71.14 Modelos y Optimización I

1º Cuatrimestre 2022
Trabajo Práctico - Segunda entrega
Facultad de Ingeniería
Universidad de Buenos Aires

Ponce Matias 89820

Análisis de la situación

Se trata de un problema combinatorio estilo del viajante ya que se cumplen las condiciones:

1.- Se vuelve al lugar de origen

2.- Todas las sucursales se visitan una vez

3.- Ninguna sucursal se visita más de una vez

4.- No se conoce el orden y el valor optimo del funcional depende del orden elegido

Se tiene la problemática que no se pueden armar todas las configuraciones posibles y luego elegir la mejor ya que al ser un numero grande sucursales el número de combinaciones es exponencial.

Ideas de cómo lo van a intentar resolver

Primera entrega

Partiendo desde la central se seleccionan los posibles destinos (según capacidad mínima y máxima del camión).

De los posibles destinos se selecciona el mejor (menor distancia)

Luego en cada iteración (para cada sucursal) se buscan los siguientes destinos posibles (según capacidad mínima y máxima del camión. Además, debe ser una sucursal que no se haya visitado antes).

De los posibles destinos se selecciona el mejor (menor distancia).

Al final se muestra el circuito recorrido.

Segunda entrega

Se mantiene lógica inicial usada en la primera entrega pero se aplica algunas mejoras ya que el programa propuesto presentaba errores de memoria **STD::BAD_ALLOC**

Tercer Entrega

Para esta tercera entrega se quita la condición de validación de demandas de cada sucursal. También se quita la condición de capacidad del camión quedando solamente la elección de la sucursal con menor distancia

Comentarios sobre los cambios

Primera entrega

En un primer momento intenté armar todas las combinaciones posibles y luego quedarme con la mejor, pero vi que esto es inviable dada la cantidad de sucursales. Como se tenia un incremento exponencial se consumía mucho tiempo y memoria.

Segunda entrega

Se presentó el principal problema de **STD::BAD_ALLOC,** que es una excepción arrojada por el sistema por falta de recursos de memoria.

La falta de recursos de memoria se debía a una ineficiente programación que se había propuesto en la primera entrega.

Entre las mejoras propuestas se implementaron:

- Uso de memoria dinámica y liberación.
- Pasaje de variables por referencia en lugar de copias.
- Evitar recorridos en listas innecesarios.
- Evitar recursividad
- No evaluar sucursales ya visitadas innecesariamente.

Tercera entrega

Los cambios implementados no modificación el problema inicial que es resolver un problema estilo el viajante. Solo se quitan condiciones que se deben cumplir con respecto a la capacidad del camión y la demanda de cada sucursal.

Objetivo

Determinar el orden en qué visitar las sucursales para poder visitar todas cumpliendo las restricciones del camión para minimizar distancias en el periodo determinado.

Hipótesis y supuestos:

1.- Desde la central se puede visitar cualquier sucursal

2.- Desde una sucursal se puede visitar cualquier otra sucursal

3.- Todas las sucursales deben ser visitadas una única vez

4.- Se dispone del tiempo para cumplir con el recorrido determinado por el modelo

5.- Se puede ir a la central desde cualquier sucursal

6.- Se toma como ubicación de la sucursal central el (0,0)

7.- El camión sale desde la central sin dinero

Constantes

CANT_SUCUR: Cantidad de sucursales a recorrer

SUCURSALES: sucursales a las que se debe hacer el recorrido

$MAX_DINERO: Capacidad máxima que tiene el camión

Distanciai,j: Distancia para ir de sucursal i a sucursal j

VMonto$_i$: Variación de monto de en sucursal i

Variables

Monto$_i$: monto con el que cuenta el camión durante el recorrido en el lugar i

$Y_{i,j}$: bivalente que indica si una sucursal es visitada desde i a j

Restricciones

Llegar una única vez a una sucursal

$$\sum_{\substack{i=0 \\ j \neq i}}^{SUCURSALES} Y_{i,j} = 1 \quad \forall i \in SUCURSALES \text{ U Central}$$

Salir una única vez a una sucursal

$$\sum_{\substack{j=0 \\ j \neq i}}^{SUCURSALES} Y_{i,j} = 1 \quad \forall j \in SUCURSALES \text{ U Central}$$

Evitar subtorus

$$Ui - Uj + CANT\_SUCUR * Yi,j \leq CANT\_SUCUR - 1 \quad \forall i,j \in SUCURSALES$$

Restricciones de montos

Nota: por no negatividad de las variables no hace falta validar que Monto > 0.

