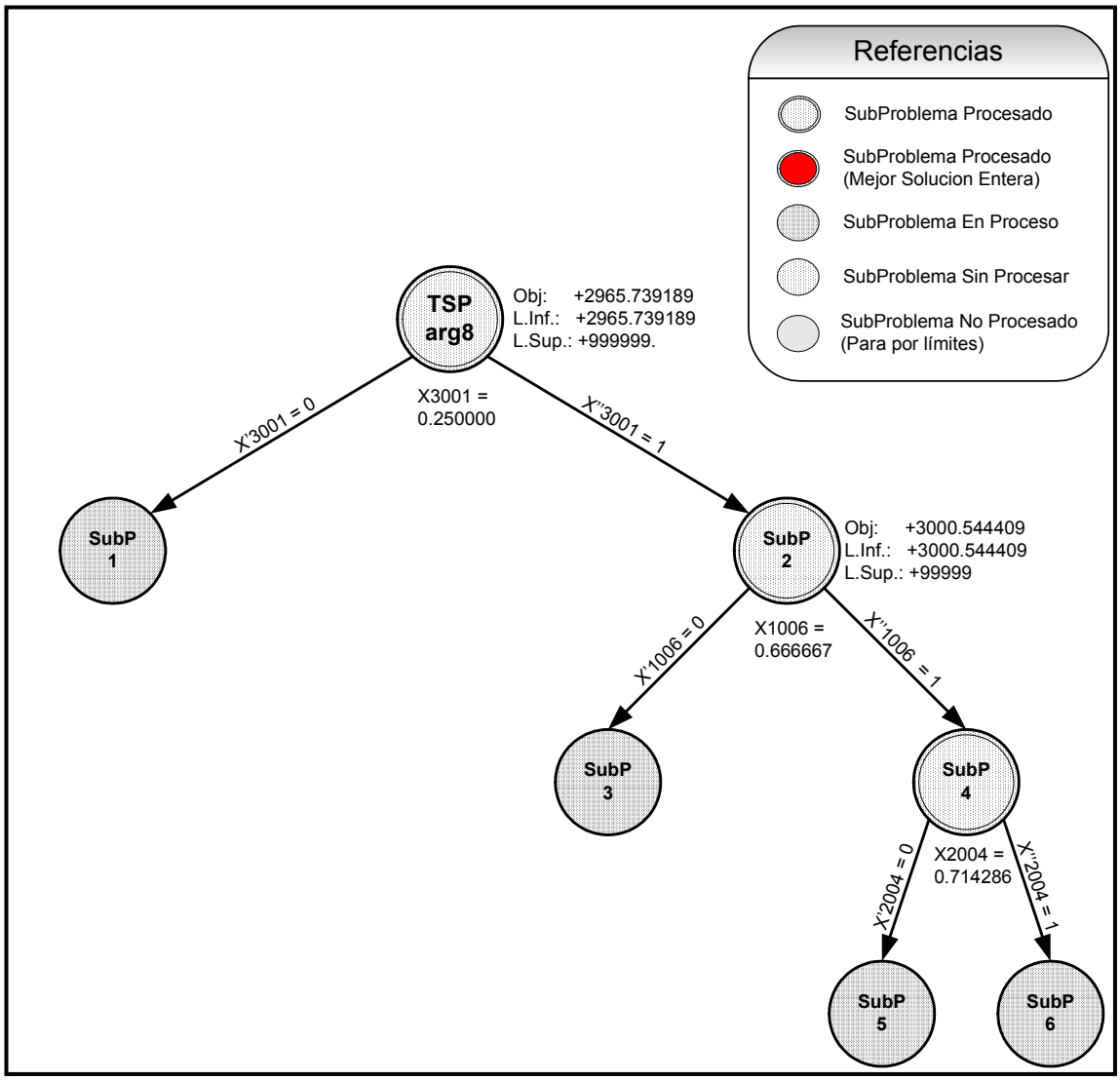
***identificador variable x = <III><JJJ>,*** donde ***III***: Identificador de la Ciudad i.
***JJJ***: Identificador de la Ciudad j.

## ÁRBOL BRANCH AND CUT GENERADO

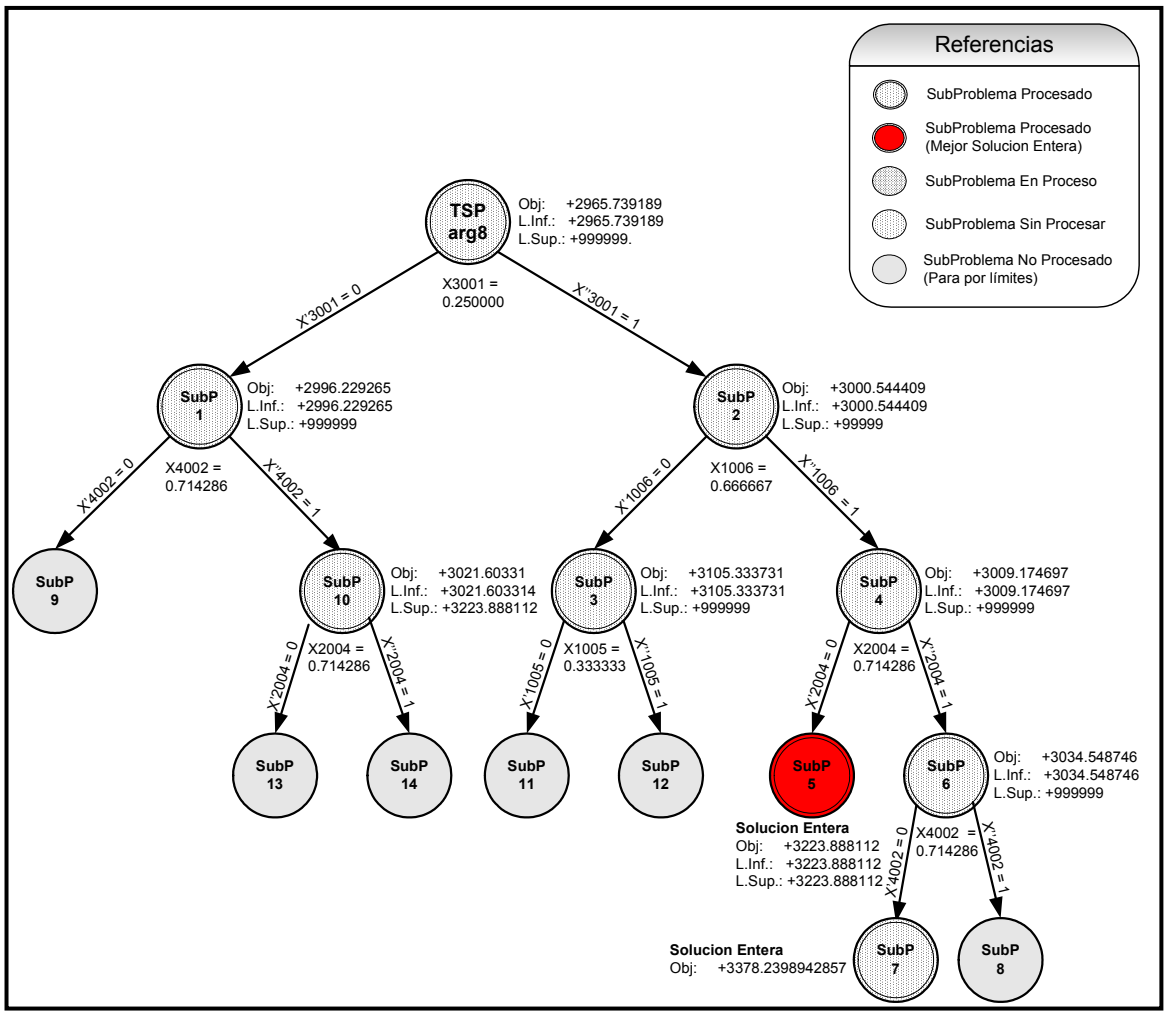
Se muestran dos gráficos del árbol, en el primero se muestra en un instante *t* los subproblemas actuales y sus estados con los resultados parciales.
En un segundo gráfico se puede ver la generación completa del árbol *branch and cut* a partir de la resolución del primer subproblema (problema original *arg8* ).
En cada nodo, se puede ver la información del resultado de cada Subproblema, y la variable utilizada para la bifurcación.

Para un mayor detalle de los valores de cada variable en cada subproblema, se puede observar el archivo del árbol en formato XML, generado por la librería.

## Árbol *Branch and Cut* - Parcial

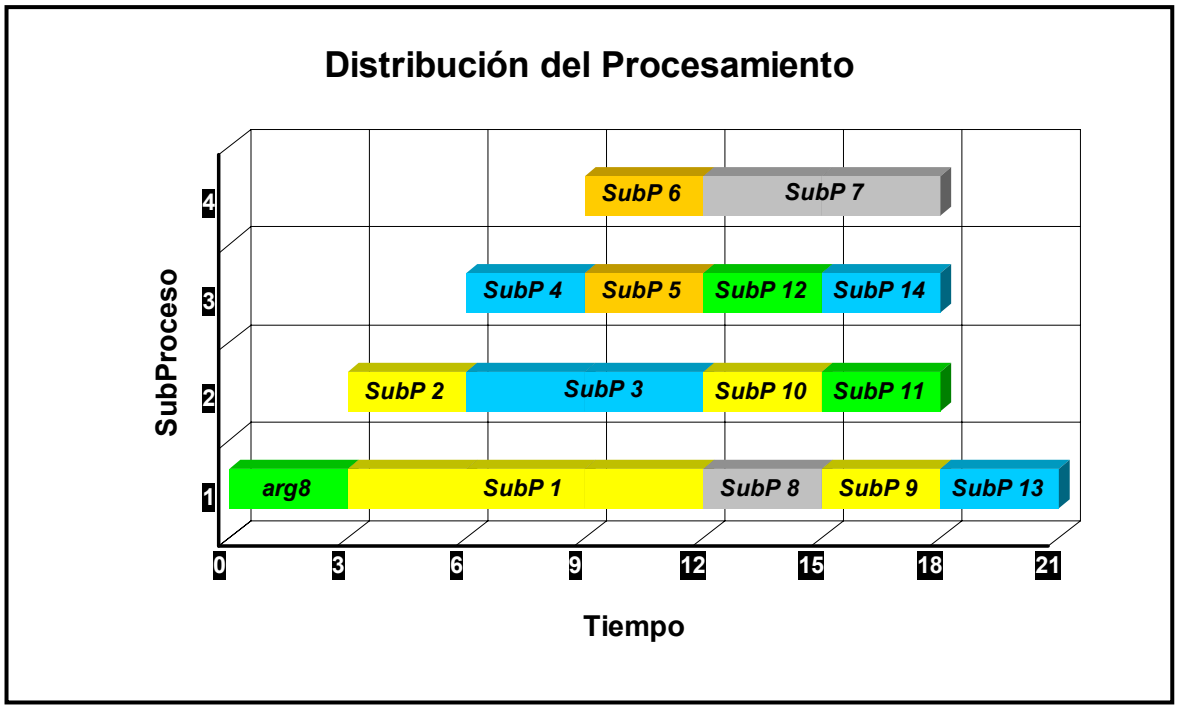


# Árbol Branch and Cut - Final



## DISTRIBUCIÓN DE PROCESAMIENTO

En el siguiente gráfico se muestra la distribución del procesamiento de los subproblemas generados. Se puede observar que el grado de paralelismo aumenta a medida que crece la cantidad de subproblemas a resolver. Por consiguiente la cantidad de SubProcesos debe ser acorde a la magnitud del problema, de lo contrario existirá una sobrecarga de recursos del Sistema Operativo innecesaria.
La escala de tiempos es sólo representativa, así también como la duración de cada subproblema (que incluye el tiempo de análisis de los resultados por el objeto Maestro). Simplemente se quiere mostrar el procesamiento en instante *t*.

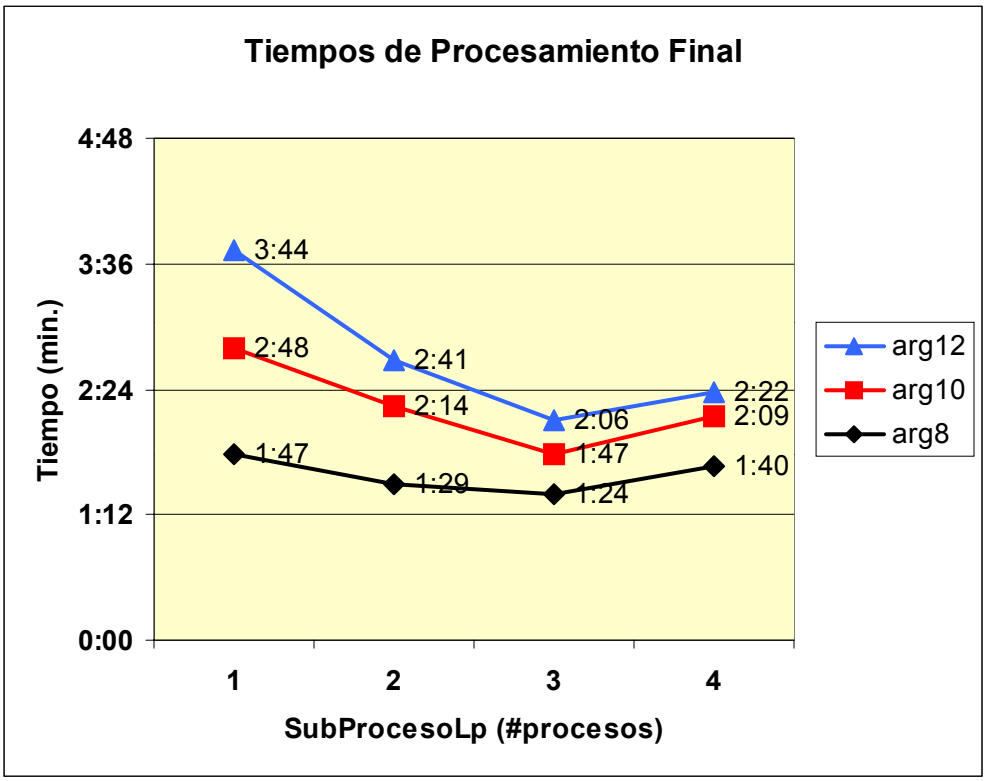


## GRADO DE PARALELISMO

Se realizaron pruebas con tres instancias del problema TSP con diverso grado de paralelismo, variando la cantidad de SubProcesoLp.
Para el análisis de esta información, es importante tener en cuenta el hardware utilizado y los parámetros de configuración de la librería, que son especificados más abajo.
La magnitud de las instancias es la siguiente:

| ***arg8:*** | ***arg10:*** | ***arg12:*** |
|---|---|---|
| #variables = 128 | #variables = 200 | #variables = 288 |
| #restricciones = 57 | #restricciones = 81 | #restricciones = 109 |

### Tiempos de Procesamiento




Observando los datos del gráfico, se pueden realizar las siguientes conclusiones:

- A mayor información a procesar, mayor es la mejora en los tiempos de proceso.
- La óptima cantidad de procesos, depende de la información a procesar.
  En nuestras muestras de instancias pequeñas, se puede observar que si aumentamos a 4 la cantidad de SubProcesoLp, el tiempo final de procesamiento aumenta, esto se debe a que tenemos un *overhead* que supera a los tiempos del procesamiento paralelo.

