71.14 Modelos y optimización I

Trabajo Práctico

Grupo 5

Datos Personales	
Padrón:	95795
Apellido, Nombre:	Aguilera, Santiago
Turno:	Martes Noche
Ayudantes:	Levy, Jonathan - Cavallero, Alejandro

Evaluación de las entregas del Trabajo Práctico			
	Fecha de Aprobación	Firma	Observaciones
1ª Entrega			
2ª Entrega			

Evaluación Final del TP			
	Fecha de Aprobación	Firma	Observaciones
Escrito			
Carpeta			

Nota final del Trabajo Práctico	
Nota	
Firma	

Datos Personales	
Padrón:	95929
Apellido, Nombre:	Dubiansky, Nicolás
Turno:	Martes Noche
Ayudantes:	Levy, Jonathan - Cavallero, Alejandro

Evaluación de las entregas del Trabajo Práctico			
	Fecha de Aprobación	Firma	Observaciones
1ª Entrega			
2ª Entrega			

Evaluación Final del TP			
	Fecha de Aprobación	Firma	Observaciones
Escrito			
Carpeta			

Nota final del Trabajo Práctico	
Nota	
Firma	

Datos Personales	
Padrón:	96453
Apellido, Nombre:	Ripari, Sebastián
Turno:	Martes Noche
Ayudantes:	Levy, Jonathan - Cavallero, Alejandro

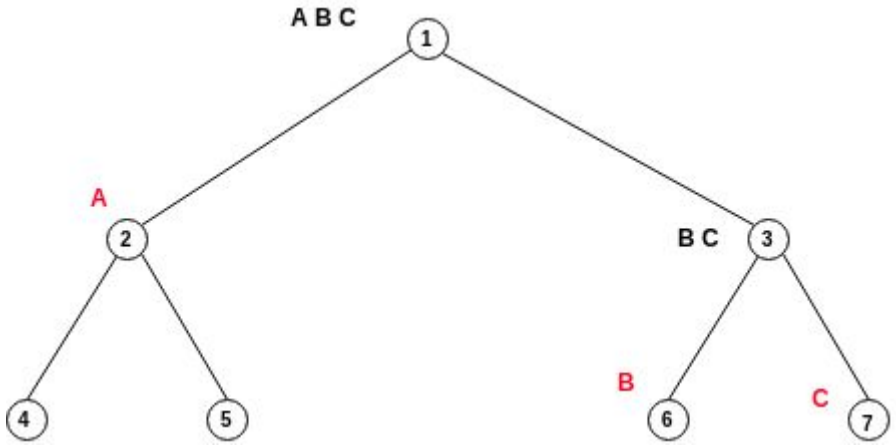
Evaluación de las entregas del Trabajo Práctico			
	Fecha de Aprobación	Firma	Observaciones
1ª Entrega			
2ª Entrega			

Evaluación Final del TP			
	Fecha de Aprobación	Firma	Observaciones
Escrito			
Carpeta			

Nota final del Trabajo Práctico	
Nota	
Firma	

Análisis del problema

El problema dado lo tratamos de comprender mediante un árbol de N branches (siendo N la cantidad de destinos por pasadas brindados a priori). Dado este árbol, cada nodo representa un “escaneo” de los códigos postales que ese nodo contiene, descomponiendo así a cada nodo hijo sus códigos postales. Esto significa que el árbol va reduciéndose a medida que caminamos en niveles más profundos.
Para dar un poco más de claridad a como comprendimos el problema, podemos usar el siguiente gráfico



Donde podemos ver que en el nodo ‘root’ (el “1”) tenemos los 3 codigos postales iniciales (“A”, “B”, “C”), esto quiere decir que al comienzo del problema se van a escanear todas las cajas de dichos códigos.

Si bajamos al segundo nivel podemos ver cómo se descomponen nuestros 3 codigos postales en un par de [1,2], donde un codigo postal “A” se envia automaticamente a su destino, mientras que los otros dos (“B” y “C”) requieren de un segundo escaneo.

Finalmente, en el tercer nivel podemos ver cómo se descomponen los códigos postales restantes (“B” y “C”) y son enviados ya que ningún nodo tiene más de un código postal (lo cual representaría un nuevo escaneo).

Habiendo comprendido el problema, podemos observar que hay varios factores a ser tenidos en cuenta a la hora de buscar la solución óptima, es decir lograr la clasificación correcta de las cajas para que lleguen a destino lo más rápido posible. Al contar con un scanner que nos facilitará dicha tarea de clasificación, aparecen diferentes escenarios según cómo se configure dicho scanner: cantidad de “destinos” o “rutas de salidas”, cantidad de códigos postales, cantidad de cajas por cada código postal. La combinación de todas estas variables hace que ante una solución que no es la óptima se demore más

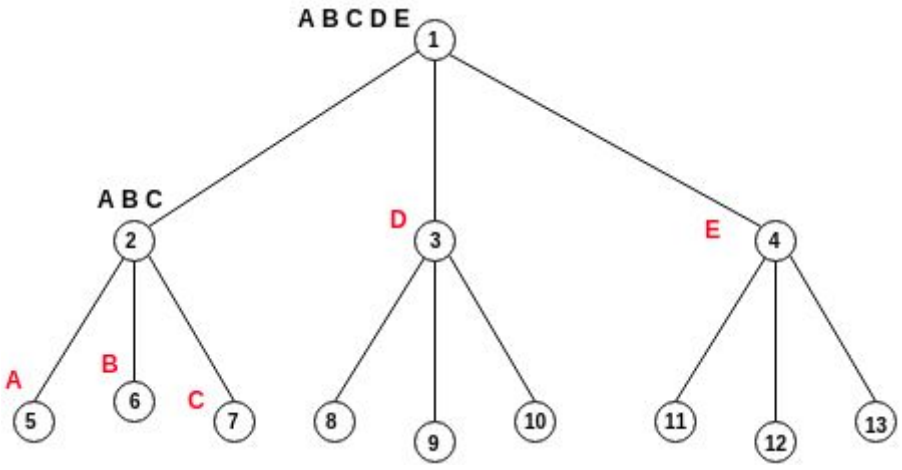
tiempo del ideal para cumplir que las cajas lleguen al destino correspondiente.

A continuación se ejemplifica mediante gráficos, la comparativa de tiempos totales ante diferentes configuraciones propuestas con el mismo escenario de datos:

Referencias:
Código postal que aún no ha sido enviado
Código postal que ha sido enviado

Datos:
Códigos postales: A, B, C, D, E
Cantidad de cajas por códigos postales:
A = 30, B = 10, C = 10, D = 10, E = 10
Tiempo de procesamiento de una caja: 1 segundo

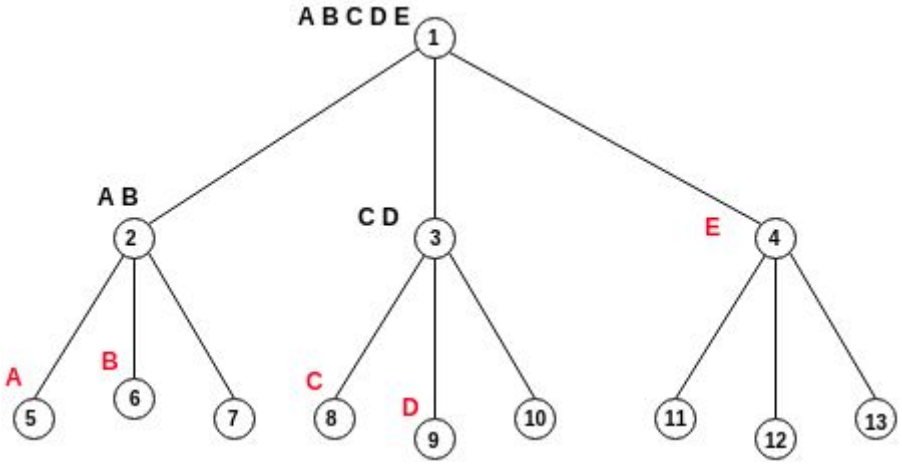
Caso 1



Conteo de tiempos:
Del destino 1 al 2 tiempo procesamiento: 50 segundos
Del destino 1 al 3 tiempo procesamiento: 10 segundos
Del destino 1 al 4 tiempo procesamiento: 10 segundos
Del destino 2 al 5 tiempo procesamiento: 30 segundos
Del destino 2 al 6 tiempo procesamiento: 10 segundos
Del destino 2 al 7 tiempo procesamiento: 10 segundos
Tiempo total de procesamiento de todas las cajas: 120 segundos

