Trabajo Práctico

Facultad de Ingeniería, Universidad de Buenos Aires
[71.14] Modelos y Optimización I

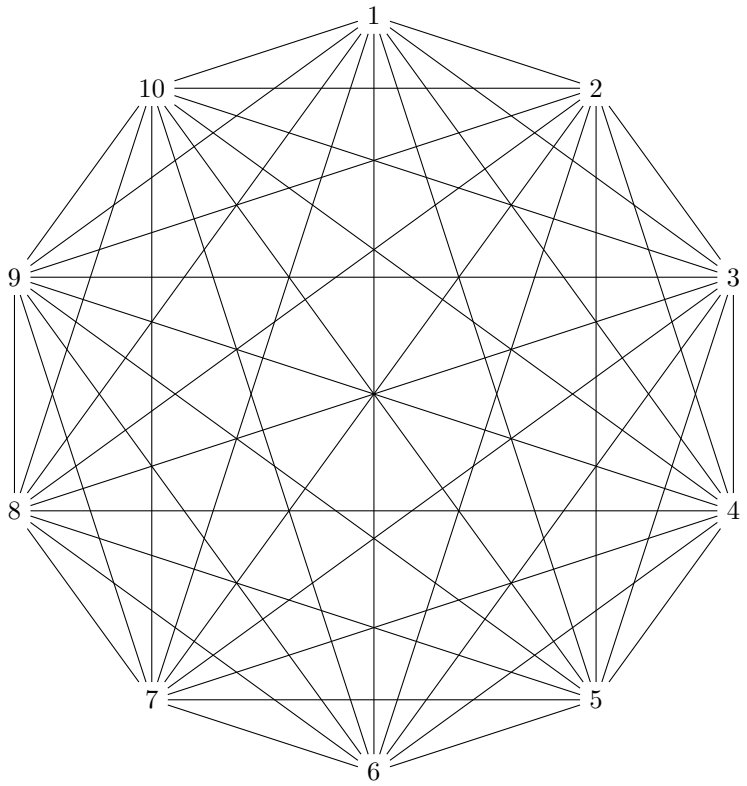
Federico del Mazo - 100029

02 de Mayo del 2022

Situacion problemática

Se tienen distintos bancos en el mapa a los cuales tengo que visitar exactamente una vez a cada uno, y en cada banco que visite tengo que recaudar/depositar dinero, siempre teniendo en cuenta que no puedo pasarme de ciertos límites.

Se trata de un problema de optimización parecido al del viajante. Por un lado, quiero que la distancia recorrida sea la mínima posible, y por el otro quiero siempre asegurarme que el dinero que recaude no se pase de un monto fijo.



Objetivo

Determinar el camino a hacer para minimizar la distancia, teniendo en cuenta que nunca podemos tener saldo negativo ni salgo que supere un monto definido, durante un recorrido del día.

Hipótesis y Supuestos

- Es negligible la distancia entre la central y el primer banco que visite (ya que no tengo el dato)
- Nunca me roban en todo el trayecto, ni pierdo dinero
- No hay limitantes no especificados
- Los montos preestablecidos no cambian
- Todas las distancias entre bancos son conocidas y fijas
- La ruta entre bancos es siempre la misma, sin variar su distancia
- El monto inicial es cero

Constantes

$CANT_{BANCOS} \in \mathbb{N}$: Cantidad de bancos a visitar.

$CAPACIDAD \in \mathbb{N}$: Cuanto dinero puede transportar el camión.

$DISTANCIA_{i,j} \in \mathbb{N}$: Distancia del banco i al banco j.

$DEMANDA_i \in \mathbb{N}$: Cuanto dinero entrega o recibe el banco i.

Variables de decisión controlables

$$BANCOS \in \{1...CANT_{BANCOS}\}$$

$Y_{i,j} \in \{0,1\}$: Variable bivalente que indica si el recorrido incluye ir del banco i al banco j.

$U_i \in \mathbb{N}$: Número de secuencia del banco i en el recorrido.

$P_i \in \mathbb{N}_0$: Plata que tiene el camión al llegar al punto i.

Adicionalmente, por fuera de las variables, M es una constante de valor muy grande y m es una constante de valor muy pequeño.

Vinculaciones y Restricciones

Solo se visita una vez cada banco.

$$\sum_{\substack{i=1 \\ j \neq i}}^{BANCOS} Y_{i,j} = 1 \quad \forall j \in BANCOS$$

$$\sum_{\substack{j=1 \\ j \neq i}}^{BANCOS} Y_{i,j} = 1 \quad \forall i \in BANCOS$$

Evitamos subtours.

$$\forall i, j \in BANCOS$$
$$U_i - U_j + CANT_{BANCOS} Y_{i,j} \leq CANT_{BANCOS} - 1$$

En todo momento debo tener dinero (esto es redundante por la condición de no-negatividad pero lo escribimos igual) y este dinero no debe superar la capacidad de mi camión.

$$\forall i \in BANCOS$$
$$0 \leq P_i \leq CAPACIDAD$$

Vinculo la variable de dinero con la diferencia de demandas entre cada par de bancos.