## Hardware

Las pruebas se realizaron sobre el siguiente hardware:

PC Portátil, 1 Procesador Mobile Pentium III, 850 MHz, RAM: 256 Mb
S.O.: Linux Suse 7.2

## Configuración

Parámetros de configuración utilizados durante la ejecución de las pruebas, se remarcan los parámetros que influyen en los tiempos de procesamiento:

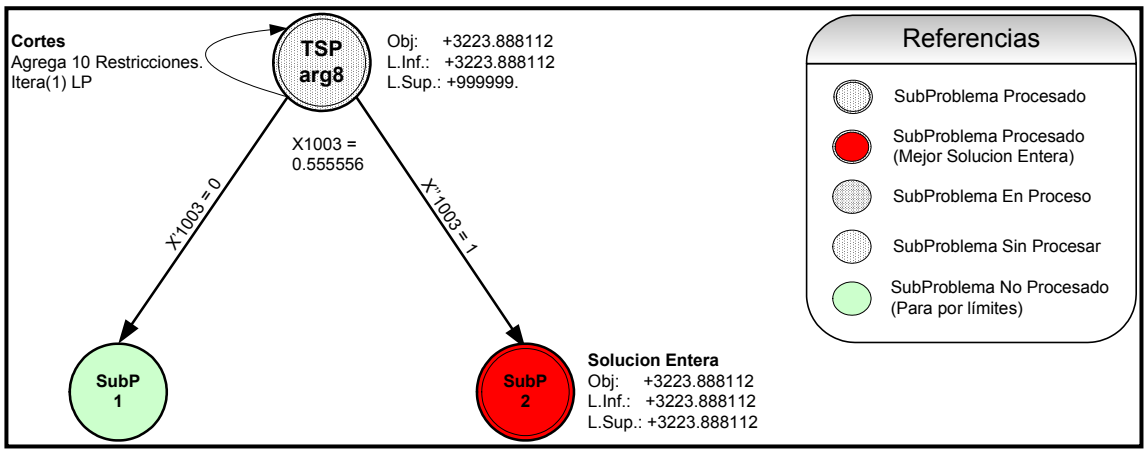
| | |
|---|---|
| username | ldiez |
| **nivelLog** | **2** |
| pathBin | /home/ldiez/tesis/usr/bin |
| pathLog | /home/ldiez/tesis/usr/bin/log |
| **cantSubpEje** | **1 (2, 3, 4)** |
| **cantInfoXSubp** | **2** |
| metodoOpt | p |
| maxIteracionesLp | 1 |
| maxCantNodos | 500 |
| **timeOutCtrl** | **3600** |
| timeOutStart | 600 |
| maxInfoLocal | 1000 |
| porcInfoLocal | 10 |
| maxInfoMemComp | 20 |
| **refreshArbol** | **1800** |
| **timeCkp** | **3600** |
| reintentosMsg | 5000 |

## EJEMPLO CON CORTES

Ejecutamos el mismo ejemplo, pero en nuestra clase ***Subproblema_Usr***, se redefine el método ***separar()*** heredado de la clase *Subproblema*.
El código de este método implementa cortes, agregando nuevas Restricciones al subproblema. Para generar los cortes se utilizan las rutinas CVRPSEP **[20]**.

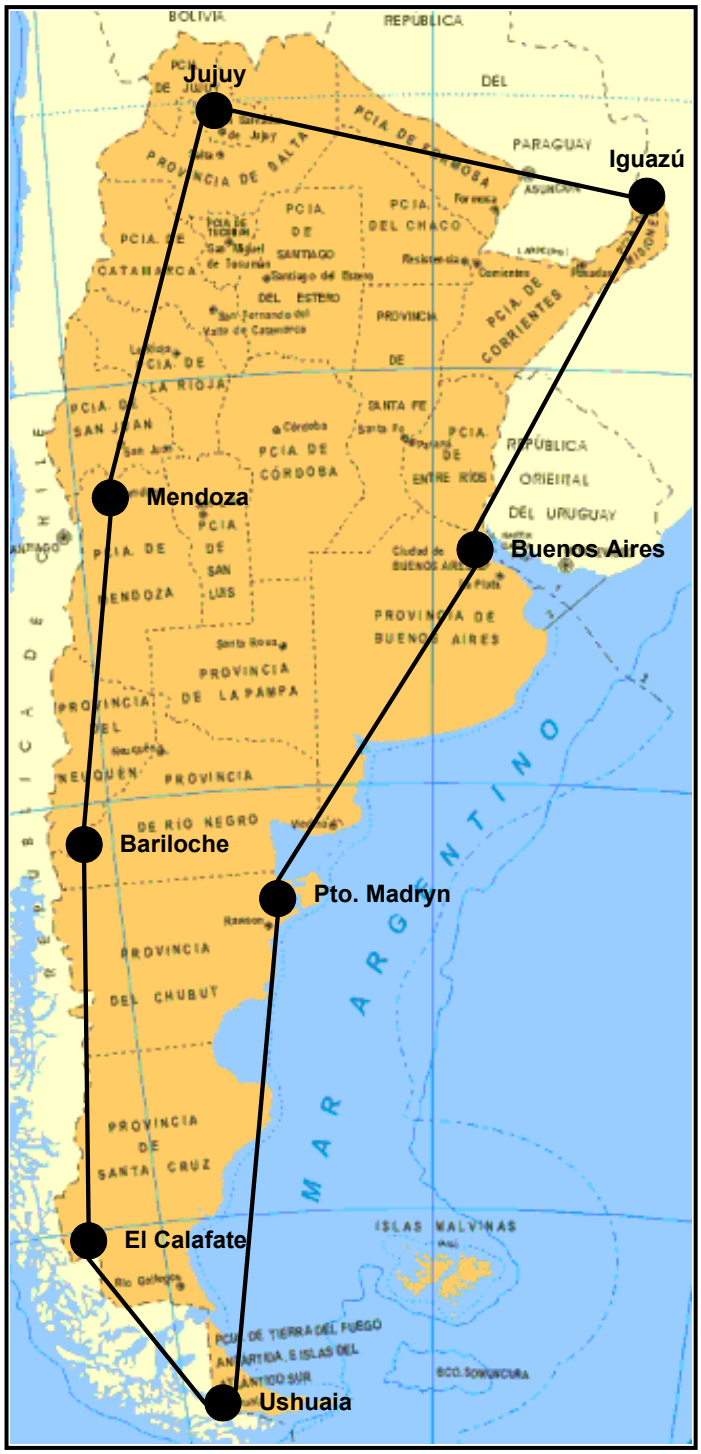
Al aplicar cortes en el problema original (subproblema 0), se puede observar en el árbol la disminución de la cantidad de subproblemas, llegando a la misma solución óptima.

### Árbol *Branch and Cut* - Final



## REPRESENTACIÓN DE LA SOLUCIÓN

La siguiente figura muestra los resultados en el problema real. Es decir se indica el circuito óptimo para recorrer las ciudades del problema.



# CONCLUSIONES

En esta tesis se desarrollo un "framework" para implementaciones personalizadas con procesamiento paralelo. El principal objetivo fue realizar una librería aprovechando los recursos actuales de hardware, y mejorar así los tiempos de procesamientos y el manejo de las soluciones de los tipos de problemas de optimización.

Se estudiaron los tipos de algoritmos existentes para las resoluciones de problemas de optimización, y analizando ventajas y desventajas de los mismos, se implementó un procesamiento paralelo del tipo de algoritmo *branch and cut*, y se enriqueció con funcionalidades muy ventajosas en la práctica.

Se analizaron varias alternativas para la implementación de *branch and cut*, y como conclusión, la librería permite que el usuario implemente su propio código como parte de la librería, es decir, redefiniendo métodos propios de las funcionalidades de *branch and cut*. De esta manera, el usuario utiliza de forma totalmente abstracta el procesamiento paralelo para resoluciones de los subproblemas.

La principal ventaja obviamente es el procesamiento paralelo, donde la ventaja es más importante en instancias de gran escala, cuando el grado de paralelismo tiene una suma importancia frente al *overhead* propio de la comunicación y sincronización entre procesos.
Un punto importante a tener en cuenta para una buena optimización en los tiempos de procesamiento, es el hardware utilizado, ya que frente a nuestro software paralelo, cuando más recursos de hardware existan, mayor será el grado de paralelismo que podemos alcanzar.
En instancias pequeñas se debe alinear muy bien el grado de paralelismo (la cantidad de instancias de SubProcesoLp), de lo contrario el *overhead* del procesamiento es superior a las supuestas ventajas del paralelismo.
Parte del *overhead* mencionado se debe a la mensajería entre procesos, y se puede mejorar alineando los parámetro IPC del Sistema Operativo, ya que cuando alguno de los procesos de la librería no disponen de algún recurso IPC, realiza *n* reintentos hasta conseguirlo. Esto se puede observar en los logs de los procesos.

La persistencia de datos es una importante ventaja, frente a otros software existentes, la existencia de puntos de control automáticos implementa satisfactoriamente esta funcionalidad.

Las funcionalidades de control, provistas para el usuario desde su interfaz, favorecen el tratamiento de las resoluciones de los problemas de optimización.
Con ellas se pueden tener soluciones deseables "parciales" en corto tiempo, realizar procesamientos en forma parcial, y analizar los resultados parciales que se van obteniendo, sin la necesidad de llegar a la mejor solución.

# TRABAJOS FUTUROS

El objetivo de esta tesis esta cumplido, pero podemos describir algunos puntos en los que es posible continuar el desarrollo y obtener nuevos resultados.

- **Grado de paralelismo dinámico**, el grado de paralelismo actual se configura estáticamente por la cantidad de instancias de los SubProcesoLp. Es posible implementar una política de instancias de procesos "a demanda", y de esta manera no existirían instancias ociosas, y la sobrecarga de procesamiento estaría limitada por hardware y no por un parámetro de configuración.

- **Implementar *Branch and Price*** con esta misma arquitectura paralela. De la misma manera que se implementó *Branch and Cut*, es posible generalizar el "framework" para así también cubrir estos otros tipos de algoritmos, utilizando las mismas ventajas ya implementadas.

- **Disminuir el *overhead*** entre procesos, principalmente en la administración de las Variables y Restricciones de *Branch and Cut*. En la actualidad existe un *overhead* importante entre el proceso AdmPool y los SubProcesoLp como mensajes de datos.

# BIBLIOGRAFIA


**[1]** **LA Wolsey, *Integer Programming***
Wiley-interscience series in discrete mathematics and optimization

**[2]** **CPLEX - Using the CPLEX Callable Library - Version 6.0**
ILOG, Inc. CPLEX Division (1997) http://www.cplex.com

**[3]** **CPLEX - Advanced Reference Manual**
ILOG CPLEX 7.1 (2001) http://www.ilog.com

**[4]** **Stefan Thienel, *ABACUS - A Branch-And-CUt System - PhD thesis.***
UniversitÄat zu KÄoln, (1995)

**[5]** **ABACUS - A Branch-And-CUt System - User's Guide and Reference Manual**
Universität zu Köln, Universität Heidelberg (1999)

**[6]** **T.K. Ralphs, L. Ladányi, *Computational Experience with Branch, Cut, and Price.***

**[7]** **T.K. Ralphs, L. Ladányi, *SYMPHONY: A Parallel Framework for Branch and Cut.***
Rice University (1999)

**[8]** **T.K. Ralphs, L. Ladányi, *SYMPHONY 4.0 User's Manual***
(2003) http://www.branchandcut.org/SYMPHONY

**[9]** **T.K. Ralphs, L. Ladányi, *SYMPHONY Developed***
(2003)

**[10]** **T.K. Ralphs, L. Ladányi, L.E. Trotter Jr., *Branch, Cut, and Price: Sequential and Parallel***

**[11]** **T.K. Ralphs, *Course Project Description - IE418 – Discrete Optimization***
http://www.lehigh.edu/~tkr2 http://www.lehigh.edu/~tkr2/courses/ie418/

**[12]** **DIMACS Workshop: Theory and Practice of Integer Programming in honor of Ralph E. Gomory on the Occasion of his 70th Birthday**
August 2 - 4, 1999 - IBM Watson Research Center, Yorktown Heights, NY

**[13]** **Michael Trick, *Simple Walkthrough for Using SYMPHONY***
(2003)

**[14]** **Charles Audet Gilles Savard Walid Zghal,** ***New Branch-and-Cut Algorithm for Bilevel Linear Programming***
(2004) http://www.gerad.ca/Charles.Audet

**[15]** **Spring School on Computational Combinatorial Optimization**
SchloB Dagstuhl, Germany, (2000)

**[16]** **A. J. Hoffman and P. Wolfe**, ***History in The Traveling Salesman Problem***
Lawler, Lenstra, Rinooy Kan and Shmoys, eds., Wiley, 1-16. (1985)

**[17]** **D. Applegate, R.Bixby, V.Chvátal and W.Cook,** ***On the Solution of Traveling Salesman Problems***
(1998)

**[18]** **G. B. Dantzig, R. Fulkerson, and S. M. Johnson,** ***Solution of a large-scale traveling salesman problem***
*Operations Research* 2 (1954), 393-410.