---

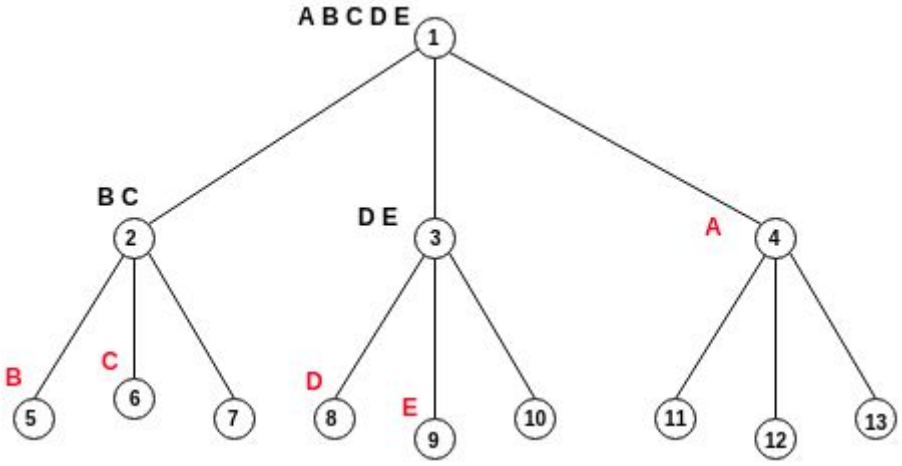
Caso 2



Conteo de tiempos:
Del destino 1 al 2 tiempo procesamiento: 40 segundos
Del destino 1 al 3 tiempo procesamiento: 20 segundos
Del destino 1 al 4 tiempo procesamiento: 10 segundos
Del destino 2 al 5 tiempo procesamiento: 30 segundos
Del destino 2 al 6 tiempo procesamiento: 10 segundos
Del destino 3 al 8 tiempo procesamiento: 10 segundos
Del destino 3 al 9 tiempo procesamiento: 10 segundos
Tiempo total de procesamiento de todas las cajas: 130 segundos

---

Caso 3

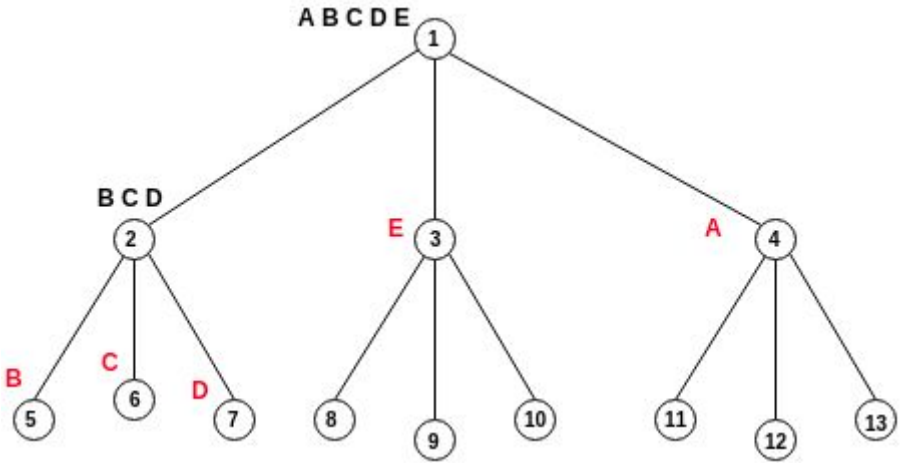


Conteo de tiempos:
Del destino 1 al 2 tiempo procesamiento: 20 segundos
Del destino 1 al 3 tiempo procesamiento: 20 segundos
Del destino 1 al 4 tiempo procesamiento: 30 segundos
Del destino 2 al 5 tiempo procesamiento: 10 segundos

Del destino 2 al 6 tiempo procesamiento: 10 segundos
Del destino 3 al 8 tiempo procesamiento: 10 segundos
Del destino 3 al 9 tiempo procesamiento: 10 segundos
Tiempo total de procesamiento de todas las cajas: 110 segundos

---

Caso 4



Conteo de tiempos:
Del destino 1 al 2 tiempo procesamiento: 30 segundos
Del destino 1 al 3 tiempo procesamiento: 10 segundos
Del destino 1 al 4 tiempo procesamiento: 30 segundos
Del destino 2 al 5 tiempo procesamiento: 10 segundos
Del destino 2 al 6 tiempo procesamiento: 10 segundos
Del destino 2 al 7 tiempo procesamiento: 10 segundos
Tiempo total de procesamiento de todas las cajas: 100 segundos

Con estos gráficos, lo que se quiere ejemplificar es que se deberá tener en cuenta a qué destino enviar cada código postal según la cantidad de cajas que tenga, así como también el hecho de si es conveniente agrupar códigos postales o “enviarlos” cuando se tiene la posibilidad. Claramente, ante un mismo escenario de datos, los tiempos de procesamiento finales variarán con todas la decisiones expuestas que debemos modelar para lograr la solución óptima.

Análisis de la solución

Inicialmente comenzamos planteando una solución basada en cada caja, en vez de por códigos postales. Esto fue porque intentamos hilar lo más fino posible y tener una decisión para cada elemento que era escaneado.

Este approach lo terminamos descartando debido a que nos generaba un problema de modelado bastante mayor a pensarlo como batches de “cajas en un código postal”, principalmente debido a que teníamos que relacionar cada caja con un código postal puntual y teníamos que tener una cantidad de variables astronómica.
Nos percatamos que lógicamente las cajas de un código postal nunca va a convenir particionarse en dos nodos del árbol, debido a que matemáticamente lo que harías sería “crear un nuevo código postal”, y esto solamente incrementa tu tiempo de procesamiento. De acá creamos una “pre condición tácita” que es que las cajas de un código postal siempre viajan juntas. No la declaramos como tal debido a que este caso como bien se dijo, no es óptimo y por lo tanto el modelo lo va a descartar de todas formas.

Continuando con nuestro modelado, inicialmente planteamos un modelo basado en bivalentes por aristas, las cuales iniciaban al prenderse si un código postal estaba viajando de un nodo a otro (parecido al problema del viajante). Esto tampoco nos resultó útil debido a que teníamos que generar un cubo de variables con dimensiones de [1..nodo inicial, 1..nodo final, A..codigo_postal]. Posiblemente sea un modelo viable, pero nos resultaba difícil procesarlo.

Finalmente tomamos un approach por nodo, donde teníamos una matriz de dimensiones [1..cantidad de nodos, A..codigo_postal], de valores bivalentes. Dicha matriz indicaba si el codigo postal se encontraba en el nodo o no. Teniendo esto como “tablero”, lo único que nos faltaba era marcar restricciones lógicas de cómo caminar el árbol y descomponerlo, haciendo hincapié en la matriz.

Por nombrar algunas:

- El primer nodo deberá tener todos los códigos postales (es decir, todas sus bivalentes prendidas)
- Si el nodo padre tiene más de un código postal, entonces la suma de los códigos postales en los nodos hijos deberá ser la misma.
- Si el nodo tiene 1 solo elemento, se lo considera un envío
- No se puede repetir un código postal en un nivel del árbol dado
- La cantidad de nodos enviados tiene que ser igual a la cantidad de códigos postales

Objetivo

El objetivo es **determinar** la clasificación y distribución de las cajas según su código postal y cantidad **para** que lleguen al destino, minimizando el tiempo de procesamiento de las cajas **para** el próximo despacho/pedido a realizar por la empresa de calzado deportivo.

Supuestos e Hipótesis

- Los tiempos de escaneo son iguales para cualquier caja, independiente del código postal
- El escáner no tiene errores y no habrá que cambiarlo por roturas o fallas, siempre escanea correctamente
- El tiempo de setup del escáner es despreciable
- El tiempo de escaneo es constante
- En el primer escaneo se escanean todos los códigos postales posibles
- Se considera un codigo postal 'enviado' si está en una cinta sin otros códigos-postales

Variables

a. **CANT_COD_POST:** Es la cantidad de códigos postales diferentes. Si tenemos por ejemplo "A, B y C", entonces **CANT_COD_POST** vale 3.

b. **DESTINOS_POR_PASADA:** Es la cantidad de salidas posibles que puede tomar una caja cuando entra al escáner.

c. **TIEMPO_PROC_CAJA:** Es el tiempo que tarda el escáner en procesar una caja, es decir cuánto le toma en decir a donde va.

d. **CAJAS**: Este es un array que indica la cantidad de cajas de cada código postal. Si tenemos por ejemplo [10, 20, 30], quiere decir que hay 10 cajas del primer codigo postal, 20 del segundo y 30 del tercero.

e. **NIVELES_ARBOL:** Indica la cantidad de niveles que posee el árbol sin tener en cuenta el inicial. Por ejemplo para uno de 7 nodos, con 2 DESTINOS_POR_PASADA y 3 CANT_COD_POST, esta variable toma valor 2.

f. **CANT_NODOS:** Es la cantidad de nodos que posee el árbol en total (incluyendo al nodo "root" (inicial)..

g. **CANTIDAD_CODIGOS_POSTALES_EN_NODO:** Es un array de 1 a la cantidad de nodos, el cual almacena en cada índice cuantos codigos postales se encuentran en dicho nodo. Por ejemplo el nodo inicial siempre tendrá la cantidad de códigos postales total (porque todos se escanean la primera vez).

h. **ENVIA_CODIGO_POSTAL:** Es un array de bivalentes para cada uno de los nodos. Para cada posición indica con un 1 si en ese nodo se envio algun codigo postal (o no hay directamente codigos postales en el nodo), 0 si no.

i. **ESCANEA_CODIGOS_POSTALES:** Es un array de bivalentes para cada uno de los nodos. Para cada posición indica con un 1 si en ese nodo se escanean códigos postales, 0 si no. Es el complemento de **ENVIA_CODIGO_POSTAL.**
j. **CODIGO_POSTAL_EN_NODO:** Es una matriz de bivalentes. Donde en cada posición está la bivalente que indica con un 1 si en el subárbol que arranca a partir de ese nodo se encuentra el código postal, 0 sino. Por ejemplo **COGIDO_POSTAL_EN_NODO**[2][3] indica si el segundo código postal se encuentra en el subárbol que arranca en el nodo 3.
k. **M**: Es un valor suficientemente grande usado para anular ciertas restricciones.
l. **CANTIDAD_NODOS_PADRE**: Es la cantidad de nodos padres que hay, es decir la cantidad de nodos menos los que son hoja.