$Monto_i$ <= **$MAX_DINERO**

$Monto_i = \sum_{i=1}^{sucursales} Y_{i,j} * VMonto_i$

Función Objetivo

$$Z(min) = \sum_{i=1}^{SUCURSALES} \sum_{j=1}^{SUCURSALES} Y_{i,j} * Distancia_{i,j}$$

CPLEX

Eliminación de subtours TPS

En la formulación TPS, se propone la eliminación de subtours agregando nuevas condiciones de la forma:

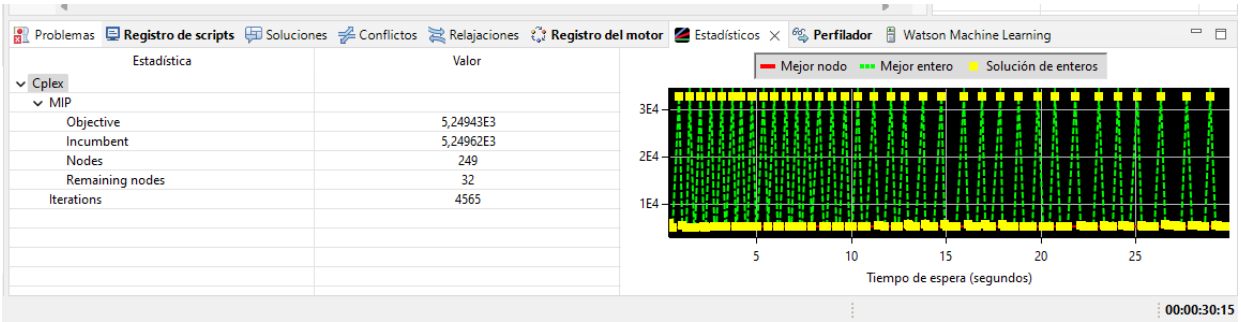
$$\sum_{\substack{i,j \in M \\ i \neq j}} x_{ij} \leq |M| - 1 \qquad \forall\, M \subset N \text{ such that } \{1\} \notin M, |M| \geq 2$$

En el modelo entregado podemos observar la eliminación de los subtours con el agregado del siguiente código:

```
// Subtour elimination constraints.
forall ( s in subtours )
  sum ( i in Cities : s.subtour[i] != 0 ) x[< minl ( i, s.subtour[i] ), maxl
    ( i, s.subtour[i] ) >] <= s.size - 1;
```

En este código se observa que para cada subtour verifica que la sumatoria debe ser siempre menor al tamaño - 1. Esta es la implementación de las condiciones dadas por TPS mostrada más arriba.

Grafica TPS



Eliminación de subtours MTZ

En la formulación MTZ, se propone la eliminación de subtours agregando nuevas condiciones de la forma:

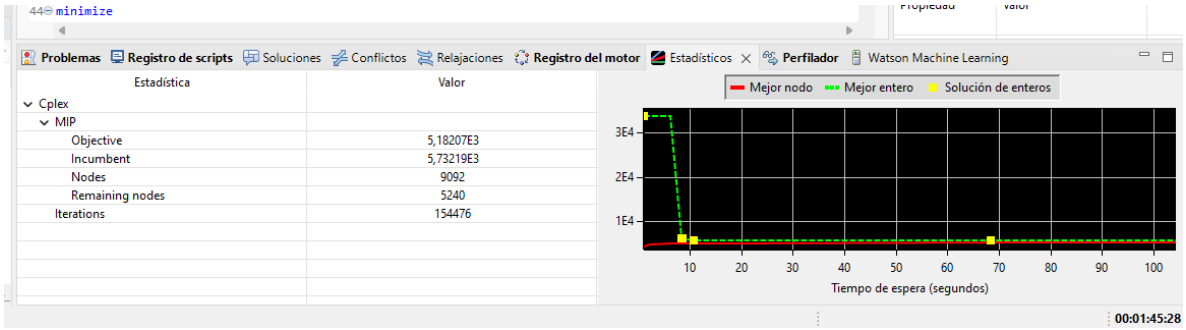
$$Ui - Uj + n\,Yij \leq n - 1 \qquad \forall i, \forall j, i \neq j \; i=1...n$$