**[19]** **National Imagery and Mapping Agency**
http://earth-info.nga.mil/gns/html/

**[20]** **Jens Lysgaard,** ***CVRPSEP: A package of separation routines for the Capacitated Vehicle Routing Problem***
Department of Management Science and Logistics, Aarhus School of Business, Denmark, (2004)

# APÉNDICE

Este apéndice contiene el ***Manual de Usuario*** de la librería, con el mismo se quiere mostrar como utilizar un armazón ("framework") poderoso para desarrollos personalizados de algoritmos tipo *branch and cut*, aplicando el poder del procesamiento paralelo a las resoluciones de los nodos del árbol de búsqueda.
Ver documento ***PBC_Manual_v22.pdf***

*Tesis de Licenciatura*

# *Software de Multiprocesamiento Paralelo para implementación de "Branch And Cut" (Parallel Branch and Cut)*

# *Manual de Usuario*

***Luis Rubén Diez***

*Ciencias de la Computación*
*Departamento de Computación*
*Facultad de Ciencias Exactas y Naturales*
*Universidad de Buenos Aires*

*Director: Dr. Min Chih Lin*
*Codirectora: Lic. Paula Lorena Zabala*

*Diciembre 2004*

# *ÍNDICE GENERAL*

**INTRODUCCION** ..... **4**

**ARQUITECTURA** ..... **4**

**Maestro** ..... **4**

**SubProcesoLp** ..... **5**

**AdmPool** ..... **5**

**AdmSubProcesoLp** ..... **5**

**Interfaz De Usuario** ..... **5**

**Diseño De Arquitectura (Fig. 1)** ..... **6**

**INSTALACION** ..... **7**

**Contenidos** ..... **7**

**Pasos de Instalación** ..... **7**

**DISEÑO** ..... **10**

**Diagrama de Clases** ..... **10**
Diseño de Clases de Programación Lineal (Fig.2) ..... 10
Diseño de Clases de Los Procesos (Fig.3) ..... 11

**CLASES** ..... **12**

**Maestro** ..... **12**
Descripción: ..... 12
Métodos Públicos ..... 12

**Subproblema** ..... **14**
Descripción ..... 14
Métodos Públicos ..... 14

**Variable** ..... **19**
Descripción ..... 19
Atributos ..... 19
Métodos publicos ..... 19

**Restriccion** ..... **22**
Descripción ..... 22
Atributos ..... 22
Métodos publicos ..... 23

**Pool** ..... **25**
Descripción ..... 25
Métodos publicos ..... 26

**Reglas Branch** ..... **27**
Descripción ..... 27
Atributos ..... 27
Métodos publicos ..... 28

**Info Problema** ..... **29**
Descripción ..... 29
Atributos ..... 29
Métodos publicos ..... 29

**Info Res Var** ..... **30**
Descripción ..... 30
Atributos ..... 30
Métodos publicos ..... 30

**Solucion (Estructura)** ..... **30**
Descripción ..... 30
Atributos ..... 30

**Solucion (Clase)** ..... **31**
Descripción ..... 31
Atributos ..... 31
Métodos públicos ..... 31

**AdmInfo** ..... **32**
Descripción ..... 32
Métodos Principales ..... 32

**Nodo** ..... **33**
Descripción ..... 33
Métodos Principales ..... 33

**ArbolBq** ..... **35**
Descripción ..... 35
Métodos Principales ..... 35

**Excepcion** ..... **37**
Descripción ..... 37
Métodos Principales ..... 37

**Constantes** ..... **37**
Descripción ..... 37

**Enumerados** ..... **37**
Descripción ..... 37

**Clase Maestro** ..... **39**

**Clase Subproblema** ..... **39**

**PROCESOS ..... 40**

**Proceso Usuario ..... 40**

**Proceso Administrador de Pool ..... 41**

**Proceso Administrador de procesos SubProcesoLp ..... 41**

**Proceso SubProcesoLp ..... 41**

**Proceso Interfaz de Usuario ..... 41**

**Comunicación Entre Procesos ..... 42**
Estructura de Colas (Fig.4) ..... 43

**CONFIGURACION ..... 44**

**Parámetros ..... 44**

**Niveles de log por Tipos de mensajes ..... 45**

**INTERFAZ DE USUARIO ..... 46**

**Opciones del Problema ..... 47**
Comienzo de Procesamiento ..... 47
Solución Global ..... 47
Arbol de Branch and Cut ..... 47
Estados de Subproblemas ..... 49

**Opciones de Control ..... 50**
Control de Ejecución ..... 50
Configuración ..... 52
Logs ..... 53
Backups ..... 54

**Archivo de Inicialización ..... 55**
Parámetros ..... 55

**UTILITARIOS AUXILIARES DE CONTROL ..... 56**
comenzar.sh ..... 56
terminar.sh ..... 56
ps-pbc.sh: ..... 56

**EJEMPLO ..... 56**

# INTRODUCCION

La librería **P**arallel **B**ranch and **C**ut, tiene como objetivo realizar la implementación de optimización de problemas enteros, mediante algoritmos Branch and Cut con procesamiento paralelo de los subproblemas. Para resolver las relajaciones utiliza la librería Cplex, por consiguiente es indispensable tener dicha librería previamente instalada en el equipo.

Para implementar el uso de la librería es necesario realizar los siguientes desarrollos:

- Crear una clase SubproblemaUsr (se debe respetar este nombre), que hereda las funcionalidades de la clase Subproblema de la librería, y redefinir los métodos que el usuario crea conveniente.
- (Opcional) Crear una clase MaestroUsr (nombre opcional), que hereda de la clase Maestro de la librería.
  Un objeto de esta clase se utilizará en el programa principal construido por el usuario, y de no desarrollarse esta clase, se utilizará directamente un objeto de la clase Maestro.

La librería esta implementada en el Lenguaje C++, y compilada con el compilador gcc, versión 2.95.3. Utiliza IPC para el manejo de mensajería entre procesos, tanto de control como de datos.

# ARQUITECTURA

Se detallan los principales componentes de la arquitectura de la librería.

## MAESTRO

Es un objeto de la Clase Maestro. Tiene el conocimiento del problema global, administra la información para cada subproblema a optimizar, y mantiene la solución global factible. También realiza la inicialización de los Pools de Variables y Restricciones del problema.
La información del problema es comunicada a los procesos SubProcesoLp a través de memoria compartida, completando el input a procesar para cada uno, y levantando los resultados de cada uno.
Realiza puntos de control de la información procesada, bajando a archivos toda la información necesaria, para luego poder continuar con el procesamiento, reusando las soluciones parciales ya encontradas. Los intervalos de estos puntos de control, son configurables con un parámetro de la librería.
Posee comunicación de control entre los procesos Administradores, y también recibe las ordenes de las opciones de control de la *Interfaz de Usuario.*

**SUBPROCESOLP**

Es un proceso SubProcesoLp, tiene la funcionalidad de optimizar un problema en particular. El input con la información del problema es leído de memoria compartida, y luego le solicita al proceso Administrador de Pool (AdmPool) las variables y restricciones necesarias.
Como resultado del proceso, puede existir una mejor solución global, la generación de información para nuevos subproblemas, hijos de subproblema procesado, o cerrar la rama correspondiente, ya sea porque se encontró una solución entera o porque el valor de la solución obtenida es mayor que la mejor solución actual.
Existen varias instancias de este proceso, la cantidad es configurable por la librería.
Cada una de estas instancias resuelven los subproblemas que son generados a partir del problema global.

**ADMPOOL**

Es un proceso Administrador de Pool, tiene la funcionalidad de administrar todas las variables y restricciones del problema global.
Recibe pedidos de variables y restricciones de cada una de las instancias de los procesos SubProcesoLp.

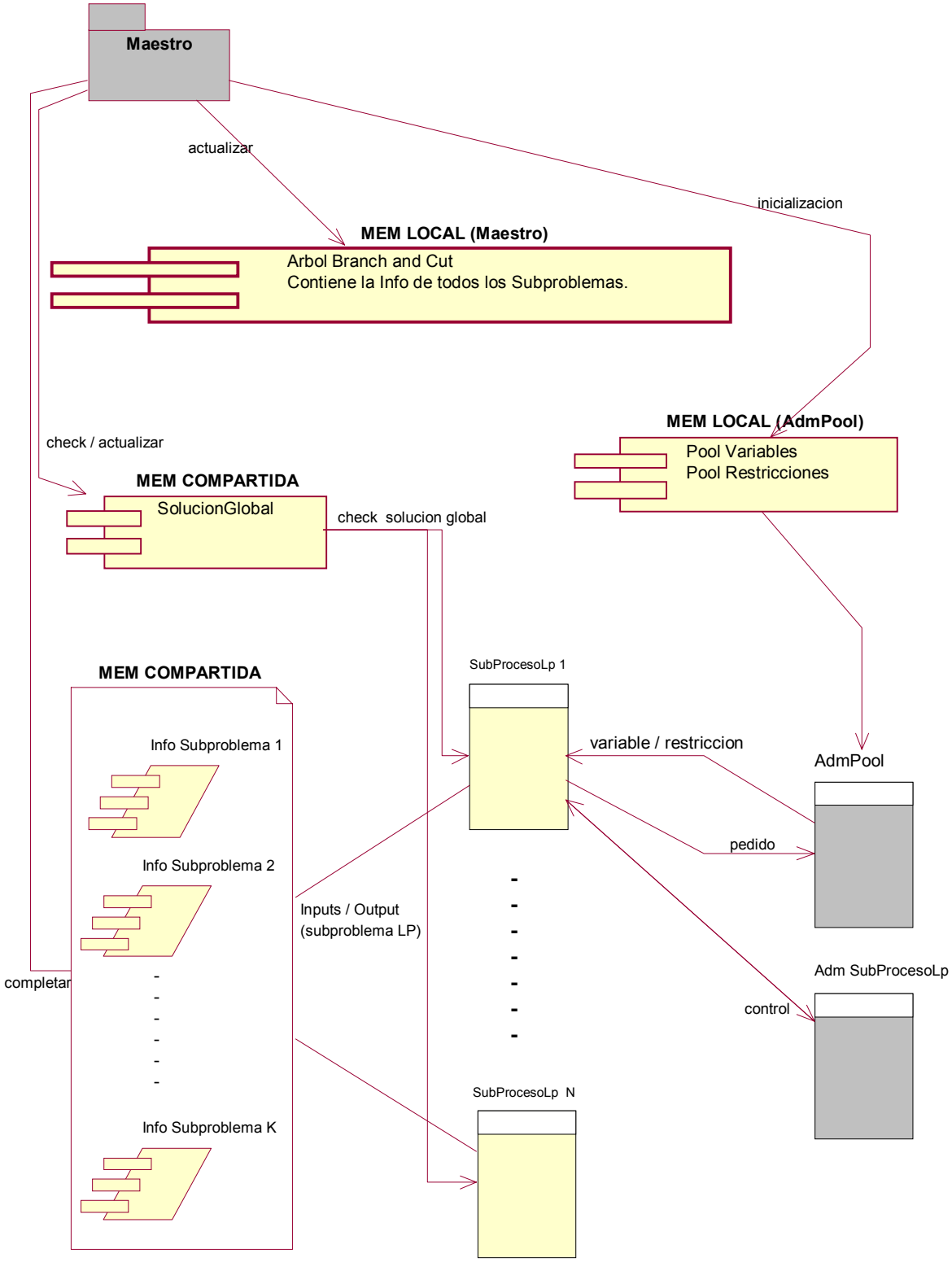
**ADMSUBPROCESOLP**

Es un proceso Administrador de procesos SubProcesoLp, tiene la funcionalidad de administrar los procesos SubProcesoLp. Tiene el control de cada instancia, existe entre ellos una comunicación exclusivamente de control.

**INTERFAZ DE USUARIO**

Es un proceso independiente de los procesos de la librería, el mismo provee una interfaz de usuario para una mayor flexibilidad en el manejo de la resolución del problema.

## DISEÑO DE ARQUITECTURA (FIG. 1)



# INSTALACION

## CONTENIDOS

- **Librería PBC:** archivo ***libpbc.a***, que tiene las funcionalidades de la librería (con este archivo se debe vincular el programa del usuario).

- **Interfaces (*.h*):** Interfaces de la funcionalidad de la librería, utilizados para compilar el programa de usuario.
  También existe un archivo de interface que se debe copiar como modelo de la clase a desarrollar por el usuario (**subp_usr.h**) .

- **Ejecutables:** archivos binarios ejecutables que utiliza la librería:
  Proceso administrador de los procesos SubProcesoLp (***proc_admsubp***).
  Proceso administrador de los Pools de variables y restricciones (***proc_admpool***).
  Proceso interfaz de usuario (***pbc_menu***).

- **Pre-Ejecutable:** archivo **proc_subp.o**, es la compilación del proceso SubProcesoLp. Este archivo se debe vincular con el objeto generado de la clase SubproblemaUsr, (creada por el usuario.), junto con la librería libpbc.a.

- **Configuración:** archivo de configuración utilizado por los procesos de la librería (libpbc.cfg).

- **Inicialización:** archivo de inicialización utilizado por el proceso interfaz de usuario (libpbc.ini).

- **Utilitarios:** make y scripts varios para chequeos de estados de procesos,

- **Ejemplos:** programas de ejemplos.

- **Documentación:** documentación de teoría y manual de usuario de esta tesis.

## PASOS DE INSTALACIÓN

1. **Crear directorio** de instalación, ***<dir instalación>*** por ejemplo ***/opt/pbc/***

2. **Copiar** el paquete de instalación
   En el directorio de instalación, recientemente creado, hacer:
   - Copiar el archivo de instalación al directorio de instalación.
     ***cp <dir origen>\libpbc.2.2.tar.gz <dir instalación>***
   - Descomprimir el archivo:
     ***gunzip libpbc.2.2.tar.gz***

- Extraer los archivos:
  ***tar -xvf libpbc.2.2.tar***

Una vez realizados estos pasos, se genera la siguiente estructura de directorios:
<dir instalación>/lib/
/include/
/bin/
/obj/
/util/
/ejemplos/
/doc/

**3. Ambiente de desarrollo**

Definimos el directorio de trabajo ***<dir trabajo>*** al directorio de desarrollo del usuario, se asume que dicho directorio tiene la siguiente estructura de subdirectorios:
<dir trabajo>/src/
/include/
/obj/
/bin/
/bin/backups/
/log (es una sugerencia, se puede crear en cualquier lugar)
En el momento de ejecución del proceso de usuario, deben estar inicializadas las variables de entorno CPLEXLICDIR y LD_LIBRARY_PATH, esta última es utilizada por la librería libxml2.

**4. Makefile**

Copiar el archivo <dir instalación>/util/**make-usr.mk** a <dir trabajo>.
Editar el archivo make-usr.mk y modificar las siguientes variables:

USR_MAIN = <nombre del programa fuente principal del usuario, sin ext.>
Por ejemplo: USR_MAIN = mainTSP

USR_SUBP = <nombre del archivo fuente de la clase SubproblemaUsr >.
Por ejemplo: USR_SUBP = subp_usr

USR_MAESTRO = <nombre del archivo fuente de la clase MaestroUsr >.
Por ejemplo: USR_MAESTRO = maestro_usr

PATH_PBC = <nombre del directorio de instalación de la librería PBC>
Por ejemplo: PATH_PBC=/opt/pbc

Si se desea compilar con otros objetos necesarios para los programas del usuario, se deben agregar a las siguientes variables:

SUBP_OBJS = <objetos para la generación del proceso proc_subp >.

USR_OBJS = <objetos para la generación del proceso principal del usuario>.

Con estas variables el "make" estaría en condiciones de ser utilizado, hacer:
***make -f make-usr.mk [all/clean]***

Luego de esta compilación deberían estar generado el archivo binario del proceso de usuario en el directorio bin.

5. **Archivo de Configuración**
   Copiar el archivo <dir instalación>/util/**libpbc.cfg** al directorio <dir trabajo>/bin
   Editar el archivo libpbc.cfg y modificar las variables que sean necesarias.
   (ver**CONFIGURACION**)

6. **Archivo de Inicialización**
   Copiar el archivo <dir instalación>/util/**libpbc.ini** al directorio <dir trabajo>/bin
   Editar el archivo libpbc.ini y modificar las variables que sean necesarias.
   (verArchivo de Inicialización)

7. **Procesos de Librería**
   Copiar los archivos binarios de los procesos que utiliza la librería, hacer:
   - copiar <dir instalación>/bin/**proc_admsubp** a <dir trabajo>/bin
   - copiar <dir instalación>/bin/**proc_admpool** a <dir trabajo>/bin
   - copiar <dir instalación>/bin/**pbc_menu** a <dir trabajo>/bin

   (ver **PROCESOS**)

8. **Include modelo.**
   Copiar el archivo de interface para la clase SubproblemaUsr a desarrollar:
   - copiar <dir instalación>/util/**subp_usr.h** a <dir trabajo>/include

   Tener en cuenta que se deben respetar los nombres de la clase y métodos que existan en el archivo.