Restricciones

CONDICION_INICIAL

Nos permite setear las condiciones iniciales del problema modelado. Para nuestro caso basta con decirle al modelo que la cantidad de códigos postales en el nodo ‘1’ (root) tiene que ser igual a la cantidad de códigos postales total.

NODO_ENVIA_CODIGO_POSTAL_O_ESCANEA

Nos permite relacionar ambas bivalentes como complementos entre sí. Un nodo que envía códigos postales no puede ser un nodo que escanea y viceversa. El nodo si o si debe ser una de las dos.

CANT_COD_POSTAL_POR_NODO

Nos permite relacionar el array de “cantidad de códigos postales por nodo” con la matriz que nos dice qué “código postal se encuentra en cada nodo”.

Esto hace que justamente nuestra condición inicial setee en la matriz para el nodo root todos en ‘1’ (porque todos los códigos postales están en el nodo inicial).

Obviamente, también nos permite relacionar ambas variables para que las restricciones de una apliquen a otra y no podamos tener cosas fuera de la lógica como códigos postales N veces (debido a que son bivalentes, van a tener que existir los necesarios para que sumados den la cantidad de CPs en el nodo)

RELACION_COD_POSTAL_PADRE_HIJO

Aca relacionamos la cantidad de códigos postales en el padre con la de los nodos hijos. **Si el nodo es de escaneo** entonces la suma de los códigos postales en los hijos deben tener la misma cantidad de códigos postales que el padre (en otra restricción validamos que sean los mismos!)

Por ejemplo, si tenemos 3 codigos postales en el nodo Padre, y son 2 nodos Hijos. La suma de la cantidad de códigos postales en los 2 nodos hijos deberá dar '3'.

COD_POSTAL_ENVIADOS

Por lo dicho anteriormente también podemos visualizar esta limitante. Nunca forzamos a que se 'envíen' todos los códigos postales. De esto se encarga esta restricción.

La cantidad de bivalentes prendidas como '**ENVIA_CODIGO_POSTAL**' debe ser igual a la cantidad de códigos postales total.
A su vez, aca definimos al modelo qué significa enviar un código postal. Para ello simplemente acotamos la "cantidad de códigos postales en un nodo" con la bivalente y un número **M** muy grande, para que, si hay **1** solo código postal en el nodo, entonces la bivalente '**ENVIA_CODIGO_POSTAL**' vale **1**, si no vale **0**

COD_POSTAL_FORWARDING

Esta regla es la descompuesta por la relación entre padre e hijo. Para cada código postal, si el padre 'escanea' entonces la suma de las bivalentes en los hijos para ese codigo postal tiene que ser igual a la bivalente del padre.
Dicho de otra forma: si el nodo padre tiene el código postal (bivalente en 1), entonces en todos los hijos debe prenderse **una sola bivalente**. Si el padre no lo tiene (bivalente en 0), entonces en todos los hijos todas las bivalentes deberán ser 0.

Función de minimización

La función de minimización la hicimos un poco más simple debido a una simplicidad matemática que notamos.
Idealmente debería ser todos los nodos que tienen más de un código postal (porque van a ser escaneados) deben aportar su cantidad de cajas por el tiempo de procesamiento.

Notamos que podemos desechar al padre y contar así todos los códigos postales restantes del árbol, ya que todos los enviados hacen la misma cantidad que el padre.

Por ende nuestra función de minimización es iterar a lo largo de todos los nodos (sin contar al padre) y códigos postales, sumando:
*"la bivalente en el nodo y código postal * (constante) la cantidad de cajas en ese código postal * (constante) el tiempo de procesamiento"*

Problemas encontrados

Más allá de los problemas al plantear la solución (los cuales fueron mencionados anteriormente, como atacar el problema a través de aristas o por cajas), también tuvimos algunos problemas durante el desarrollo, aunque fueron bastante sencillos de resolver:

1. No teníamos forma de aplicar un forAll solo a algunos elementos que cumplieran una condición (es decir, crear restricciones solo bajo un cierto caso). Existe un operador en OPL '=>' el cual nos permitió definir estos casos de forma totalmente transparente
2. No pudimos parsear el 'JSON' de códigos postales del '.dat'. Nos dimos cuenta que no nos interesaba en absoluto saber la letra del código postal sino que su índice. Por lo que cambiamos en todos los '.dat' el JSON de códigos postales por la cantidad ({'A', 'B', 'C'} ⇒ 3)
3. Tuvimos problemas con los 'ranges' de OPL, básicamente a veces comenzábamos en rangos distintos a los esperados, lo cual nos generaba Index Out Of Bounds.
4. Tuvimos algunos problemas a la hora de plantear algunas restricciones las cuales las estábamos "sobre analizando", como por ejemplo para especificar que los hijos de un padre que escanea tienen que tener todos los códigos postales del mismo, inicialmente planteamos una restricción enorme que buscaba que no se repitieran por nivel y más cosas. Esta restricción estaba llena de errores, y simplemente terminamos lográndolo con 2 restricciones mucho más legibles y simplificadas ("Divide y triunfarás").

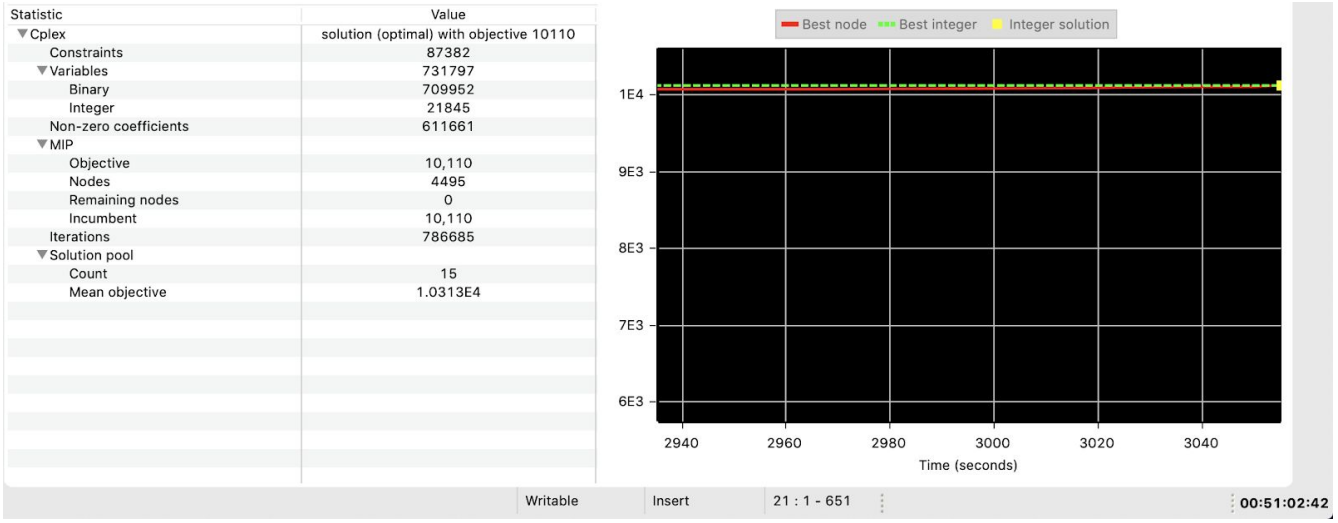
Tiempos de procesamiento

A simple vista parecería que el modelo escala cuadráticamente en función de la cantidad de códigos postales / destinos por pasada y de la entropía de las cajas (ya que si son todas con la misma cantidad el modelo converge mucho más rápido, ya que claro está es indistinto qué código postal poner en cada lugar).

En los modelos provistos por la cátedra, todos tardaron menos de un minuto con la excepción de los últimos 2. Los cuales tardaron 5' el de 19 códigos postales con distintas cajas y 50' el de 20 códigos postales con distintas cajas.

Esto lo atribuimos a la entropía de las cajas y a la cantidad de códigos postales / destinos, debido a que el modelo tiene una cantidad astronómica de variables (730.000 para el último caso) y tiene que ir yendo nodo a nodo acercándose al óptimo.

Adjuntamos una muestra del último caso provisto por la cátedra, donde podemos ver tiempos, iteraciones hechas, nodos evaluados, etc.