En el modelo entregado podemos observar la eliminación de los subtours con el agregado del siguiente código:

```
forall ( i in cities : i > 1, j in cities : j > 1 && j != i )
  subtour:
    u[i] - u[j] + ( n - 1 ) * x[< i, j >] <= n - 2;
```

En este código se observa la condición de secuencia y de correlación unitaria, pero con el detalle de la condición N - 2 en lugar de N - 1 dada en la formula general. Esto se debe a que el orden de visita 1 está asignado al punto de partida para el modelo del viajante.

Grafica MTZ



Comparación de modelos

Si bien en ambas formulaciones se deben agregan nuevas restricciones para el control de la formación de los subtours, en la formulación TPS se depende mucho del tamaño del problema. Al agregar las condiciones para la eliminación de subtours se puede hacer inmanejable el número de restricciones que el modelo debe cumplir.

En los modelos entregados, se observa una diferencia en los armados de las aristas:

1.- El armado para el modelo TPS

```
setof ( edge ) Edges = {< i, j > | ordered i, j in Cities};
```

2.- El armado para el modelo MTZ

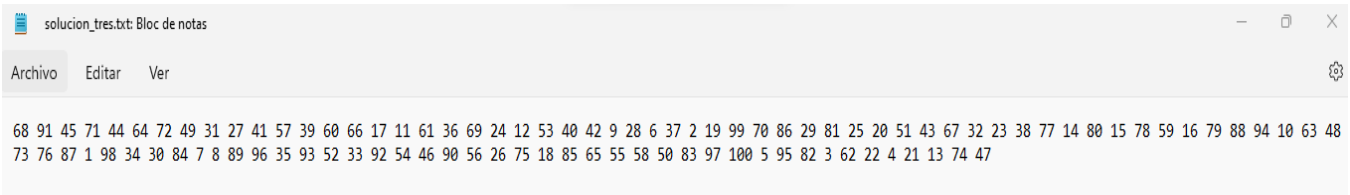
```
setof ( edge ) edges = {< i, j > | i, j in cities : i != j};
```

Si bien en general el modelo TPS tiene muchas mas restricciones para evitar los subtours, en esta implementación se observa el uso del **ORDERED,** lo cual hará que el número de variables sea mucho menor ya que crea una única arista entre los vértices <i,j>. Esto provoca una gran mejora en la respuesta a la solución a favor del modelo TPS.

Para que el **ORDERED** sea valido los pesos de cada arista deben ser iguales para ir del i -> j cómo para ir del j -> i.