9. **Scripts auxiliares utilitarios**
   Copiar el archivo <dir instalación>/util/*.sh al directorio <dir trabajo>/bin.
   ( ver **UTILITARIOS AUXILIARES DE CONTROL**)

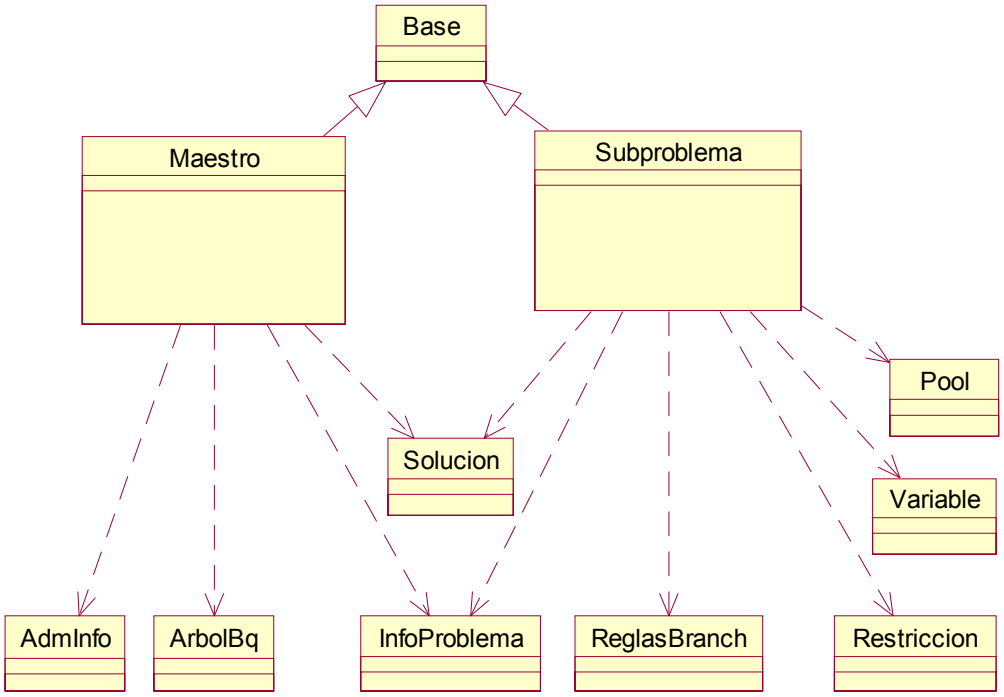
# DISEÑO

Se define un diseño orientado a objetos, utilizando jerarquía de clases implementando herencia simple y composición. El diseño completo contiene más de 30 clases, a continuación se diagrama solo las más importantes para dar una idea general.
El diseño se divide en tres módulos:

## DIAGRAMA DE CLASES

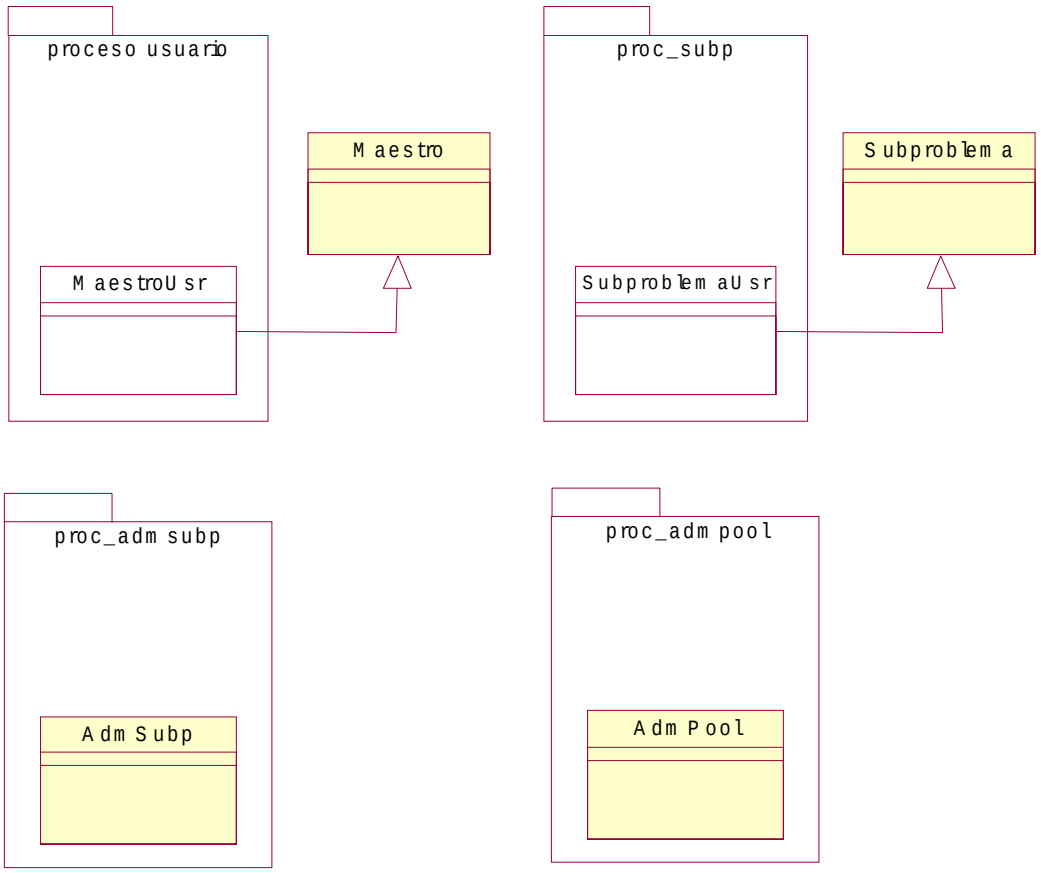
- **Programación Lineal**
  Representa el grupo de clases que definen las funcionalidades de la programación lineal. En este diagrama se representan solo las principales.

**DISEÑO DE CLASES DE PROGRAMACIÓN LINEAL (FIG.2)**



- **Aplicaciones**
  Representa las clases que forman parte de un proceso. El diagrama representa los programas principales (procesos) y las clases que estos involucran.

  **DISEÑO DE CLASES DE LOS PROCESOS (FIG.3)**




- **Auxiliar**
  Representa las clases que contiene funcionalidades más particulares, principalmente de control y administración de recursos.

# CLASES

## MAESTRO

**DESCRIPCIÓN:**

Es la clase principal de la funcionalidad de optimización, crea el problema original, y posee el control sobre los subproblemas que se generan durante el procesamiento. Controla los subproblemas a nivel de información de optimización, y no de instancias de subproblemas, (esta última funcionalidad la tiene el Administrador de SubProcesosLp).
Tiene una visión global del problema. Con la finalización de cada subproblema, genera los nuevos subproblemas hijos y así va construyendo el Árbol de Branch and Cut.

Posee un objeto de la clase AdmInfo que se encarga del manejo de memoria, tanto en forma local como compartida para los demás procesos. Esta clase posee parámetros de performance configurables.

También es la encargada de levantar las instancias de los procesos Administradores, levanta un proceso administrador de SubProcesosLp y un administrador de pool.

**MÉTODOS PÚBLICOS**

### Maestro

Crea un objeto maestro para un determinado problema. El problema esta vacío, solo se conoce el nombre.

***Maestro*** *(char * nombreProblema, bool cutting, bool pricing, SENTIDO sentido)*

- *nombreProblema*: nombre del Problema
- *cutting*: boolean que indica si se hace cortes, es decir si realiza llamadas a las rutinas de separación.
- *pricing*:
- *sentido*: indica el sentido de la optimización ( MAXIMIZAR / MINIMIZAR ).

### inicializarPool

Inicializa los datos del problema cargando todas las variables y restricciones originales. También crea el primer subproblema, es decir la raíz del árbol.

*void* ***inicializarPool****(Pool<Variable>& poolVar, Pool<Restriccion>& poolRes )*

- *poolVar:* Referencia al pool de las variables originales del problema.
- *poolRes:* Referencia al pool de las restricciones originales del problema.

### optimizar

Optimiza el problema global. Distribuye el procesamiento de cada subproblema en forma paralela entre los procesos SubProcesoLp (proc_subp) y arma el árbol de Branch and Cut.

Mientras procesa se pueden observar los resultados de las soluciones parciales de cada subproblema. Una vez finalizada esta operación, se pueden ver los resultados de la solución global y el árbol generado.

*void* ***optimizar()***

**outputParcial**
Muestra la solución de cada subproblema que se termina de procesar. Este método es invocado desde el método *optimizar()* por cada subproblema que finaliza.

*virtual void* ***outputParcial****(InfoProblema& info)*
- *info*: estructura de datos de la información del problema.

**getNombreProblema**
Retorna el nombre del problema global.

*char* ******* ***getNombreProblema()*** *const*

**cantSubp**
Retorna la cantidad de subproblemas generados, ya procesados o pendientes de procesamiento.

*int* ***cantSubp()*** *const*

**limiteInf**
1º instancia: Inicializa el valor del Límite Inferior Global con valor l.
2º instancia: Retorna el valor del Límite Inferior Global.

*void* ***limiteInf****(double l)*
*double* ***limiteInf()*** *const*

**limiteSup**
1º instancia: Inicializa el valor del Límite Superior Global con valor l.
2º instancia: Retorna el valor del Límite Superior Global.

*void* ***limiteSup****(double l)*
*double* ***limiteSup()*** *const*

**limitePrimal**
1º instancia: Inicializa el límite primal con el valor *l*.

2º instancia: retorna el límite primal del problema.

*void* ***limitePrimal****(double l);*
- *l:* valor de límite a inicializar.

*double* ***limitePrimal()*** *const*

**limiteDual**
1º instancia: inicializa el límite dual con el valor *l*.
2º instancia: retorna el límite dual del problema.

*void* ***limiteDual*** *(double l);*
- *l:* valor de límite a inicializar.

*double* ***limiteDual()*** *const*

**setTipoBusqueda**
Inicializa el tipo de búsqueda en el Arbol con el modo *b*.
*void* ***setTipoBusqueda****( MODO_BUSQUEDA b )*
- *b:* constante que identifica el tipo de búsqueda: (BFS):Best, (DFS): Depth

**getTtipoBusqueda**
Retorna el tipo de recorrido utilizado en el árbol para seleccionar el próximo subproblema a procesar.

*MODO_BUSQUEDA* ***getTipoBusqueda()***

## SUBPROBLEMA

### DESCRIPCIÓN

Esta clase es la encargada de la optimización de un subproblema en particular, utiliza Cplex para resolver la relajación. Genera los nuevos subproblemas a partir de las reglas de ramificación (branch).
Luego de la relajación, en base a la solución parcial y a los límites globales, decide si se deben generar o no los nuevos subproblemas. Si se deben generar, lo hace a partir de las reglas de "branch".
Contiene la funcionalidad del procesamiento en cada nodo del árbol. Posee métodos virtuales que el usuario puede redefinir.

### MÉTODOS PÚBLICOS

**Subproblema**
Construye un objeto subproblema y envía mensajes de control al proceso Administrador de SubProcesoLp. También crea el entorno de optimización de Cplex.

***Subproblema****( int argc, char** argv );*

- *argc:* cantidad de argumentos del proceso proc_subp
- *argv:* puntero a los argumentos.

**inicializar**
Crea un nuevo subproblema inicializando variables y restricciones.
Solicita al proceso Administrador de Pool las variables y restricciones para el problema a optimizar.

*void* ***inicializar()***

**optimizar**
Resuelve las relajaciones del problema actual. Internamente invoca al método *separar()* e itera n veces según la cantidad de cortes del resultado del *separar()*.

*void* ***optimizar()***

**getPoolVar**
Referencia al pool de variables del problema actual.

*Pool<Variable>&* ***getPoolVar()***

**getPoolRes**
Referencia al pool de restricciones del problema actual.

*Pool<Restriccion>&* ***getPoolRes()***

**separar (virtual)**
Genera los planos de cortes, agregándolos al pool de cortes. Internamente al finalizar la generación de los planos de cortes, se debe invocar el método *agregarCortes()*.

*virtual int* ***separar()***
Retorna la cantidad de planos de corte generados.

**generarReglasBranch (virtual)**
Crea reglas para ramificar. En base a cierta condición genera nuevas variables y/o restricciones para realizar así la bifurcación al momento de generar los nuevos subproblemas.

*virtual int* ***generarReglasBranch****()*
Retorna: 0, si genero alguna regla.

1, caso contrario.

**generarHijos (virtual)**
Genera la información para los nuevos subproblemas, genera uno por cada regla de "branch" que exista.
No genera las instancias de los subproblemas, sino genera la información necesaria para que luego el objeto Maestro genere las instancias.

*virtual void* ***generarHijos()***

**preOptimizacionLp (virtual)**
Muestra el problema actual previo a la optimización, este método es invocado inmediatamente previo al método *optimizar()*.

*virtual void* **** preOptimizacionLp()***

**output (virtual)**
Muestra los resultados del problema actual. Datos de la solución parcial. Este método es invocado inmediatamente después del método *optimizar()*.

*virtual void* **** output()***

**mostrarProblema**
Retorna un cadena con formato, mostrando el contenido del problema cargado en el Cplex.

*char* **** mostrarProblema()***

**idSubproblema**
Retorna el identificador del subproblema actual.

*int* ***idSubproblema()***

**problema**
Retorna el nombre del subproblema actual.

*char* **** problema()***

**metodoOpt**

1º instancia: inicializa el método de optimización según Cplex, por defecto el método se inicializa con el valor del archivo de configuración.
2º instancia: retorna el método de optimización según Cplex.

*void* ***metodoOpt****( char metodo )*
- *metodo:* p: primopt()
  d: dualopt()
  h: hybnetopt()
  i: mipopt()
  b: baropt()
  y: hybbaropt()

*char* ***metodoOpt()***

**nivel**
Nivel del problema actual en el árbol de optimización.

*int* ***nivel()***

**estado**
Retorna el estado del proceso de optimización.

*ESTADO_SP* ***estado()***

**estadoOpt**
Retorna el estado de optimización de subproblema actualmente procesado.

*ESTADO_OPT* ***estadoOpt()***

**limiteInf**
Retorna el límite Inferior del subproblema actual.

*double* ***limiteInf()***

**limiteSub**
Retorna el límite Superior del subproblema actual.

*double* ***limiteSup()***

**limiteDual**
Retorna el valor de la relajación del subproblema actual.

*double* ***limiteDual()***

**getSolucion**
Retorna la solución del subproblema actual.

*Solucion&* ***getSolucion()***

**reglasBranch**
Retorna un puntero a las Reglas de Branch ( a un objeto de la clase ReglasBranch ).

*ReglasBranch* **** reglasBranch()***

**infoResVar**
Retorna la referencia a la información de Restricciones y Variables del subproblema actual.

*InfoResVar&* ***infoResVar()***

**setMaxIteraciones**
Inicializa la cantidad máxima de iteraciones a realizar en la optimización. Por defecto se inicializa el valor del archivo de configuración.

*void* ***setMaxIteraciones****(int i)*

- *i:* entero que representa la cantidad máxima.

**agregarCortes**
Agrega el Pool de Cortes al Pool de Restricciones al problema. Se utiliza en el método *separar()* para agregar los cortes generados al Pool del problema.

*int* ***agregarCortes()***

*Retorna la cantidad de cortes agregados.*

**agregarVariables**
Agrega el Pool de nuevas Variables al pool de Variables al problema.

*int* ***agregarVariables()***

*Retorna la cantidad de variables agregadas.*

**getPoolVarAgregadas**
Referencia al pool de Variables agregadas al problema actual.

*Pool<Variable>&* ***getPoolVarAgregadas()***

**getPoolResCortes**
Referencia al pool de Restricciones de Cortes del problema actual.

*Pool<Restriccion>&* ***getPoolResCortes()***

**nuevoCorte**
Este método agrega una Restricción al pool de cortes.

*void* ***nuevoCorte(*** *Restriccion& res* ***)***
- *res:* Referencia al objeto Restricción a agregar al pool de corte.