Statistic	Value
Cplex	solution (optimal) with objective 10110
Constraints	87382
Variables	731797
Binary	709952
Integer	21845
Non-zero coefficients	611661
MIP	
Objective	10,110
Nodes	4495
Remaining nodes	0
Incumbent	10,110
Iterations	786685
Solution pool	
Count	15
Mean objective	1.0313E4

Código del modelo

El código del modelo fue hecho en OPL, lo adjuntamos a continuación.

```
/*********************************************
 * OPL 12.9.0.0 Model
 * Author: Grupo 5
 *********************************************/

int CANT_COD_POST = ...; // Cambiamos el .dat de la catedra para que
utilice en vez del json {'A','B','C'} => CANT_COD_POST = 3;
int DESTINOS_POR_PASADA = ...; // Idem .dat de la catedra
int TIEMPO_PROC_CAJA = ...; // Idem .dat de la catedra
int CAJAS[1..CANT_COD_POST] = ...; // Idem .dat de la catedra

int NIVELES_ARBOL = ftoi(ceil((CANT_COD_POST - 1) / (DESTINOS_POR_PASADA
- 1)));

int CANT_NODOS = ftoi((sum(i in 1..NIVELES_ARBOL)
pow(DESTINOS_POR_PASADA, i)) + 1);

dvar int CANTIDAD_CODIGOS_POSTALES_EN_NODO[1..CANT_NODOS];
dvar int ENVIA_CODIGO_POSTAL[1..CANT_NODOS] in 0..1;
dvar int ESCANEA_CODIGOS_POSTALES[1..CANT_NODOS] in 0..1;

dvar int CODIGO_POSTAL_EN_NODO[1..CANT_COD_POST][1..CANT_NODOS] in 0..1;
```

```
int M = 10000000;
int CANTIDAD_NODOS_PADRE = (CANT_NODOS - ftoi(pow(DESTINOS_POR_PASADA,
NIVELES_ARBOL)));

minimize sum(nodo in 2..CANT_NODOS) sum(cp in 1..CANT_COD_POST)
CODIGO_POSTAL_EN_NODO[cp][nodo] * CAJAS[cp] * TIEMPO_PROC_CAJA;
subject to {
      CONDICION_INICIAL:
        CANTIDAD_CODIGOS_POSTALES_EN_NODO[1] == CANT_COD_POST;
      NODO_ENVIA_CODIGO_POSTAL_O_ESCANEA:
        forall(nodo in 1..CANT_NODOS) {
            ENVIA_CODIGO_POSTAL[nodo] + ESCANEA_CODIGOS_POSTALES[nodo]
== 1;
        }
      CANT_COD_POSTAL_POR_NODO:
       forall(nodo in 1..CANT_NODOS) {
             (sum(cp in 1..CANT_COD_POST)
CODIGO_POSTAL_EN_NODO[cp][nodo]) ==
CANTIDAD_CODIGOS_POSTALES_EN_NODO[nodo];
       }
      RELACION_COD_POSTAL_PADRE_HIJO:
        forall(padre in 1..CANTIDAD_NODOS_PADRE) {
            (ESCANEA_CODIGOS_POSTALES[padre] == 1) =>
                  (sum(hijo in ((padre * DESTINOS_POR_PASADA) + 1 -
DESTINOS_POR_PASADA + 1)..(padre * DESTINOS_POR_PASADA + 1))
CANTIDAD_CODIGOS_POSTALES_EN_NODO[hijo]) ==
CANTIDAD_CODIGOS_POSTALES_EN_NODO[padre];
        }
    COD_POSTAL_ENVIADOS:
      forall(nodo in 1..CANT_NODOS) {
         CANTIDAD_CODIGOS_POSTALES_EN_NODO[nodo] <=
ENVIA_CODIGO_POSTAL[nodo] + ((1 - ENVIA_CODIGO_POSTAL[nodo]) * M);
         CANTIDAD_CODIGOS_POSTALES_EN_NODO[nodo] >=
ENVIA_CODIGO_POSTAL[nodo];
      }
      (sum(nodo in 1..CANT_NODOS) ENVIA_CODIGO_POSTAL[nodo]) ==
CANT_COD_POST;
    COD_POSTAL_FORWARDING:
      forall(cp in 1..CANT_COD_POST) {
        forall(padre in 1..CANTIDAD_NODOS_PADRE) {
             (ESCANEA_CODIGOS_POSTALES[padre] == 1) =>
                 (sum(hijo in ((padre * DESTINOS_POR_PASADA) + 1 -
DESTINOS_POR_PASADA + 1)..(padre * DESTINOS_POR_PASADA + 1))
```

```
CODIGO_POSTAL_EN_NODO[cp][hijo]) == CODIGO_POSTAL_EN_NODO[cp][padre];
          }
        }
}
```

Para mas informacion podemos ver el repositorio github (https://github.com/sebastianripari/71.14_modelos_y_optimizacion_I) donde se encuentra el codigo hosteado.

Pruebas

Ejemplos de la cátedra

Como nota para los ejemplos a continuación, en las secciones de resultados no vamos a mostrar a dónde va a parar cada código postal porque sus matrices son enormes. Solo vamos a mostrar la función objetivo obtenida (validando con la de la cátedra). Si quisiéramos conocer la trazabilidad de cada código postal, basta con mirar dicha matriz de bivalentes y sabremos por qué nodo fue viajando cada uno (y finalmente en qué nodo "se envía").

Por ejemplo para 3 codigos postales con 2 destinos por pasada, podemos ver claramente que flujo siguió cada código postal.
En la pestaña de "*solutions*" encontraremos (entre muchas otras cosas):

CODIGO_POSTAL_EN_NODO = [[1 1 0 1 0 0 0]
[1 1 0 0 1 0 0]
[1 0 1 0 0 0 0]];
CANTIDAD_CODIGOS_POSTALES_EN_NODO = [3 2 1 1 1 0 0];

Podemos observar como:

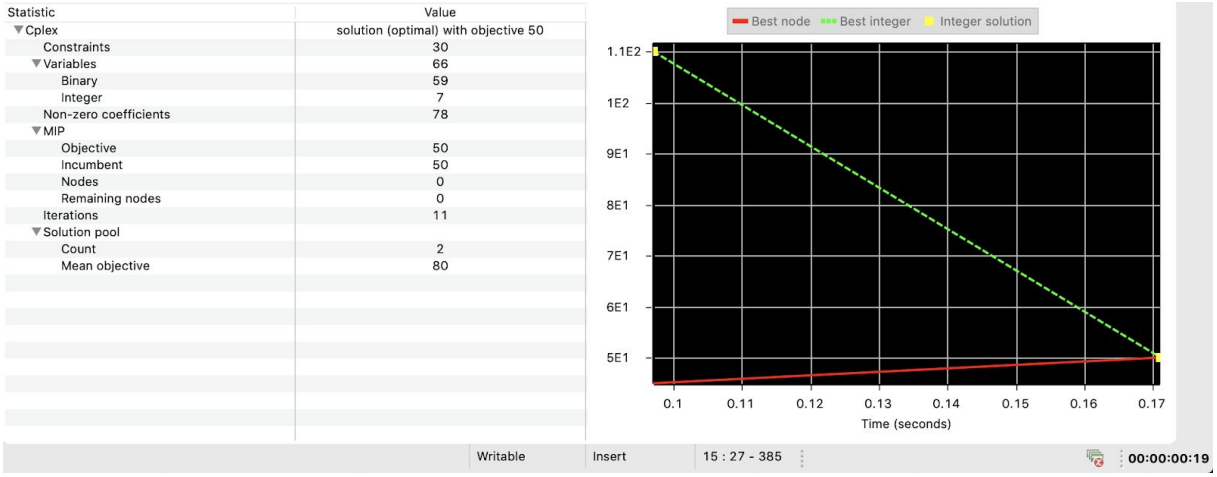
- En el nodo '1' (root) estan los 3 codigos postales -> Escanea
- En el nodo '2' (hijo de root, izquierda) esta el codigo postal 'A' y 'B' -> Escanea
- En el nodo '3' (hijo de root, derecha) esta el codigo postal 'C' -> Envía
- En el nodo '4' (hijo de 2, izquierda) esta el codigo postal 'A' -> Envía
- En el nodo '5' (hijo de 2, derecha) esta el codigo postal 'B' -> Envía
- En el nodo '6' y '7' no hay nada, porque ya se envió 'C' en su padre.