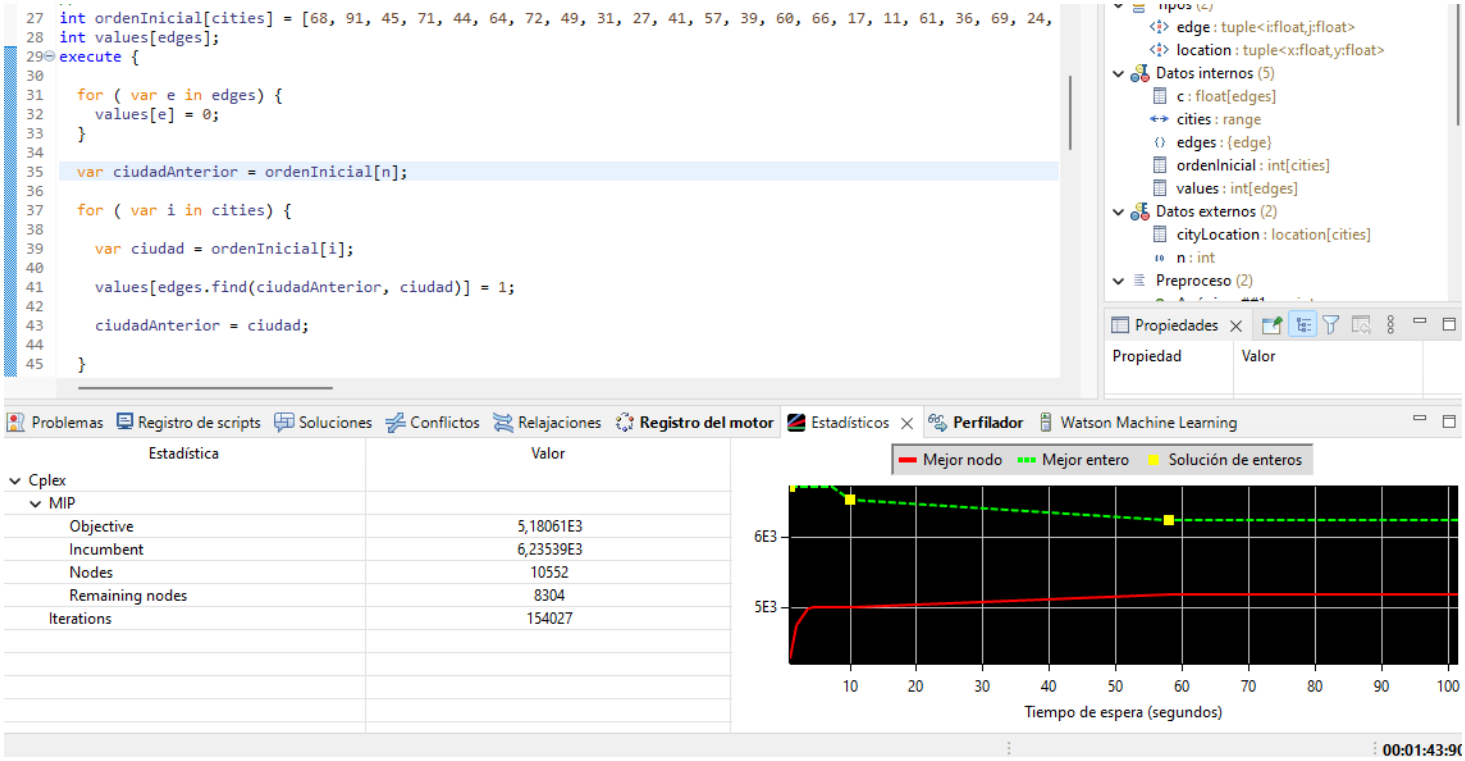
Modificación de Heurística

Luego de la modificación realizada al desarrollo para no tener en cuenta las demandas se encuentra la siguiente solución:



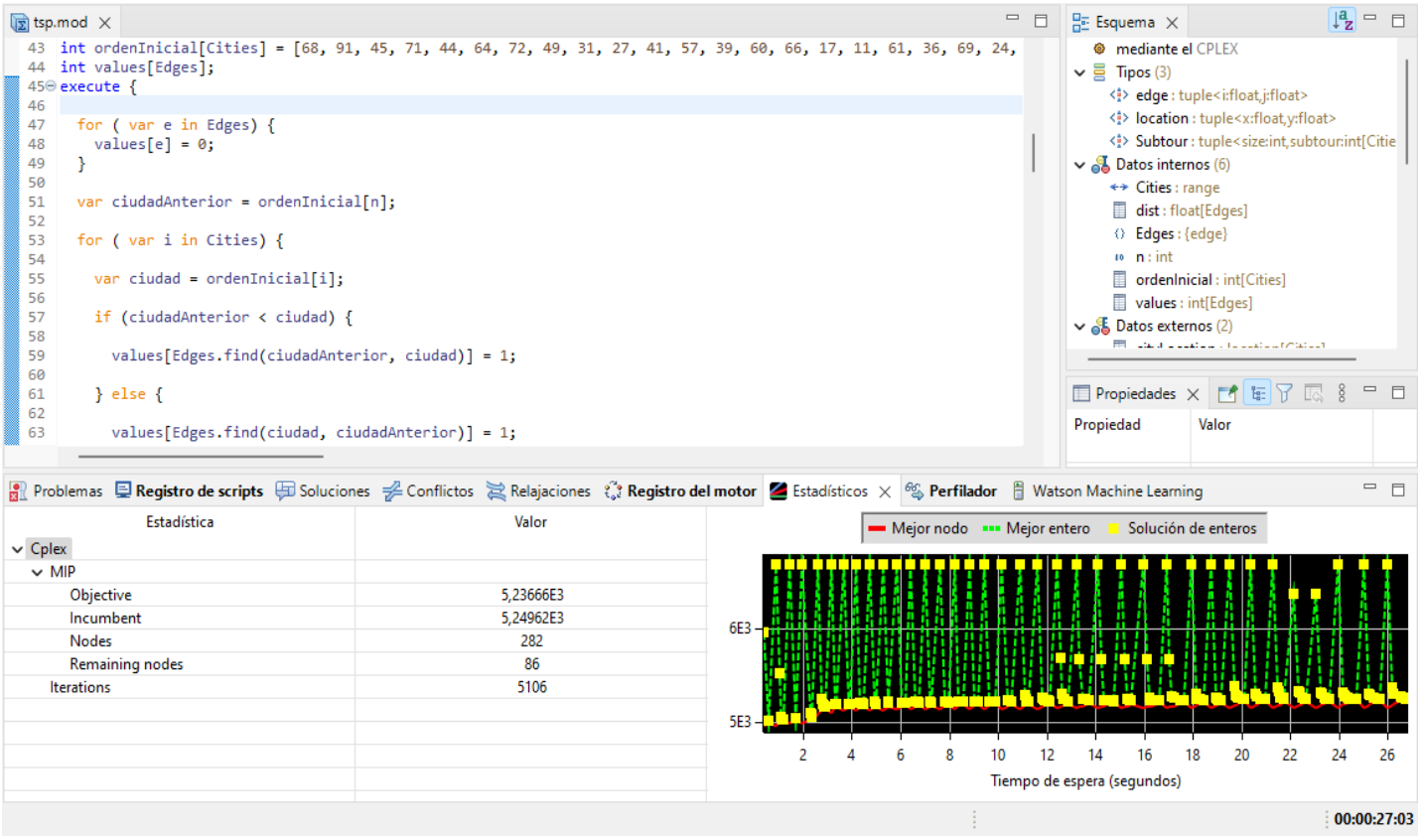
Se modifican ambos códigos entregados para inyectar como solución inicial la obtenida de la modificación de la Heurística:

Ejecución MTZ + solución inicial



Se observa que luego de ingresar la nueva solución inicial el sistema se demora más tiempo en empezar a converger a la mejor solución. Esto se debe al algoritmo que se utilizó para encontrar la solución inicial el cual generó un orden en las sucursales que provocó esta mala convergencia.

Ejecución TPS + solución inicial


tsp.mod
43 int ordenInicial[Cities] = [68, 91, 45, 71, 44, 64, 72, 49, 31, 27, 41, 57, 39, 60, 66, 17, 11, 61, 36, 69, 24,
44 int values[Edges];
45 execute {
46
47 for (var e in Edges) {
48 values[e] = 0;
49 }
50
51 var ciudadAnterior = ordenInicial[n];
52
53 for (var i in Cities) {
54
55 var ciudad = ordenInicial[i];
56
57 if (ciudadAnterior < ciudad) {
58
59 values[Edges.find(ciudadAnterior, ciudad)] = 1;
60
61 } else {
62
63 values[Edges.find(ciudad, ciudadAnterior)] = 1;
Esquema
mediante el CPLEX
Tipos (3)
edge : tuple<i:float,j:float>
location : tuple<x:float,y:float>
Subtour : tuple<size:int,subtour:int[Citie
Datos internos (6)
Cities : range
dist : float[Edges]
Edges : {edge}
n : int
ordenInicial : int[Cities]
values : int[Edges]
Datos externos (2)
Propiedades
Propiedad
Valor
Problemas
Registro de scripts
Soluciones
Conflictos
Relajaciones
Registro del motor
Estadísticos
Perfilador
Watson Machine Learning
Estadística
Valor
Cplex
MIP
Objective 5,23666E3
Incumbent 5,24962E3
Nodes 282
Remaining nodes 86
Iterations 5106
Mejor nodo
Mejor entero
Solución de enteros
6E3
5E3
2
4
6
8
10
12
14
16
18
20
22
24
26
Tiempo de espera (segundos)
00:00:27:03