**nuevaVariable**
Este método agrega una Variable al pool de Variables Agregadas.

*void* ***nuevaVariable(*** *Variable& var* ***)***
- *var:* Referencia al objeto Variable a agregar al pool.

## VARIABLE

### DESCRIPCIÓN
Esta clase representa una variable del problema de optimización. Para inicializar un problema se debe completar un Pool de Variables con objetos de esta clase.

### ATRIBUTOS
Los principales atributos son los siguientes, cada uno tiene sus métodos asociados.
- *identificador:* identifica unívocamente a una variable.
- *tipo:* tipo de la variable (discreta, continua, binaria ).
- *coeficiente*: es el valor de la variable en la función objetivo (real).
- *limite inferior*: es el valor del límite inferior de la variable (real).
- *limite superior*: es el valor del límite superior de la variable (real).

También tiene otros atributos privados de control.

### MÉTODOS PUBLICOS

**Variable**
1º instancia: construye una Variable vacía, con todos los atributos inicializados a nulo.

2º instancia: construye una Variable, inicializando el tipo y el coeficiente en la función objetivo, los límites se inicializan con nulo.
3º instancia: construye una Variable, como copia de la variable recibida como parámetro, pero cambia los límites por los nuevos valores.

***Variable()***
***Variable****( TIPO tipo, double coef)*
- *tipo:* tipo de la variable
- *coef:* coeficiente en la función objetivo.

***Variable****( Variable& var, double limInf, double limSup )*
- *var:* referencia a un objeto variable a copiar.
- *limiteInf:* nuevo valor del límite inferior.
- *limiteSup:* nuevo valor del límite superior.

**getId**
Retorna el identificador único de la variable.

*int* ***getId****()*

**discreta**
Retorna "True" si la variable es discreta. "False" en caso contrario.

*bool* ***discreta****()*

**continua**
Retorna "True" si la variable es continua. "False" en caso contrario.

*bool* ***continua****()*

**setTipo**
Inicializa el tipo de la variable.

*void* ***setTipo****( TIPO tipo)*

**tipo**
Retorna el tipo de la variable.

*TIPO* ***tipo****()*

**setCoef**

Inicializa el valor del coeficiente en la función objetivo.

*void* ***setCoef****( double coef)*

**coef**
Retorna el valor del coeficiente en la función objetivo.

*double* ***coef****()*

**setLimiteInf**
Inicializa el límite inferior de la variable con el valor *l.*

*void* ***setLimiteInf****( double l)*

**limiteInf**
Retorna el límite inferior de la variable.

*double* ***limiteInf****()*

**setLimiteSup**
Inicializa el límite superior de la variable con el valor *l.*

*void* ***setLimiteSup****( double l)*

**limitSup**
Retorna el límite superior de la variable.

*double* ***limiteSup****()*

**esIgual**
Retorna "True" si la variable es igual a la variable referencia por parámetro. El concepto de igualdad se extiende a comparar los atributos identificador, tipo y coeficiente. Retorna "False" en caso contrario.

*bool* ***esIgual****(Variable& var)*
- *var:* referencia a la variable a comparar.

**mostrar**
Retorna una cadena con un determinado formato, mostrando el contenido de la variable.

*char* ****** ***mostrar()***

**getIdxCplex**
Retorna el índice de la variable en el ambiente de la librería Cplex.

*int* ***getIdxCplex()***

**setActiva**
Inicializa la variable como activa, la misma será utilizada para el procesamiento de las relajaciones.

*void* ***setActiva()***

**setDesactiva**
Inicializa la variable como no activa, la misma existe, pero no será utilizada para el procesamiento de las relajaciones.

*void* ***setDesactiva()***

**activa**
Retorna "True" si la variable esta activa, o "False" en caso contrario.

*bool* ***activa()***

## RESTRICCION

### DESCRIPCIÓN
Esta clase representa una restricción del problema de optimización. Para inicializar un problema se debe completar un Pool de Restricciones con objetos de esta clase.

### ATRIBUTOS
Los principales atributos son los siguientes, cada uno tiene sus métodos asociados.

- *identificador:* identifica unívocamente a una Restriccion.
- *sentido:* es el sentido de la desigualdad. Posibles valores:
  "L" = menor o igual.
  "E" = igualdad.
  "G" = mayor o igual.
- *parteDerecha:* valor del lado derecho (real)
- *variables*: arreglo con los identificadores de las variables (array int).

- *coeficientes*: arreglo con los coeficientes de cada variable(array real).
- *cantidad:* cantidad de variables que intervienen.

También tiene otros atributos privados de control

**MÉTODOS PUBLICOS**

**Restricción**

1º instancia: construye una Restriccion vacía, con todos los atributos inicializados a nulo.
2º instancia: construye una Restriccion con los valores de los atributos recibidos como parámetros.
3º instancia: construye una Restriccion como copia de otra, pero con la particularidad que la nueva restricción tiene reemplazada la variable indicada en el parámetro por otra nueva.

***Restriccion****()*
***Restriccion****( int cantVar, int * ids, double * cf, char s, double pd)*

- *cantVar:* cantidad de variables de la restricción.
- *ids:* puntero a arreglo de enteros que contiene los identificadores de las variables.
- *cf:* puntero a arreglo de reales que contiene los coeficientes de las variables.
- *s:* sentido de la desigualdad.
- *pd:* valor del lado derecho de la restricción.

***Restriccion****( Restriccion& res, int idVarVieja, int idVarNueva )*

- *res:* referencia al objeto restricción a copiar.
- *idVarVieja:* identificador de la variable a reemplazar.
- *idVarNueva:* identificador de la nueva variable.

**getId**

Retorna el identificador único de la restricción.

*int* ***getId****()*

**setParteDer**

Inicializa el lado derecho de la restricción con el valor *pd.*

*void* ***setParteDer****( double pd );*

**parteDer**

Retorna el valor del lado derecho de la restricción.

*double* ***parteDer****()*

**setSentido**
Inicializa el sentido de la desigualdad con el valor *s*.

*void* ***setSentido****( char* ***s*** *)*
*donde* ***s*** *es:*
*"L" = menor o igual.*
*"E" = igualdad.*
*"G" = mayor o igual.*

**sentido**
Retorna el sentido de la desigualdad en la restricción.

*char* ***sentido****()*

**setCantVar**
Inicializa la cantidad de variables de la restricción con el valor *c*.

*void* ***setCantVar****( int c )*

**cantVar**
Retorna la cantidad de variables de la restricción.

*int* ***cantVar****()*

**coef**
Retorna el valor del coeficiente en la restricción de la variable recibida por parámetro.

*double* ***coef****( int idxVar )*
- *idxVar:* identificador de la variable.

**esIgual**
Retorna "True" si la restricción es igual a la restricción referenciada por parámetro. El concepto de igualdad se extiende a comparar los atributos sentido, lado derecho y cantidad de variables. Retorna "False" en caso contrario.

*bool* ***esIgual****( Restriccion& res )*
- *res:* referencia a la restricción a comparar.

**existeVar**
Retorna "True" si la variable existe en la restricción. Retorna "False" en caso contrario.

*bool* ***existeVar****( int idVar)*
- *idVar:* identificador de la variable a buscar.

**mostrar**
Retorna una cadena con un determinado formato, mostrando el contenido de la restricción.

*char* * ***mostrar()***

**setActiva**
Inicializa la restricción como activa, la misma será utilizada para el procesamiento de las relajaciones.

*void* ***setActiva()***

**setDesactiva**
Inicializa la restricción como no activa, la misma existe, pero no será utilizada para el procesamiento de las relajaciones.

*void* ***setDesactiva()***

**activa**
Retorna "True" si la restricción esta activa, o "False" en caso contrario.

*bool* ***activa()***

## POOL

### DESCRIPCIÓN
Esta clase representa el almacenamiento para las Variables y Restricciones. Se almacenan de manera independiente, es decir existe un Pool de Variables y un Pool de Restricciones. Se implementa en forma de lista. Es un template pero esta acotado solo a las clases Variable y Restriccion.
Posee los métodos básicos de lista y otros particulares del pool que estamos definiendo.
En las descripciones de los métodos, cuando nos referimos a un *"elemento"* del pool, nos estamos refiriendo o bien a una Variable o bien a una Restriccion.

**MÉTODOS PUBLICOS**

**Pool**
Construye un almacenamiento (pool) vacío de un determinado tipo (Variable o Restriccion ).

*Pool<T>::**Pool()***
Donde *T*, es Variable o Restriccion.

**agregar**
Agrega un elemento al pool, realiza una copia del elemento, es decir no es necesario mantener la referencia del elemento agregado.

*void* ***agregar**(const T& elem)*
- *elem:* referencia al elemento a agregar (Variable o Restriccion).

**eliminar**
Elimina el elemento del pool.

*void* ***eliminar**(const T& elem)*
- *elem:* referencia al elemento a eliminar. (Variable o Restriccion).

**cant**
Retorna la cantidad de elementos del pool.

*int* ***cant()***

**remove**
Elimina todos los elementos del pool. Queda un pool vacío.

*void* ***remove()***

**iterador**
Define un iterador para poder operar con los elementos. Contiene los métodos básicos de cualquier iterador.

*typedef* ***PoolIter***

| | |
|---|---|
| ***ini()*:** | posiciona el iterador en el primer elemento del pool. |
| ***fin()*:** | posiciona el iterador después del ultimo elemento del pool. |
| ***operador*++:** | avanza el iterador. |
| ***operador* *:** | referencia al elemento apuntado por el iterador. |

**operador[]**
Operador para acceder al elemento i-esimo del pool.

*T&* ***operator[ ]****(int indice)*
- *indice:* índice del elemento en el pool.

**get**
Retorna la referencia a un el elemento del Pool identificado por el parámetro.

*T&* ***get****(int identificador)*
- *identificador:* identificador unívoco del elemento (Variable o Restriccion)

**existe**
Retorna un puntero al elemento si el mismo existe en el pool. En caso contrario retorna un puntero nulo.

*T** ***existe****(int identificador)*
- *identificador:* identificador unívoco del elemento (Variable o Restriccion)

**desactivar**
Desactiva todos los elementos del pool.

*void* ***desactivar()***

**mostrar**
Retorna una cadena con un determinado formato, con el contenido del pool, mostrando cada elemento del mismo.

*char ** ***mostrar()***

## REGLAS BRANCH

### DESCRIPCIÓN
Implementa el concepto de reglas de "branch", posee un pool de variables y/o un pool de restricciones, por los cuales se ramificará. También tiene un "flag" que indica si se esta ramificando por variable o por restricción.

### ATRIBUTOS
Los principales atributos son los siguientes:

- porVariable: indica si se bifurca por variable o no.
- porRestriccion: indica si se bifurca por restricción o no.
- *poolVar:* pool que contiene las variables para ramificar.
- *poolRes:* pool que contiene las restricciones para ramificar.
- *infoHijos:* lista de información para cada subproblema hijo a generar.

**MÉTODOS PUBLICOS**

**ReglasBranch**
Construye un objeto ReglasBranch vacío.

***ReglasBranch()***

**put**
1º instancia: agrega la variable al pool de variables de la regla.
2º instancia: agrega la restricción al pool de restricciones de la regla.

*void* ***put****( Restriccion& res)*
- *res:* referencia a la restricción a agregar.

*void* ***put****( Variable& var)*
- *var:* referencia a la variable a agregar.

**cant**
Retorna la cantidad de reglas que contiene el objeto. Es decir si es "por variable", retorna la cantidad de variables; si es "por restricción" retorna la cantidad de restricciones.

*int* ***cant()***

**cantVar**
Retorna la cantidad de variables que existen en el objeto.

*int* ***cantVar()***

**cantRes**
Retorna la cantidad de restricciones que existen en el objeto.

*int* ***cantRes()***

**limpiar**

Limpia todo el objeto, eliminado el contenido de todos sus atributos.

*void* ***limpiar()***

**generarRes**
Cuando se crear una nueva variable que reemplaza a otra ya existente en el problema, se deben crear nuevas restricciones asociadas a la nueva variable y eliminar las referencias de la vieja variable en las restricciones. Este método implementa esta funcionalidad.

*void* ***generarRes****( int idVarVieja, Pool<Restriccion> * poolRes )*
- *idVarVieja:* identificador de la variable a cambiar.
- *poolRes:* puntero al Pool de Restricciones del problema.

## INFO PROBLEMA

### DESCRIPCIÓN
Esta es una clase de datos, y representa la información de un problema de optimización en particular.

### ATRIBUTOS
Los principales atributos son:
- *idProblema*: identificador del problema (int).
- *nombreProblema*: cadena que representa el nombre del problema (char*).
- *estProceso*: estado del proceso de subproblemas (ESTADO_PRC).
- *estOptimizacion*: estado del resultado de la optimización (ESTADO_OPT).
- *idNodo*: identificador del nodo del Arbol de "Branch" (int).
- *infoResVar*: información de las variables y restricciones (InfoResVar).
- *solucionParcial*: estructura de la solución parcial del problema (SOLUCION).
- *cantHijos*: cantidad de subproblemas hijos (int).
- *infoHijos*: arreglo de información de las variables y restricciones de cada hijo (InfoResVar[]).
- *fechaHoraIni*: fecha y hora de inicio de la optimización del problema (char*).
- *fechaHoraFin*: fecha y hora del fin de la optimización del problema (char*).
- *ptrNodoXml:* puntero al nodo del árbol xml, que representa el subproblema. (xmlNodePtr - libxml2)

### MÉTODOS PUBLICOS

**InfoProblema**
Construye un objeto InfoProblema vacío, inicializandos sus atributos.

***InfoProblema()***

## INFO RES VAR

### DESCRIPCIÓN
Esta es una clase de datos, y representa la información de variables y restricciones de un determinado subproblema.

### ATRIBUTOS
Los principales atributos son:
- *cantVar*: cantidad de variables (int).
- *idsVar*: arreglo de tamaño MAX_VAR de identificadores de las variables (int[]).
- *cantRes*: cantidad de restricciones (int).
- *idsRes*: arreglo de tamaño MAX_RES de identificadores de restricciones (int[]).

### MÉTODOS PUBLICOS

**InfoResVar**
Construye un objeto InfoResVar vacío, con los atributos inicializados.

***InfoResVar()***

**copiar**
Copia el contenido del objeto pasado por parámetro.

*void* ***copiar****(InfoResVar * c)*
- *c:* Puntero a objeto InfoResVar a copiar.

## SOLUCION (ESTRUCTURA)

### DESCRIPCIÓN
Estructura de datos que representa la información mínima de una solución de la relajación de un problema. El tamaño de la estructura es estático.

### ATRIBUTOS
Los atributos son:
- *idProceso*: identificador del proceso subproblema que originó la solución(int).
- *fcHora*: fecha y hora de generación.(char*)
- *estado*: estado de solución (int).
- *actualizacion*: "flag" indicando si dicha solución fue actualizada contra la solución global (int).
- *entera*: indica si la solución es entera (true), o no (false) (boolean).
- *obj*: valor objetivo de la solución (double).
- *limiteInf*: límite inferior de la solución (double).
- *limiteSup*: límite superior de la solución (double).

- *cantVar*: cantidad de variables(int).
- *x*: arreglo de tamaño MAX_VAR de los valores de cada variable (double[]).