2019_2c_DistribucionCajas01_3Codpost_2destinosporpasada_mismotiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 3; //Cantidad de cod postales / cajas

CAJAS = [ 10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 2;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos

/*Output: tiempo 50 seg*/
```

Resultado



Resultado óptimo: 50

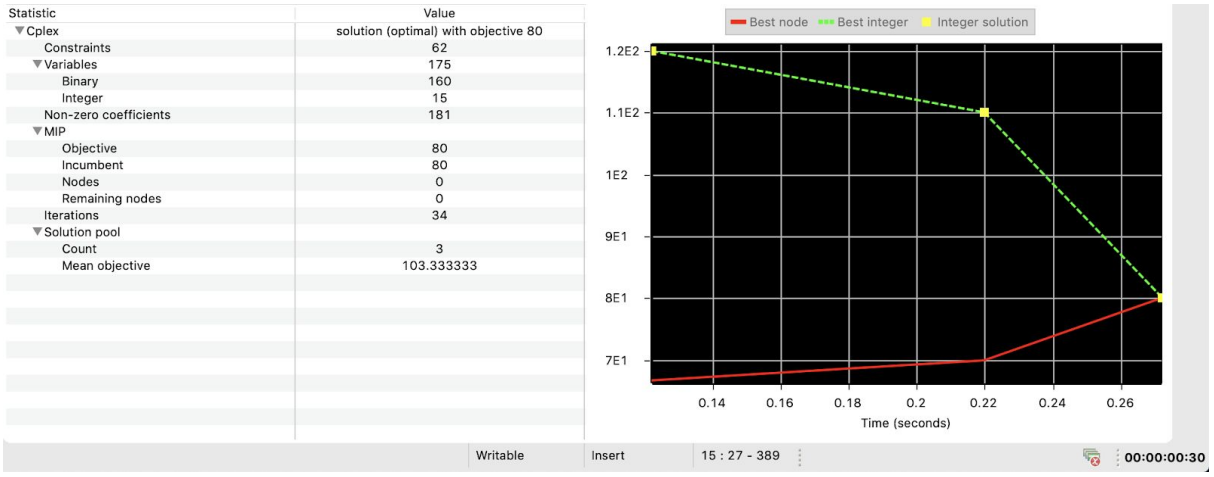
2019_2c_DistribucionCajas02_4Codpost_2destinosporpasada_mismotiempo

Dataset

```
/*********************************************
```

```
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 4; //Cantidad de cod postales / cajas

CAJAS = [ 10, 10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 2;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos

/*Output: tiempo 80 seg*/
```

Resultado



Resultado óptimo: 80

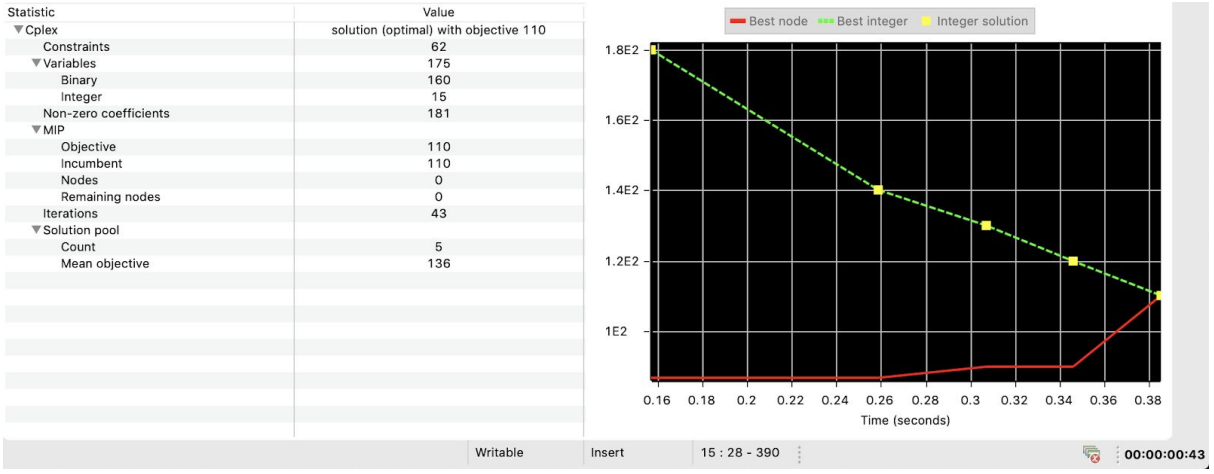
2019_2c_DistribucionCajas03_4Codpost_2destinosporpasada_disttiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 4; //Cantidad de cod postales / cajas
```

```
CAJAS = [ 30, 10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 2;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos

/*Output: tiempo 110 seg*/
```

Resultado



Statistic	Value
Cplex	solution (optimal) with objective 110
Constraints	62
Variables	175
Binary	160
Integer	15
Non-zero coefficients	181
MIP	
Objective	110
Incumbent	110
Nodes	0
Remaining nodes	0
Iterations	43
Solution pool	
Count	5
Mean objective	136

Resultado óptimo: 110

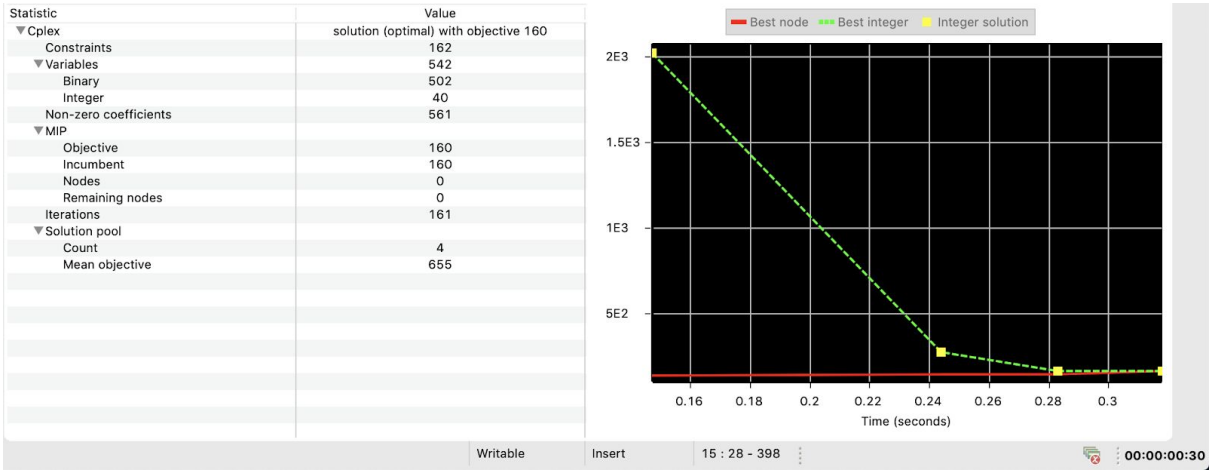
2019_2c_DistribucionCajas04_6Codpost_3destinosporpasada_disttiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 6; //Cantidad de cod postales / cajas

CAJAS = [ 30, 30, 10, 10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 3;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

```
/*Output: tiempo 160 seg*/
```

Resultado



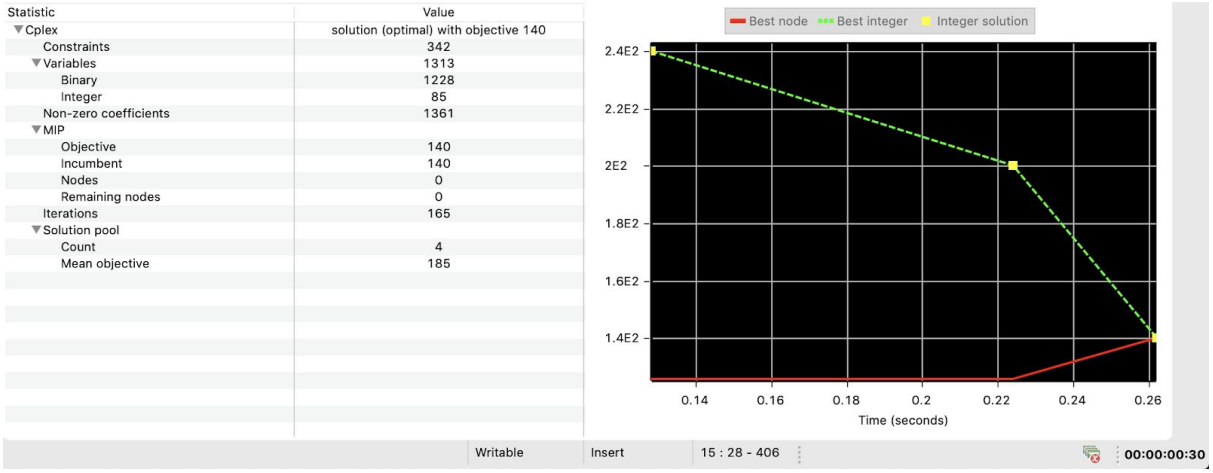
Resultado óptimo: 160

2019_2c_DistribucionCajas05_8Codpost_4destinosporpasada_mismotiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 8; //Cantidad de cod postales / cajas

CAJAS = [ 10, 10, 10, 10, 10, 10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos

/*Output: tiempo 140 seg*/
```

Resultado



Resultado óptimo: 140

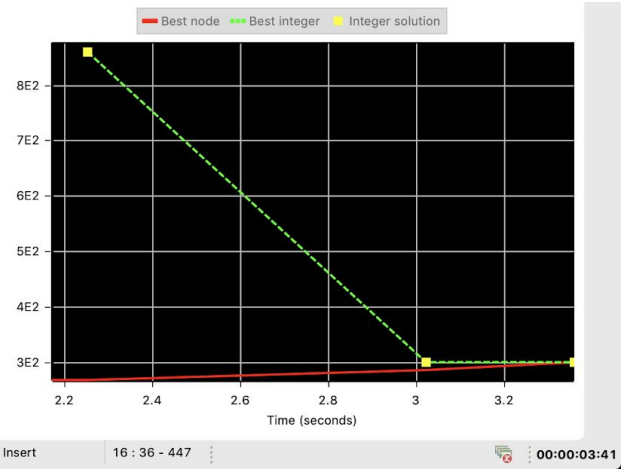
2019_2c_DistribucionCajas06_15Codpost_4destinosporpasada_mismotiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 15; //Cantidad de cod postales / cajas

CAJAS = [   10, 10, 10, 10, 10, 10, 10, 10,
                10, 10, 10, 10, 10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

Resultado

Statistic	Value
Cplex	solution (optimal) with objective 300
Constraints	5462
Variables	35482
Binary	34117
Integer	1365
Non-zero coefficients	31396
MIP	
Objective	300
Nodes	0
Remaining nodes	0
Incumbent	300
Iterations	1750
Solution pool	
Count	2
Mean objective	580



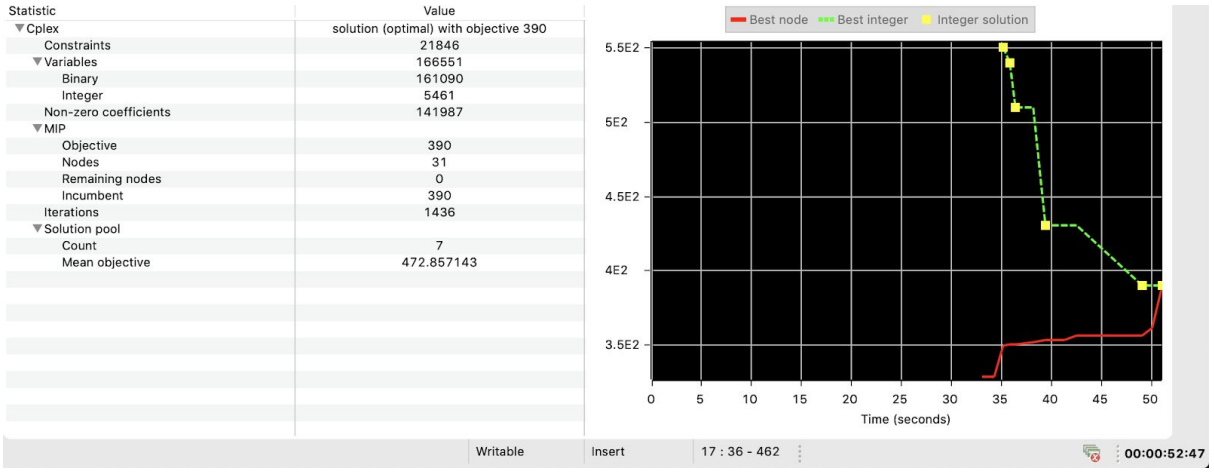
Resultado óptimo: 300

2019_2c_DistribucionCajas07_18Codpost_4destinosporpasada_mismotiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 18; //Cantidad de cod postales / cajas

CAJAS = [   10, 10, 10, 10, 10, 10, 10, 10,
                10, 10, 10, 10, 10, 10, 10, 10,
                10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

Resultado



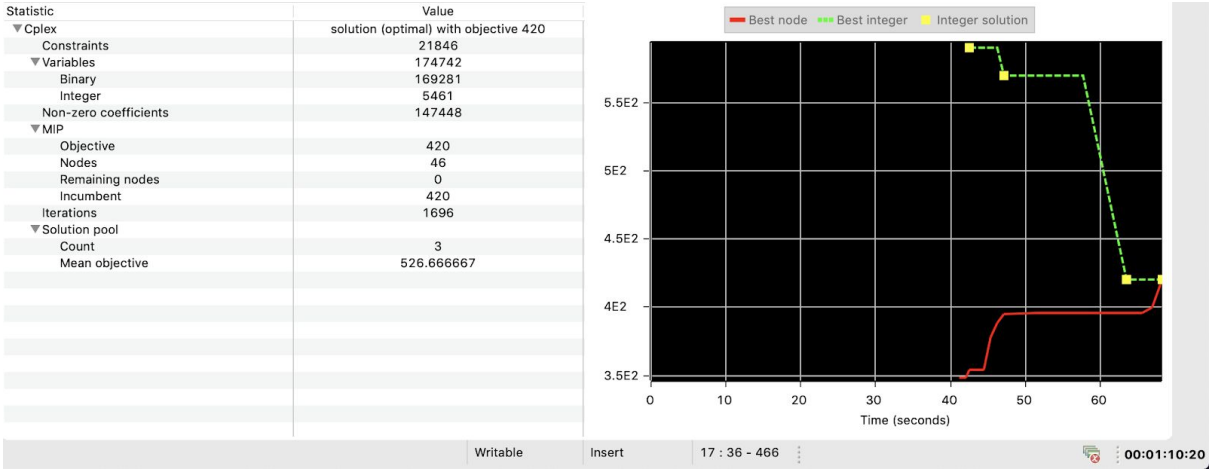
Resultado óptimo: 390

2019_2c_DistribucionCajas08_19Codpost_4destinosporpasada_mismotiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 19; //Cantidad de cod postales / cajas

CAJAS = [   10, 10, 10, 10, 10, 10, 10, 10,
                  10, 10, 10, 10, 10, 10, 10, 10,
                  10, 10, 10 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

Resultado



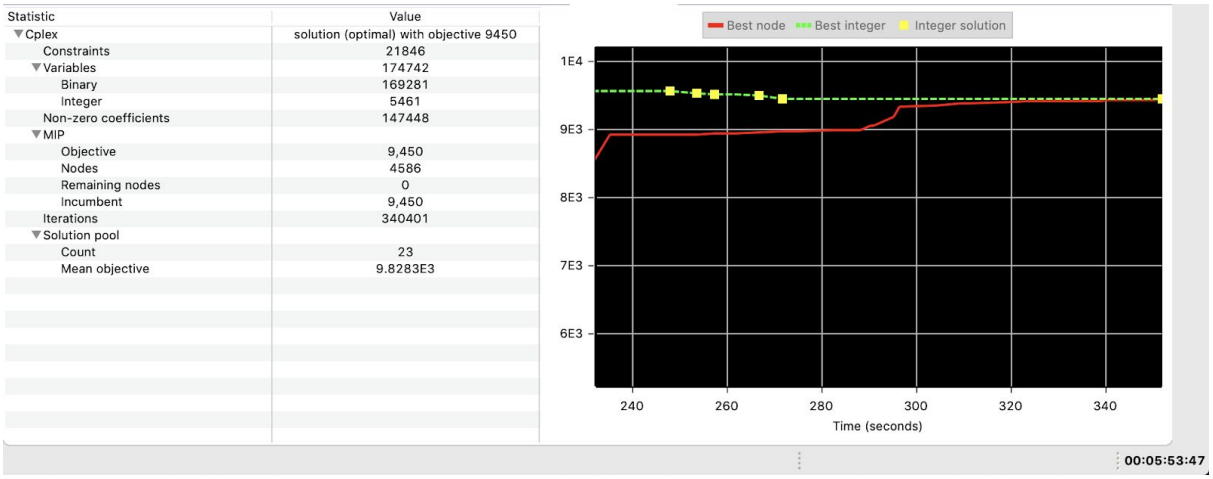
Resultado óptimo: 420

2019_2c_DistribucionCajas09_19Codpost_4destinosporpasada_disttiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 19; //Cantidad de cod postales / cajas

CAJAS = [   60,   20,   10,   40,   80,   900,  50,   10,
                  400,  450,  200,  1200,       1000,       50,   90,
500,
                  30,   120,  300 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

Resultado



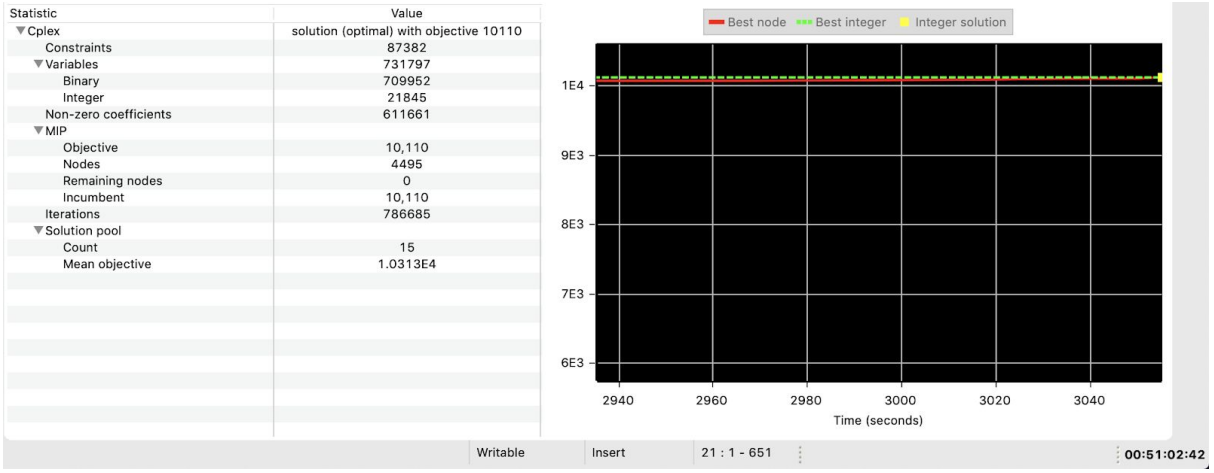
Resultado óptimo: 9.450

2019_2c_DistribucionCajas10_20Codpost_4destinosporpasada_disttiempo

Dataset

```
/*********************************************
 * OPL 12.7.1.0 Data
 * Author: La catedra
 * Creation Date: Aug 29, 2019 at 8:44:07 PM
 *********************************************/

CANT_COD_POST = 20; //Cantidad de cod postales / cajas

CAJAS = [   60,   20,   10,   40,   80,   900,  50,   10,
                  400,  450,  200,  1200,       1000,       50,   90,
500,
                  30,   120,  300,  200 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

Resultado



Resultado óptimo: 10.110

Ejemplos custom

Para los primeros 4 ejemplos fijamos una cantidad de códigos postales y una cantidad de cajas, solamente vamos a ir variando los destinos por pasada.

Nuestra idea fue buscar una cantidad grande de códigos postales pero que tenga resolucion analitica rapida, simplemente para poder validar que el modelo funciona correctamente.

Para el último ejemplo variamos la cantidad de códigos postales y destinos por pasada como bien pide el enunciado del trabajo. Utilizamos el ejemplo postulado en la fase de 'Análisis del problema' para demostrar que el resultado es el esperado.

Para todos los ejemplos presentaremos además de la captura de statistics, el tiempo de ejecución dentro de 'Engine log' como bien pide el enunciado.

Cuatro destinos

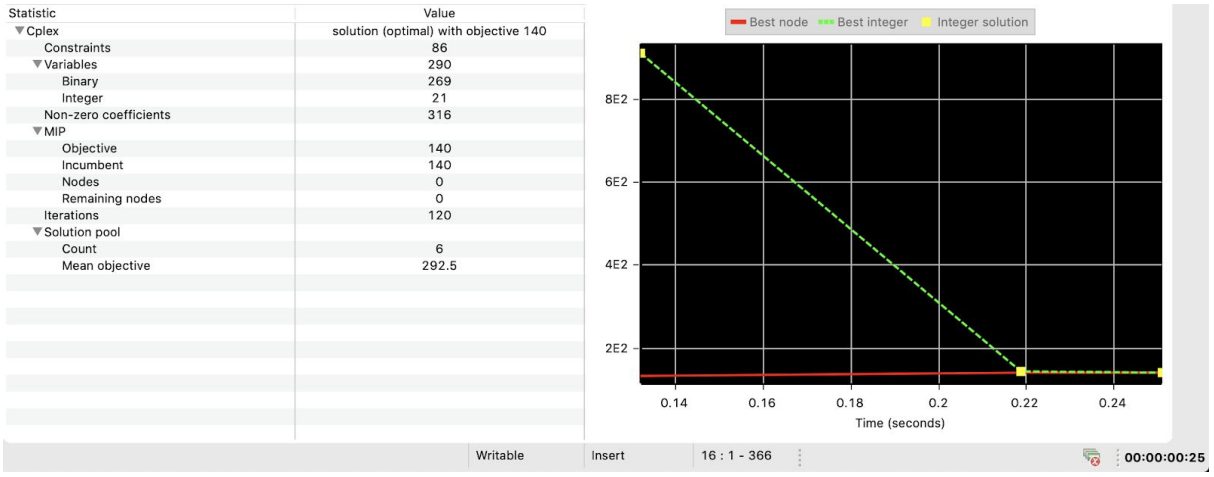
Dataset

```
CANT_COD_POST = 7; //Cantidad de cod postales / cajas

CAJAS = [ 20, 1, 3, 40, 4, 60, 2 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos

/* Resultado debería ser
```

```
   Nivel root: 60/40/20/1/2/3/4
   Nivel uno: 60 - 40 - 20 - 1/2/3/4 => 130s
   Nivel dos branch 1/2/3/4: 1 - 2 - 3 - 4 => 10s

   Total tiempo: 140s
*/
```

Resultado

```
Root node processing (before b&c):
  Real time             =    0.11 sec. (13.96 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.11 sec. (13.96 ticks)
```



Resultado óptimo: 140

Cuatro destinos con distinto tiempo de procesamiento

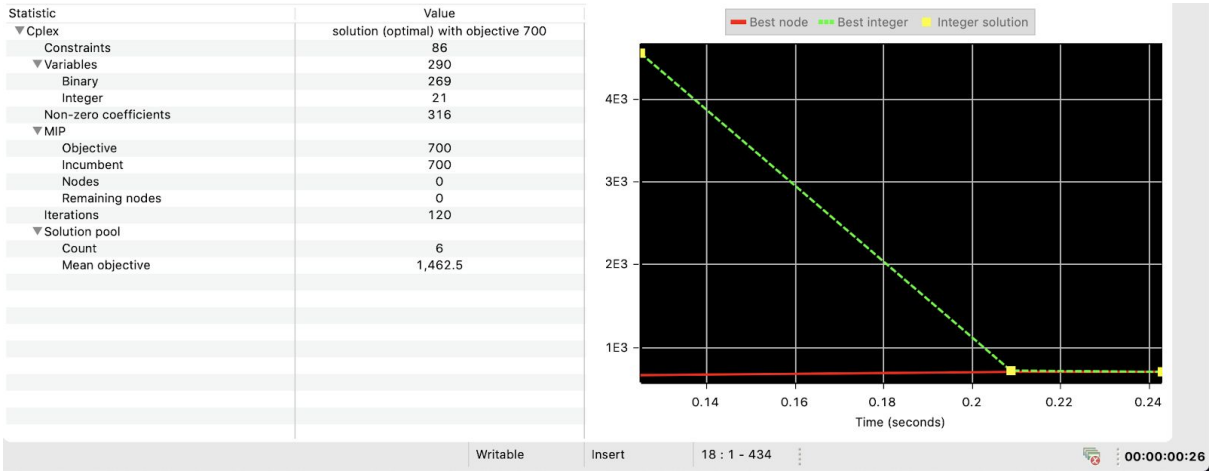
Dataset

```
// Idéntico al otro solo que con processing time de 5s

CANT_COD_POST = 7; //Cantidad de cod postales / cajas

CAJAS = [ 20, 1, 3, 40, 4, 60, 2 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 4;

TIEMPO_PROC_CAJA = 5; //Expresado en segundos
```

```
/* Resultado debería ser
   Nivel root: 60/40/20/1/2/3/4
   Nivel uno: 60 - 40 - 20 - 1/2/3/4 => 130s
   Nivel dos branch 1/2/3/4: 1 - 2 - 3 - 4 => 10s

   Total tiempo: 140s * 5s = 700s
*/
```

Resultado

```
Root node processing (before b&c):
  Real time             =    0.10 sec. (13.96 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.10 sec. (13.96 ticks)
```



Resultado óptimo: 700

Tres destinos

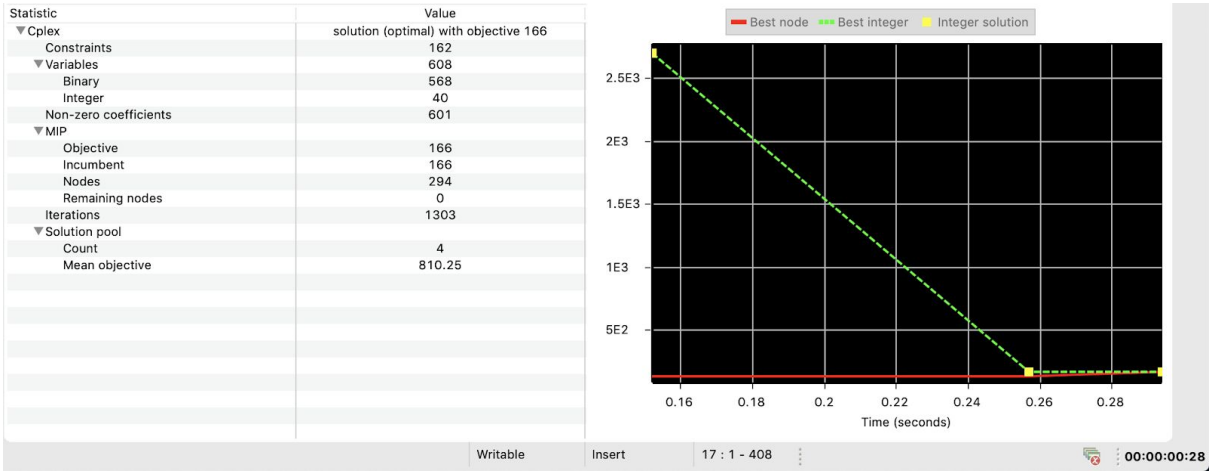
Dataset

```
CANT_COD_POST = 7; //Cantidad de cod postales / cajas

CAJAS = [ 20, 1, 3, 40, 4, 60, 2 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 3;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

```
/* Resultado debería ser
   Nivel root: 60/40/20/1/2/3/4
   Nivel uno: 60 - 40 - 20/1/2/3/4  => 130s
   Nivel dos branch: 20/1/2/3/4: 20 - 4 - 1/2/3 => 30s
   Nivel tres branch: 1 - 2 - 3 => 6s