## SOLUCION (CLASE)

### DESCRIPCIÓN

Esta clase representa la solución de la relajación de un problema. El tamaño de los atributos de la clase es dinámico.

### ATRIBUTOS

Los principales atributos son:

- *estado*: estado de la solución (int).
- *obj*: valor de la función objetivo (double).
- *x*: arreglo con los valores de cada variable (double[]).
- *pi*: arreglo con los valores de variables dual para cada restricción (double[]).
- *slack*: arreglo con los valores de las variables slack de cada restricción (double[]).
- *dj*: arreglo el costo reducido de cada variable (double[]).

### MÉTODOS PÚBLICOS

**Solucion**

Construye un objeto Solucion inicializando los atributos.

***Solucion****( int cantV, int cantR)*

- *cantV:* cantidad de variables.
- *cantR:* cantidad de restricciones.

**esEntera**

Retorna "True" si la solución es una solución entera. "False" en caso contrario.

*bool* ***esEntera****()*

**valorObj**

Retorna el valor objetivo de la solución.

*double* ***valorObj****()*

**cantVar**

Retorna la cantidad de variables de la solución.

*int* ***cantVar()***

**cantRes**
Retorna la cantidad de restricciones de la solución.

*int* ***cantRes()***

**idxVarNoEnteras**
Retorna en el parámetro por referencia un arreglo con los identificadores de todas las variables no enteras de la solución.

*int* ***idxVarNoEnteras****( int * idsVar )*
- *idsVar:* puntero a arreglos de identificadores de variables.

**mostrar**
Retorna una cadena con un determinado formato, mostrando el contenido de la solución.

*char * **mostrar()***

## ADMINFO

### DESCRIPCIÓN
Implementa las funcionalidades del manejo de memoria para los datos de los problemas de optimización. Administra también la memoria compartida entre el objeto Maestro y los objetos Subproblemas.
Cuando en esta clase nos referimos a "Info" se refiere a la información de un subproblema.
Esta clase es configurable en cuanto a la cantidad de memoria que se quiere procesar en ram (el resto lo trabaja en disco). También se configura la cantidad de memoria compartida a utilizar.

### MÉTODOS PRINCIPALES

**AdmInfo**
Crea un objeto para administrar los datos de los subproblemas, este objeto es creado desde el objeto Maestro. Sus parámetros son básicamente de configuración, y determinan la performance de su procesamiento.

***AdmInfo****( int sizeInfoL, int sizeInfoMemC, int porcInfoLocalV,*
*int cantSubpE, char * pathM, int maxInfoMemComp )*

- *sizeInfoL:* cantidad máxima de Info en Memoria Local. Superando este valor, continua procesando, copiando a disco las restantes Infos.
- *sizeInfoMemC:* cantidad de Info en Memoria Compartida.
- *porcInfoLocalV:* porcentaje mínimo a mantener en memoria local.
- *cantSubpE:* cantidad de procesos SubProcesoLp a ejecutar.
- *pathM:* path donde se mantiene los archivos de mapeo a memoria compartida.
- *maxInfoMemComp:* máxima cantidad de Info en Memoria Compartida.

## NODO

### DESCRIPCIÓN

Esta clase implementa un nodo del árbol de branch and cut.
El contenido del nodo es una referencia a la información de un subproblema.

### MÉTODOS PRINCIPALES

**Nodo**
1º instancia: crea un nodo vacío, inicializando sus atributos.
2º instancia: crea un nodo vacío, con un determinado identificador y una determinada referencia.

***Nodo()***
***Nodo****( int idNodo, void * info )*
- *idNodo:* identificador del nodo a crear.
- *info:* puntero a la información a guardar en el nodo.

**esVacio**
Retorna "True" si el nodo esta vacío, es decir inicializado con los valores por defecto. "False" en caso contrario.

*bool* ***esVacio()***

**cantHijos**
Retorna la cantidad de nodos hijos.

*int* ***cantHijos()***

**setPadre**

Inicializa el identificador del padre en el nodo actual.

*int* ***setPadre****(int idNodoPadre)*
- *idNodoPadre:* identificador del nodo padre.

**getPadre**
Retorna el identificador del nodo padre.

*int* ***getPadre****()*

**setHijo**
Inicializa el identificadores de un hijo del nodo.

*void* ***setHijo****(int idHijo)*
- *idHijo:* identificador del nodo hijo a copiar

**getHijo**
Retorna el identificador del i-esimo nodo hijo.

*int* ***getHijo****(int i)*
- *i:* índice del hijo en el nodo.

**setNivel**
Inicializa el nivel de nodo en el árbol.

*void* ***setNivel****(int n)*
- *n:* nivel en el árbol

**getNivel**
Retorna el nivel del nodo en el árbol

*int* ***getNivel****()*

**setInfo**
Inicializa el contenido del nodo.

*void* ***setInfo****( void * nuevaInfo )*
- *nuevaInfo:* puntero a la información a guardar en el nodo.

**getInfo**
Retorna un puntero a la información contenida en el nodo

*REF * **getInfo()***

**mostrar**
Retorna una cadena con un determinado formato, mostrando el contenido del nodo.

*char * **mostrar()***

## ARBOLBQ

### DESCRIPCIÓN

Esta clase implementa un árbol como estructura de datos, y además tiene las funcionalidades de búsqueda en el mismo. Contiene una lista de nodos (de la clase *Nodo*) y otros atributos privados para implementar las búsquedas.

### MÉTODOS PRINCIPALES

**ArbolBq**
Crea un árbol vacío, inicializando sus atributos.

***ArbolBq()***

**cantNodos**
Retorna la cantidad de nodos que contiene el árbol.

*int* ***cantNodos()***

**altura**
Retorna la altura del árbol.

*int* ***altura()***

**agregar**
Agrega un nuevo nodo al árbol, lo agrega como nodo hijo del padre pasado como referencia.

*void* ***agregar****( int idNodoPadre, Nodo& nodo)*
- *idNodoPadre*: identificador del nodo padre.

- *Nodo*: referencia al nodo a agregar.

**getNodo**
Retorna una referencia al nodo con identificador *id.*

*Nodo&* ***getNodo****( int id)*
- *id:* identificador de nodo a consultar.

**getRaiz**
Retorna el nodo raíz.

*Nodo&* ***getRaiz****()*

**setModo**
Inicializa el modo de búsqueda en el árbol. Por defecto al crear el árbol se inicializa BFS.

*void* ***setModo****( MODO_BUSQUEDA m )*
- *m:* Modo de búsqueda, valores: BFS, DFS

**getTipoBusqueda**
Retorna el modo de búsqueda en el árbol.

*MODO_BUSQUEDA* ***getTipoBusqueda****()*

**haySinVisitar**
Retorna "True" si todavía quedan nodos sin visitar. "False" en caso contrario.

*bool* ***haySinVisitar****()*

**proxNodo**
Retorna un puntero al próximo nodo que corresponde relacionado con la búsqueda actual.

*Nodo* ****** ***proxNodo****()*

**mostrar**
Retorna una cadena con un determinado formato, mostrando el contenido del árbol.

*char* ****** ***mostrar****()*

## EXCEPCION

### DESCRIPCIÓN

Esta clase maneja las excepciones generadas por algún método de la librería. Contiene dos atributos, una categoría y una descripción de la excepción.

### MÉTODOS PRINCIPALES

**Excepcion**
Crea una excepción con determinados valores.

***Excepcion****(const int& cat, const char* cadena,...)*

- *cat:* categoría de la excepción.
- *cadena:* mensaje descriptivo del error generado.

**mostrarError**
Retorna una cadena con el mensaje de la excepción.

*char * **mostrarError**(void) const*

**mostrarCategoriaError**
Retorna la categoría de la excepción.

*int **mostrarCategoriaError**(void) const*

## CONSTANTES

### DESCRIPCIÓN

La librería posee algunas constantes predefinidas.

| | | |
|---|---|---|
| MAX_VAR | 1536 | // Cantidad máxima de variables posibles |
| MAX_RES | 1024 | // Cantidad máxima de restricciones posibles |
| MAX_CANT_HIJOS | 512 | // Cantidad máxima de subproblemas hijos |
| SOLUCION_OK | 0 | // Resultado de la solución de la optimización |
| SOLUCION_NULA | -1 | // Sin resultado de la solución de la optimización |

## ENUMERADOS

### DESCRIPCIÓN

La librería posee algunos enumerados predefinidos.

Sentido de la optimización de la función objetivo:
SENTIDO {MAXIMIZAR, MINIMIZAR }

Modos de búsqueda en el árbol:
MODO_BUSQUEDA {BFS, DFS, NULO }

Estados de los procesos SubProcesoLp:
ESTADO_PRC {ACTIVO, TERMINADO,
ABIERTO, ESTADO_PROC_NULO}

Estado del resultado de la optimización.
ESTADO_OPT {OPT_OK, NO_OPT, NO_LIM, OPT_NULO }

Estado de la ultima ejecución (archivo de inicialización).
MODO_INI_EXE { MODO_ALIVE=1, MODO_NORMAL=2,
MODO_STOP=3, MODO_CONTINUE=4 }

# REDEFINICION DE METODOS HEREDADOS

Cada método virtual tiene ya una implementación, que se puede dejar o modificarla redefiniendo el mismo. (ver **EJEMPLO**)

**CLASE MAESTRO**

Solo se pueden redefinir los métodos output de cada solución parcial (*outputParcial*) y global para modificar la salida de la solución. (*output*).

**CLASE SUBPROBLEMA**

Se puede redefinir los siguientes métodos:

- *separar()*: realizar los planos de cortes.
- *generarReglasBranch()*: cargar las reglas de ramificación adecuadas.
- *generarHijos()*: crear los nuevos subproblemas a partir de las reglas generadas.
- *preOptimizacionLp()*: mostrar el problema previo a la relajación.
- *output()*: mostrar los resultados de la relajación.

# PROCESOS

Se definen los procesos que existen en el ambiente de la librería. Cada uno de ellos se ejecuta como un proceso en sí en el Sistema Operativo.

## PROCESO USUARIO

Este será el proceso definido por el usuario que utiliza la librería para optimizar un problema. Desde este proceso se deberá construir una instancia de la clase Maestro, y luego invocando los métodos que correspondan se obtendrá la solución del problema.
Deberá construir en forma local los Pools de variables y restricciones para así poder inicializar el problema.
Como ejemplo se muestra el mínimo código que deberá contener.

```
int main(int argc, char** argv)
{
	try
	{
		//
		// Declara los Pools
		//
		Pool<Variable>		poolVariable;
		Pool<Restriccion>	poolRestriccion;

		//
		// Carga de datos del problema real, para mapearlos a los  pools de Variables y Restricciones
		//
		cargarProblema(poolVariable, poolRestriccion);

		//
		// Crea objeto MaestroUsr
		//
		MaestroUsr * m = new MaestroUsr("Test", false, false, MINIMIZAR);

		//
		// Carga las Variables y Restricciones en la libreria.
		//
		m->inicializarPool(poolVariable, poolRestriccion);

		//
		// Optimiza el problema global.
		//
		m->optimizar();

		//
		// Muestra la Solucion
		//
		m->output();

		delete(m);
	}
	catch(Excepcion& e)
	{
		cout << e.mostrarError()<< endl;
		return -1;
	}
	catch(...)
	{
		return -1;
	}
}
```

## PROCESO ADMINISTRADOR DE POOL

Tiene la funcionalidad de administrar todas la variables y restricciones del problema. Contiene en memoria local todas las variables y restricciones, y responde a los pedidos de los procesos SubProcesoLp.
Tiene el control de la cantidad de subproblemas que están utilizando cada variable/restricción, y con esto, mantiene la depuración de los pools.
Los pedidos pueden ser de solicitar o liberar variables/restricciones.
Cada instancia de los procesos SubProcesoLp cuando comienza a procesar un nuevo problema, solicita al Administrador de Pool las variables y restricciones necesarias.

## PROCESO ADMINISTRADOR DE PROCESOS SUBPROCESOLP

Tiene la funcionalidad de levantar instancias en forma paralela de procesos SubProcesoLp y mantener el control de los mismos.
Asigna a cada instancia de los SubProcesoLp los inputs que le corresponde en memoria compartida.
La cantidad de instancias de SubProcesoLp a levantar y la cantidad de inputs de cada uno, están parametrizados en el archivo de configuración de la librería.
(ver **CONFIGURACION**)

## PROCESO SUBPROCESOLP

Es el proceso que realiza optimizaciones de los problemas. Existen varias instancias de este mismo proceso, son levantadas por el Administrador de SubProcesoLp.

El proceso tiene asignado varios inputs (estructuras en memoria compartida), cada input contiene la información necesaria para optimizar un problema. Luego el proceso toma el input que contenga información y procesa un problema, una vez terminado este, sigue con el próximo input procesando otro problema, y así cicla entre todos los inputs que tenga definidos.
Un objeto Maestro, definido en el proceso del usuario es el encargado de completar dichos inputs. La cantidad de inputs se define en el archivo de configuración.

Para optimizar un problema utiliza un objeto instancia de SubProblemaUsr, que es la clase definida por el usuario que hereda de la clase Subproblema provista por la librería.

## PROCESO INTERFAZ DE USUARIO

Este proceso es independiente de la librería y de cada proceso de la misma, es una interfaz para el usuario que implementa un menú para facilitar el uso de la librería.

## COMUNICACIÓN ENTRE PROCESOS

Cada proceso tiene definida una comunicación. Esta comunicación entre procesos esta implementada con IPC. Cada proceso al levantar, se crea su propia cola en Entrada y recibe como parámetro su cola de Salida.

Existe una particularidad para los procesos SubProcesoLp, que además de tener sus dos colas de control (entrada/salida), tiene una tercer cola exclusiva para la comunicación con el AdmPool, (para la transmisión de variables y restricciones). De esta manera el proceso AdmPool, tendrá como colas de entradas de pedidos, una por cada instancia de SubProcesoLp, evitando de esta manera congestionamiento de pedidos.