   Total tiempo: 166s
*/
```

Resultado

```
Root node processing (before b&c):
  Real time             =    0.10 sec. (21.87 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.05 sec. (14.43 ticks)
  Sync time (average)   =    0.02 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.15 sec. (36.30 ticks)
```



Resultado óptimo: 166

Dos destinos

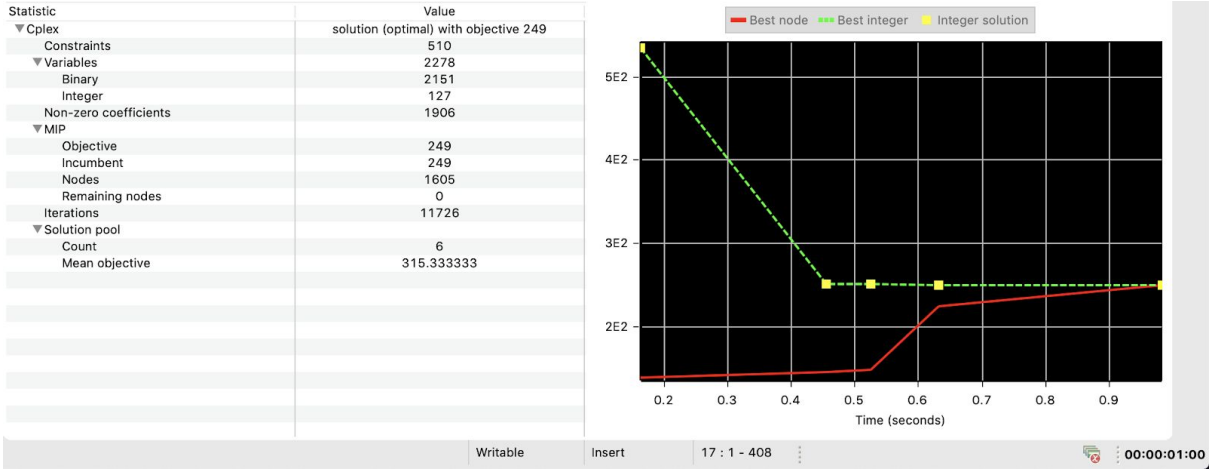
Dataset

```
CANT_COD_POST = 7; //Cantidad de cod postales / cajas

CAJAS = [ 20, 1, 3, 40, 4, 60, 2 ]; //Cajas por cod post

DESTINOS_POR_PASADA = 2;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos
```

```
/* Resultado debería ser
   Nivel root: 60/40/20/1/2/3/4
   Nivel uno: 60 - 40/20/1/2/3/4  => 130s
   Nivel dos branch: 40/20/1/2/3/4: 40 - 20/1/2/3/4 => 70s
   Nivel tres branch: 20 - 1/2/3/4 => 30s
   Nivel cuatro branch: 4 - 1/2/3 => 10s
   Nivel cinco branch: 3 - 1/2 => 6s
   Nivel seis branch: 2 - 1 => 3s

   Total tiempo: 249s
*/
```

Resultado

```
Root node processing (before b&c):
  Real time             =    0.24 sec. (97.22 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.66 sec. (238.13 ticks)
  Sync time (average)   =    0.33 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.90 sec. (335.35 ticks)
```



Resultado óptimo: 249

Caso de análisis de problema

Mostramos a continuación el caso que se utilizó para explicar el análisis del problema.

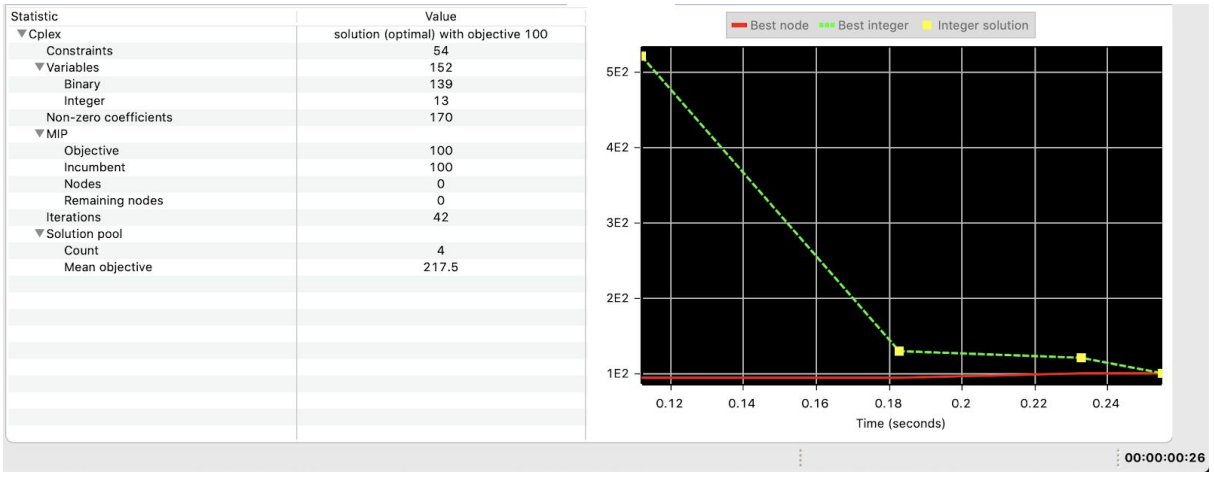
Dataset

```
CANT_COD_POST = 5; //Cantidad de cod postales / cajas

CAJAS = [ 30, 10, 10, 10, 10 ]; //Cajas por cod post
```

```
DESTINOS_POR_PASADA = 3;

TIEMPO_PROC_CAJA = 1; //Expresado en segundos

/* Resultado debería ser
   Nivel root: 30/10/10/10/10
   Nivel uno: 30 - 10 - 10/10/10 => 70s
   Nivel dos branch: 10 - 10 - 10 => 30s

   Total tiempo: 100s
*/
```

Resultado

```
Root node processing (before b&c):
  Real time             =    0.15 sec. (3.50 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.15 sec. (3.50 ticks)
```



Resultado óptimo: 100

Conclusiones

Una de las conclusiones principales que nos llevamos del TP es que al modelo solo hay que especificarle las reglas de juego, y no cómo jugar. En otras palabras: especificarle las restricciones pero no decirle qué hacer.

En un principio, cuando estábamos planteando en lápiz y papel el modelo, pasamos algunas horas pensando cómo íbamos a hacer para decirle al modelo que los códigos postales con menos cajas los envíe (“mate”) primero.

Luego, con el correr de las horas, nos dimos cuenta que esto era trabajo del modelo y no nuestro. Con darle las restricciones correctas, él mismo iba a ser capaz de elegir la mejor opción.

Otra conclusión relacionada con el rendimiento y performance, es que no se debe poner restricciones de más o sobrantes (que no agreguen valor alguno), sino la mínima cantidad, justa y necesaria, para evitar que dado un input grande el modelo nunca termine de correr debido a la excesiva cantidad de restricciones. A menor cantidad de restricciones más rápido converge. Si bien al principio es conveniente tener una versión funcional, si se quiere no del todo optimizada, luego hay que dedicar tiempo en aplicar un “refactoring” que logre un mismo resultado en menor tiempo. Esto no es menor, ya que en la vida real en muchas ocasiones, se necesita que el output sea brindado en el menor tiempo posible,permitiendo la toma de decisiones y acciones de forma rápida.

Esta última conclusión aplica también para la cantidad de variables: siempre que se pueda quitar variables “sobrantes”, mucho mejor. Nuevamente, esto hará que el modelo tenga menos decisiones a determinar y por lo tanto termine más rápido.

Finalmente, podemos concluir, que como en todo proceso con resultados, siempre hay costos y beneficios: los modelos más simples, es decir con menos restricciones y variables, y por tanto efectivos / eficientes, suelen ser los más complejos de lograr, ya que requieren de un mayor tiempo de trabajo y análisis para llegar ellos.
Creemos que se consiguen cuando uno realmente llega a una comprensión total del esquema del problema.
En este sentido, nos llamó la atención lo parecido que es al proceso de encarar un desarrollo de software, que es donde nosotros tenemos nuestro campo laboral actualmente y nos desempeñamos con mayor soltura y facilidad.

Es muy ambicioso querer lograr en una primera instancia un modelo que cumpla con todo lo pedido y que además sea simple en cuanto a restricciones y variables. En la programación ocurre lo mismo. Hoy en día hay muchas metodologías de trabajo que proponen soluciones iterativas e incrementales. Por lo tanto, creemos que el camino que hemos adoptado y el que nosotros podemos recomendar es que primero habrá que enfocarse en tener un modelo que cumpla lo pedido, despreciando la simpleza en cuanto a restricciones y variables, para luego ir simplificando y puliendo de forma iterativa e incremental, revisando por supuesto de no corromper la lógica del problema.

Lo ideal, es que el resultado de este proceso, sea el de un modelo simple, eficaz, eficiente, que facilite su legibilidad y comprensión para el que lo vaya a estudiar en el futuro y tenga que mantenerlo o seguir escalándolo.