La configuración de las colas entre cada proceso, se puede observar en la figura, y esta definida en la instanciación de cada uno. Es decir cuando se levanta cada proceso se le informa cual será su cola de Entrada. La configuración es la siguiente:

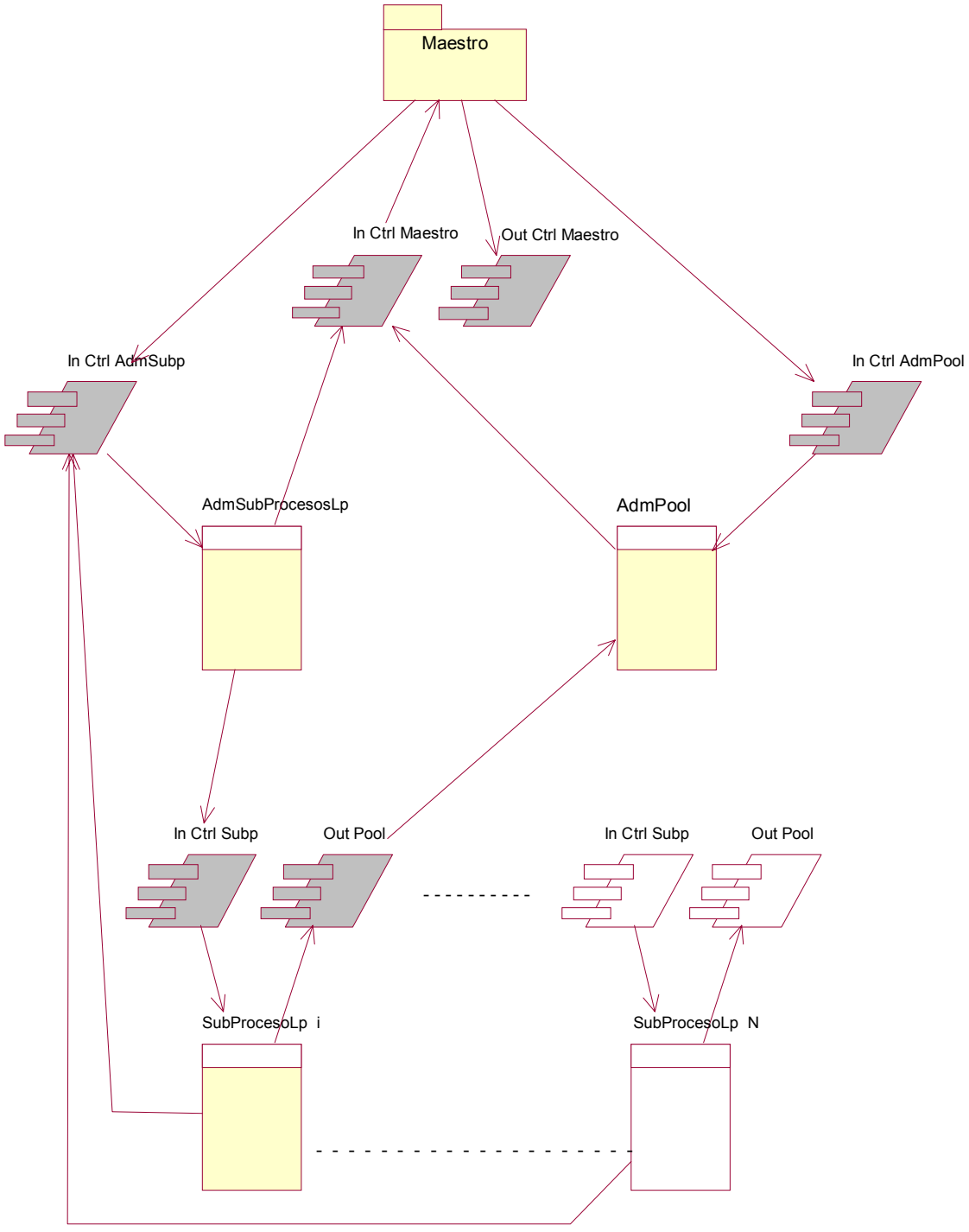
- Maestro: levanta los procesos AdmPool y AdmSubp, informándoles que la cola de salida de ellos es la de entrada del Maestro (***In Ctrl. Maestro***).

- AdmSubProcesoLp: levanta las instancias de SubProcesoLp, informándoles que la cola de salida de control será la cola de entrada del Administrador (***In Ctrl. AdmSubp***).

- AdmPool: por definición para atender los pedidos de los procesos SubProcesoLp, lee entre todas las colas de salida de los SubProcesoLp. En las colas de salida especificas para pedidos (***OutPool***), no de control.

- SubProcesoLp: al inicializar el proceso envía un mensaje de control al AdmSubp informando cual es su cola de entrada (***In Ctrl. Subp***), no de control.

## ESTRUCTURA DE COLAS (FIG.4)



# CONFIGURACION

La librería utiliza un archivo de configuración con algunos parámetros. Este archivo es leído una única vez al inicio de la ejecución de la librería.
Es importante no modificar el formato de este archivo, se debe utilizar exactamente el mismo formato previsto por la librería, solo se pueden modificar los valores de los parámetros.

## PARÁMETROS

| **Parámetro** | **Descripción** |
|---|---|
| *username* | Nombre de usuario |
| *nivellog* | Nivel máximo de log, se escriben todos los mensajes con nivel de log menor o igual al valor de este parámetro. |
| *pathbin* | Path absoluto del directorio donde se encuentran los procesos. Por ejemplo: /home/usrname/test/bin |
| *pathlog* | Path absoluto donde se crearan los archivos de log de los procesos.<br>Por ejemplo: /home/usrname/test/log |
| *cantsubpeje* | Cantidad de instancias de procesos SubProcesoLp. |
| *cantinfoxsubp* | Cantidad de inputs por cada instancia de SubProcesoLp. |
| *metodoopt* | Método de optimización utilizado por CPLEX.<br>Valores posibles: p: primopt(), d: dualopt(), h: hybnetopt(), i: mipopt(), b:baropt(), y: hybbaropt() |
| *maxiteracioneslp* | Cantidad máxima de iteraciones durante la optimización. |
| *maxcantnodos* | Cantidad máxima de nodos del árbol. Si se llega a esta cantidad y no se encontró una solución entera, se detiene la búsqueda y se retorna la mejor solución parcial hasta el momento. |
| *timeoutctrl* | Cantidad de segundos del intervalo de control de los procesos SubProcesoLp. Este parámetro es utilizado por el Administrador de SubProcesoLp. |
| *timeoutstart* | Cantidad de segundos de espera desde la orden de Comienzo de Procesamiento (desde el menú), hasta que levanta por completa la aplicación, superado dicho tiempo, la aplicación igualmente continuará levantando, pero en el menú se informara dicho timeout. |
| *maxinfolocal* | Cantidad máxima de estructuras de Información para un SubProcesoLp que se mantendrá en memoria RAM. Supero este valor se copiaran a disco las restantes y se levantaran a memoria en el instante de su proceso. Uso por el Administrado de Información (AdmInfo). |
| *porcinfolocal* | Porcentaje mínimo que se desea mantener en memoria RAM, cuando se encuentra por debajo de este valor, y existen estructuras en disco, las mismas se levantaran a memoria RAM.<br>Uso por el Administrado de Información (AdmInfo). |
| *maxinfomemcomp* | Cantidad máxima de estructuras de Información permitidas copiar en memoria compartida. Uso por el Administrado de Información (AdmInfo). |
| *refresharbol* | Cantidad de segundos que representa el intervalo de tiempo en que se actualiza el archivo xml conteniendo el Arbol Branch and Cut. |
| *reintentosmsg* | Cantidad máxima de reintentos para leer o escribir en las colas IPC, utilizas para control entre los procesos de la librería. |
| *timeCkp* | Cantidad segundos que representan el intervalo para realizar los checkpoint de la información procesada.<br>Si el valor es 0 o indefinido, no se realizara ningún checkpoint. |

# LOG

Cada proceso tiene su propio archivo de log, y escribe los mensajes según el nivel de log que corresponda. Generan los archivos en el directorio de log configurado (archivo de configuración).
El formato de los nombres de archivos es el siguiente:
***<Nombre del Proceso>.<Id Interno del Proceso>.<AAMMDD_HHMISS_99999>.log***

Cada mensaje tiene asociado un nivel de log, luego cuando un proceso esta corriendo, existe un nivel de log configurado (leído desde el archivo de configuración). Entonces se escribirán en el log los mensajes que tengan un nivel menor o igual al nivel de configuración.
El nivel de los mensajes es creciente en cuanto al nivel de detalles, todos los mensajes de error tiene nivel 0, por lo que independientemente del nivel de detalles que se desee, se escriben siempre.

**NIVELES DE LOG POR TIPOS DE MENSAJES**

| *Nivel* | *Tipos de mensajes* |
|---|---|
| 0 | Mensajes de Error |
| 1 | No utilizado |
| 2 | Tipos de mensajes principales enviados/recibidos entre los procesos.<br>Información de colas IPC de cada proceso. |
| 3 | Tipos de mensajes secundarios enviados/recibidos entre los procesos. |
| 4 | Detalles del contenido de los datos cargados en la librería Cplex |
| 5 | Contenidos de cada Variable y Restricción que contiene el subproblema. |
| 6 | Aplanado/desaplanado de los mensajes entre los procesos. |

# INTERFAZ DE USUARIO

La librería contiene una interfaz de usuario, mediante la cual se tiene el control de una instancia de procesamiento de la librería PBC.

Los parámetros de configuración para esta interfaz, están definidos como en el mismo archivo de configuración de la librería (libpbc.cfg). También toma configuración de instancias de procesamiento, estos últimos datos son leídos desde el archivo de inicialización de la librería (libpbc.ini), que contiene la información de la ultima instancia procesada.

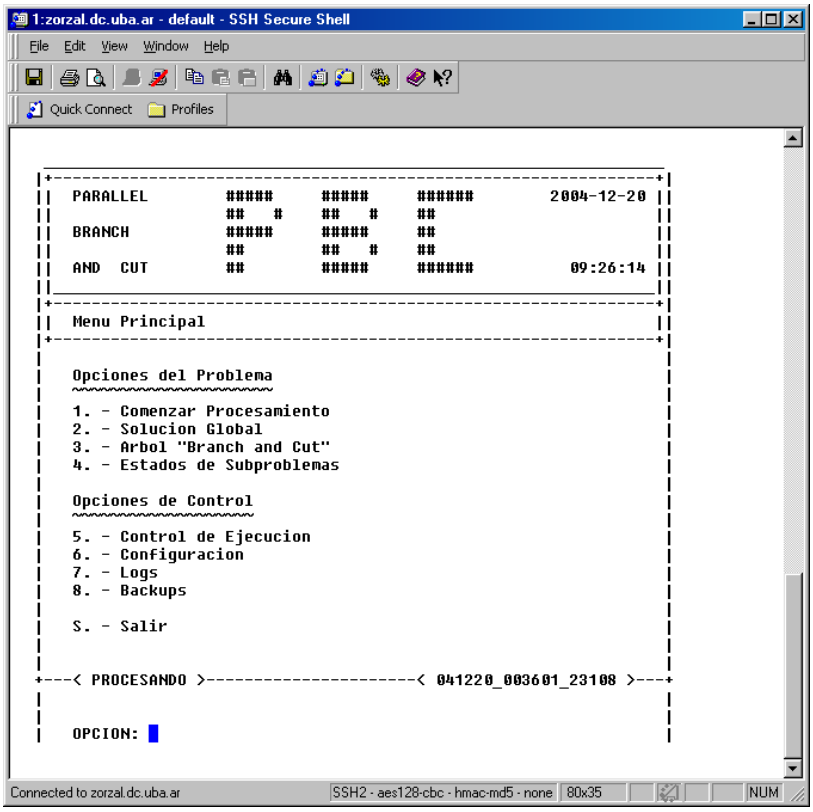
La funcionalidad de la interfaz se puede dividir en dos partes:

**Funcionalidad del Problema a Resolver**

Contiene opciones para resolver el problema, se puede consultar y visualizar las soluciones parciales y la solución global. También se puede visualizar el árbol de Branch and Cut en detalle.

**Funcionalidad de Control**

Contiene opciones de control de procesamiento contra el proceso de usuario e indirectamente contra los procesos de la librería, también se administran los parámetros de configuración. Para la comunicación entre este proceso "Interfaz de Usuario" y los procesos de la librería, se utiliza la mensajería IPC.



## OPCIONES DEL PROBLEMA

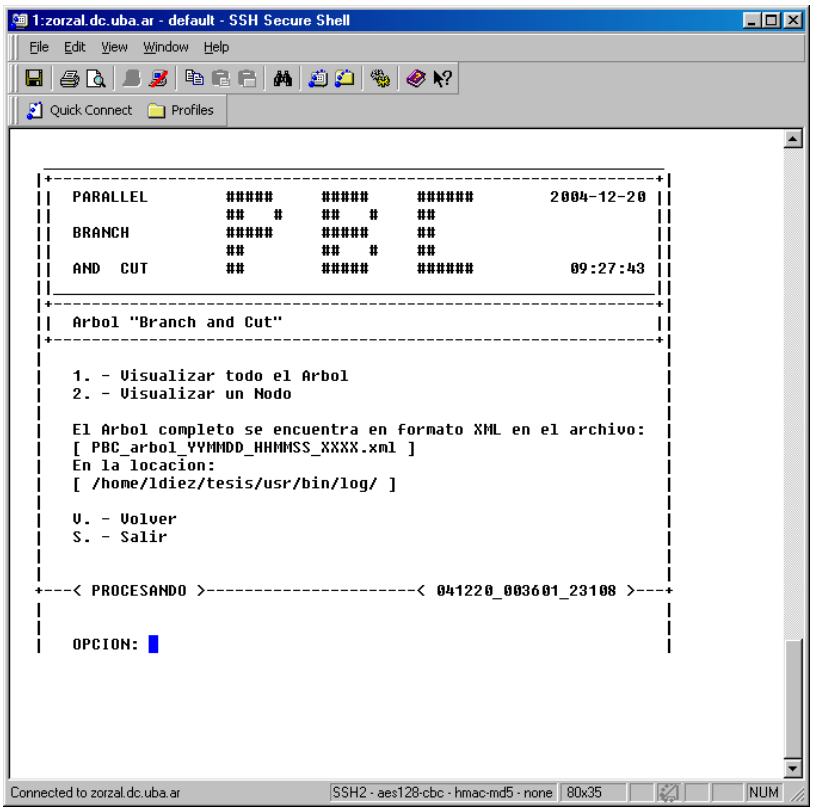
### COMIENZO DE PROCESAMIENTO

Ejecuta la aplicación según los datos de inicialización (libpbc.ini), y de esta manera se comienza a procesar para resolver el problema.

### SOLUCIÓN GLOBAL

Visualiza la mejor solución obtenida hasta el momento. Se muestra el nodo y el subproblema donde se obtuvo la mejor solución, se detallan el valor objetivo y los valores enteros de cada variable.

### ARBOL DE BRANCH AND CUT




Mientras se procesa la optimización del problema, se va construyendo el Arbol de Branch and Cut, cada nodo representa un subproblema que contiene los datos del mismo, y la solución parcial del nodo o vacío si esta sin procesar.
Además de la información del Arbol que se puede visualizar en esta opción, la librería genera un archivo en formato XML, que contiene el árbol completo. El archivo se genera en el directorio de logs con el nombre: *PBC_arbol_AAMMDD_HHMMSS_9999.xml*

Este archivo se puede exportar a algún browser para una mejor visualización.
El formato del mismo se detalla a continuación:

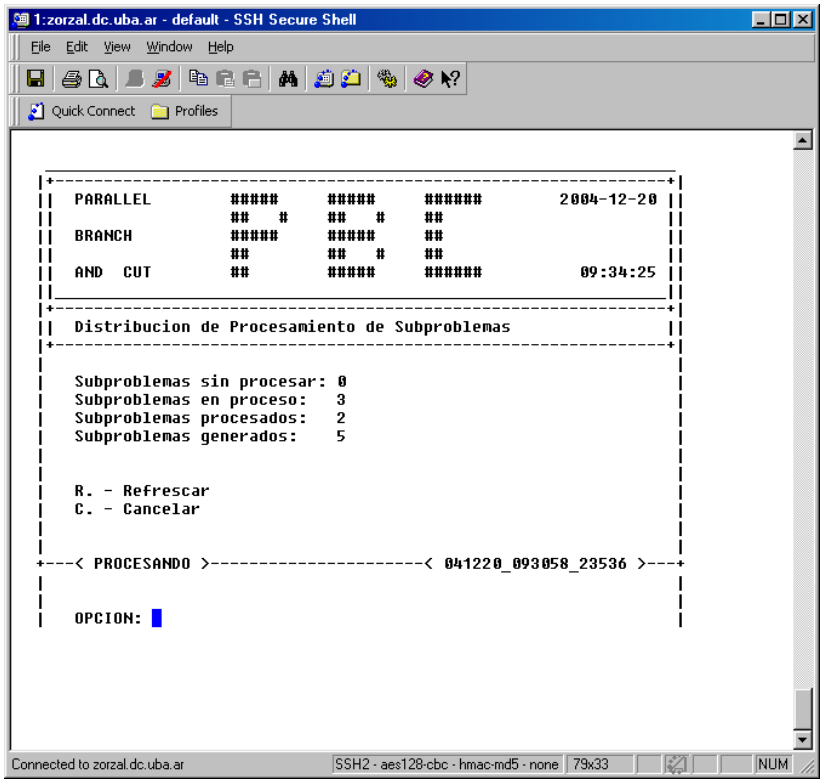
```
<ARBOL cantnodos=valor altura=valor >
        <PROBLEMA nombre=valor timeini=valor timefin=valor />
        <MEJOR_SOLUCION>
                <NODO>
                </NODO>
                …
        </MEJOR_SOLUCION>
<NODO id=valor canthijos=valor nivel=valor idpadre=valor>
        <SUBPROBLEMA nombre=valor estado=valor timeini=valor timefin=valor />
        <SOLUCION estado=valor obj=valor liminf=valor limsup=valor >
                <VAR nombre=valor coef=valor />
                <VAR nombre=valor coef=valor />
                …
        </SOLUCION>
        <NODO>
        </NODO>
        …
</NODO>
</ARBOL>
```

**Visualizar todo el árbol**
Visualiza el árbol completo con todos los nodos definidos hasta el momento. Cada nodo posee un id identificatorio y toda la información del subproblema que representa.

**Visualizar un nodo**
Visualiza un nodo en particular identificado por un id que se pide como parámetro.

### ESTADOS DE SUBPROBLEMAS

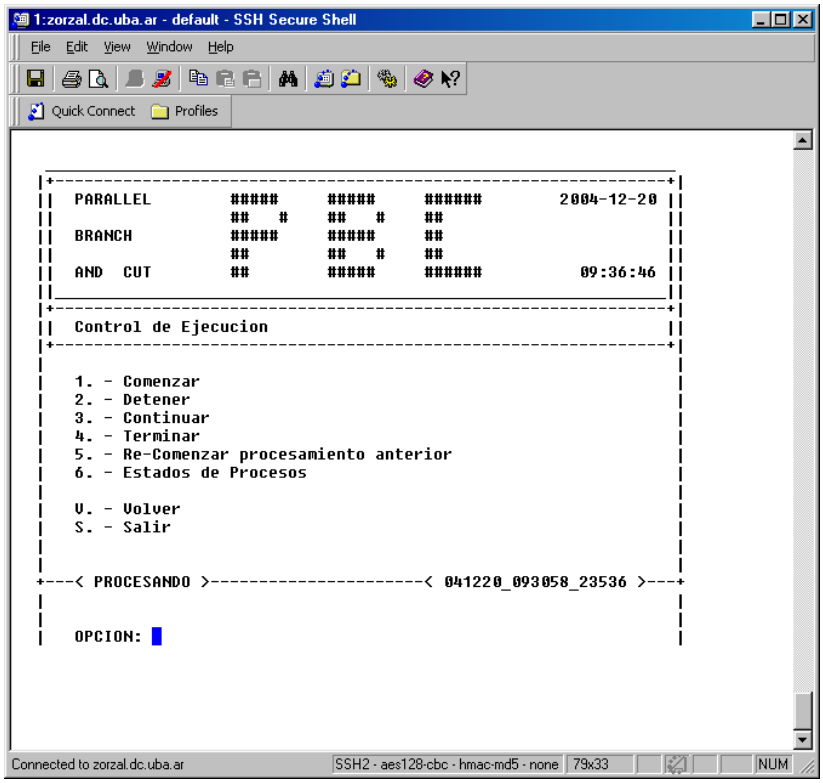
Realiza una consulta a la aplicación mediante los procesos de la librería, y de esta manera recupera la cantidad de subproblemas existentes, agrupados por los diferentes estados. Esta opción realiza una consulta online, solo es posible aplicarla si la aplicación esta corriendo.



## OPCIONES DE CONTROL

### CONTROL DE EJECUCIÓN

Este sub-menu, contiene opciones de control que se ejecutan directamente sobre el objeto Maestro.




**Comenzar**

Permite el lanzamiento del proceso del usuario. De esta manera se inicia el procesamiento para la resolución del problema. Esta opción tiene en cuenta cual fue el procesamiento anterior, informando si existe alguna inconsistencia con la corrida anterior, o si la instancia anterior termino de manera inesperada y se quiere re-procesar a partir del último checkpoint realizado.

**Detener**

Realiza una detención del procesamiento, deteniendo la ejecución de manera ordenada y resguardando los resultados obtenidos hasta el momento, para luego poder aplicar la opción *Continuar*.

De esta manera se permite realizar procesamientos de grandes volúmenes de información en procesamientos parciales administrados por el usuario.

**Continuar**
Es aplicable solo si previamente se ha aplicado la opción *Detener*. Esta opción permite continuar con la resolución del problema, a partir de la información ya procesada antes de la ejecución de la opción *Detener*.

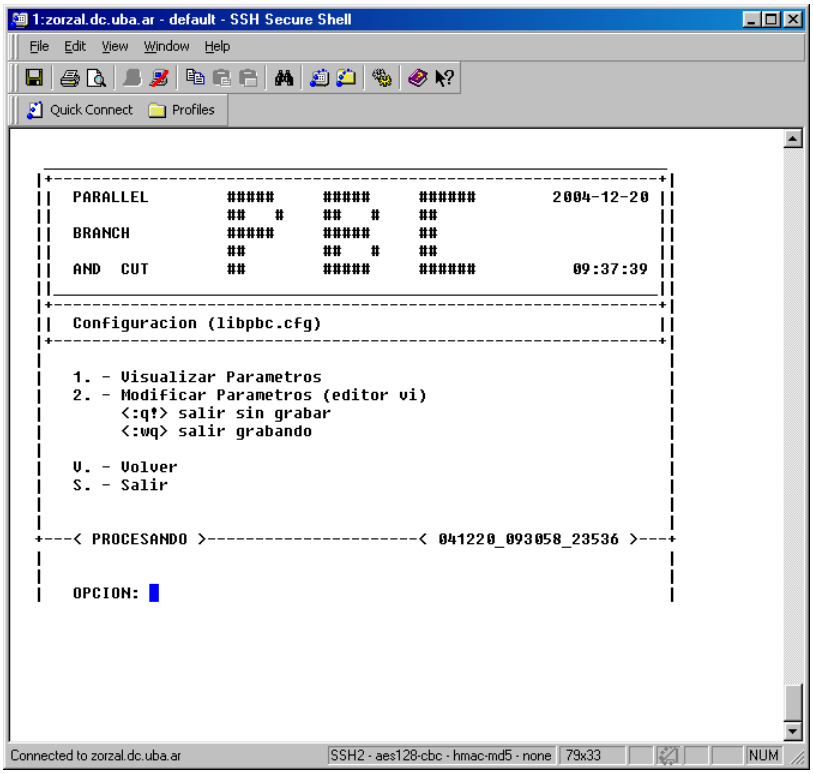
**Terminar**
Realiza la bajada controlada de los procesos de la librería. Se debe tener en cuenta que si se detiene el procesamiento con esta opción, no se resguarda la información procesada hasta el momento. Envía una orden de control a la librería para interrumpir a ejecución del método optimizar() del objeto Maestro.

**Re-Comenzar**
Esta opción permite re-procesar un problema procesado parcialmente. Es decir, se aplica si existe una instancia anterior en la que se ha ejecutado la opción *Detener*, y luego no se desea continuar con el procesamiento, sino iniciarlo nuevamente.

**Estados de Procesos**
Muestra el estado de todos los procesos de la librería en el sistema operativo.

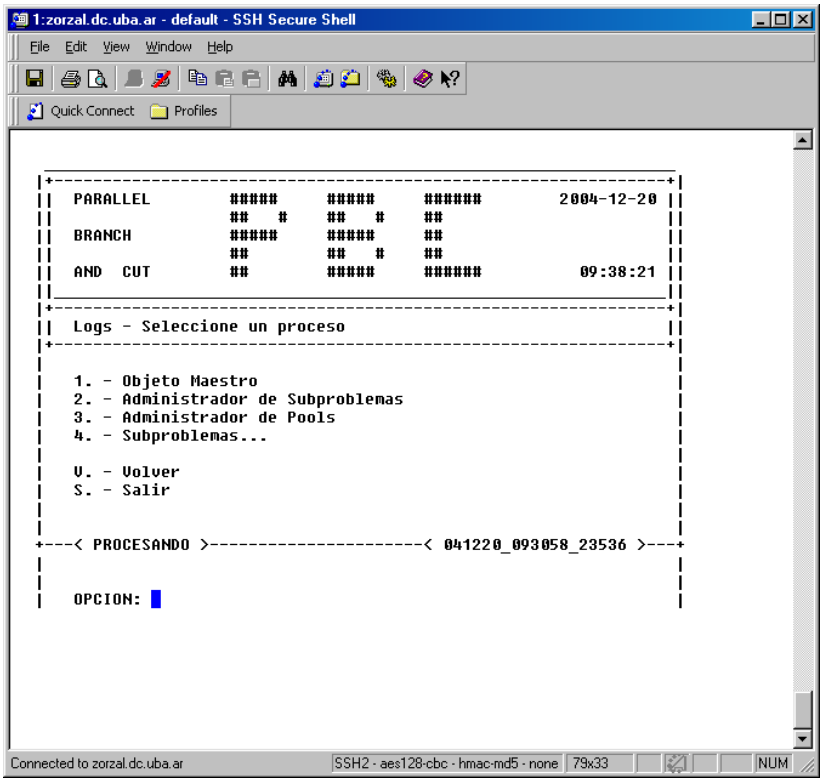
## CONFIGURACIÓN

Permite visualizar y modificar los parámetros de configuración de la librería. Cualquier cambio de valores en los parámetros toman efectos en la ejecución a partir del próximo ejecución de la librería.



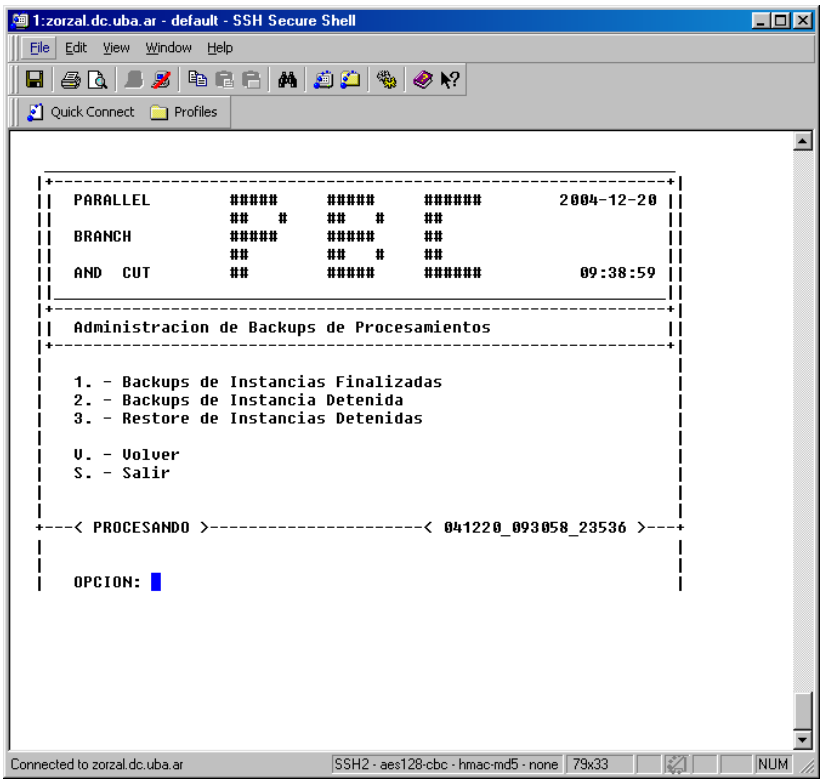
## LOGS

Permite visualizar los logs de los procesos de la librería, seleccionando por proceso.



## BACKUPS

Esta sub-menu permite realizar resguardos de información procesada, los archivos generados (.tar.gz) son copiados en el directorio <*dirtrabajo*>/*bin*/***backups***.
Existen tres opciones:




### Backups de Instancias Finalizadas

Genera un backup del archivo (en formato xml) que contiene el árbol Branch and Cut con los resultados de cada subproblema y la solución global (si existe).

### Backups de Instancia Detenidas

Genera un backup con toda la información procesada de la instancia a la que se le ha aplicado previamente la opción *Detener*. Esta información resguardada podrá ser utilizada en algún momento posterior para continuar con el procesamiento del problema.

### Restore de Instancias Detenida

Recupera algún backup previamente generado, luego de dicha recuperación es posible aplicar la opción *Continuar* a la instancia recuperada.

## ARCHIVO DE INICIALIZACIÓN

Existe un archivo de inicialización que contiene información de la ultima instancia de la librería, puede estar ejecutándose o no. La información de este archivo es utilizada por la interfaz de usuario como parámetros de configuración complementarios.

### PARÁMETROS

| **Parámetro** | **Descripción** |
|---|---|
| *procesoUsr* | Nombre del proceso del usuario. |
| *argsUsr* | Argumentos del proceso de usuario. |
| *modoIni* | Modo de ejecución de la instancia actual o ultima instancia procesada.<br>( ALIVE=1, NORMAL=2, CONTINUE=3, RE_CONTINUE=4 ) |
| idColaEntMaestro | Identificador de la cola de entrada del proceso Maestro (uso exclusivo de proceso) |
| idColaSalidaMaestro | Identificador de la cola de salida del proceso Maestro (uso exclusivo de proceso) |
| etiqueta | Etiqueta identificatoria de la ultima ejecución de la librería. |
| problema | Nombre del Problema a resolver. |
| archivoArbol | Nombre del archivo XML que contiene el árbol Branch & Cut |
| pathMap | Nombre del directorio de mapeo de información de la instancia. |

# UTILITARIOS AUXILIARES DE CONTROL

Existen algunos script para control de procesos, que se pueden utilizar, independientemente de la interfaz de usuario.

**comenzar.sh**

Simplemente inicializa la variable de entorno solicitada por Cplex (CPLEXLICDIR), y ejecuta el programa generado por el usuario.

**terminar.sh**

Es utilizado solo en el caso de una bajada no controlada de los procesos. El script libera los recursos que quedaron asignados (recursos ipcs y archivos de memoria compartida).
Si los procesos terminan en forma controlada, no es necesario ejecutar este script.

**ps-pbc.sh:**

Este script retorna por pantalla la información de los procesos corriendo.

# EJEMPLO

Se trata de un ejemplo de uso de la librería. Contiene los archivos con los códigos necesarios para realizar una prueba. Para compilarlos, seguir los pasos ya explicados en configuración del Makefile.
Los detalles del ejemplo y datos de ejecuciones ver documento ***PBC_Teoria_v22.pdf***