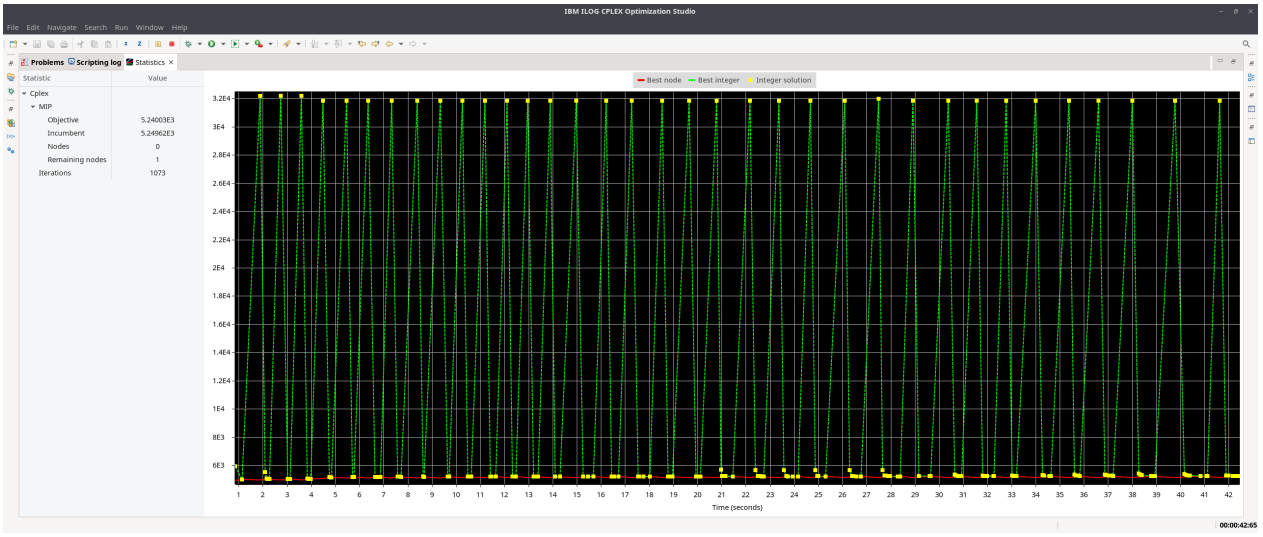
$$\forall i, j \in BANCOS$$
$$-M * (1 - Y_{i,j}) + DEMANDA_j \leq P_j - P_i \leq DEMANDA_i + M * (1 - Y_{i,j})$$

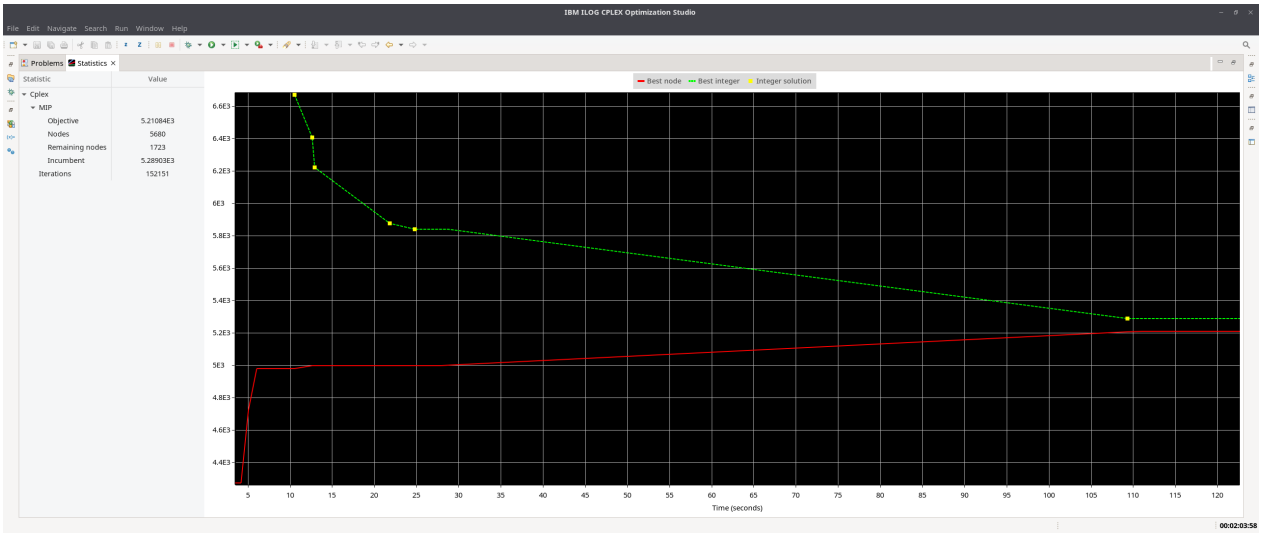
Función Objetivo

$$Min\ Z = \sum_{\substack{i=1 \\ i \neq j}}^{BANCOS} \sum_{j=1} Y_{i,j} * DISTANCIA_{i,j}$$

CPLEX

Primero se corren ambos modelos sin ningún tipo de modificación





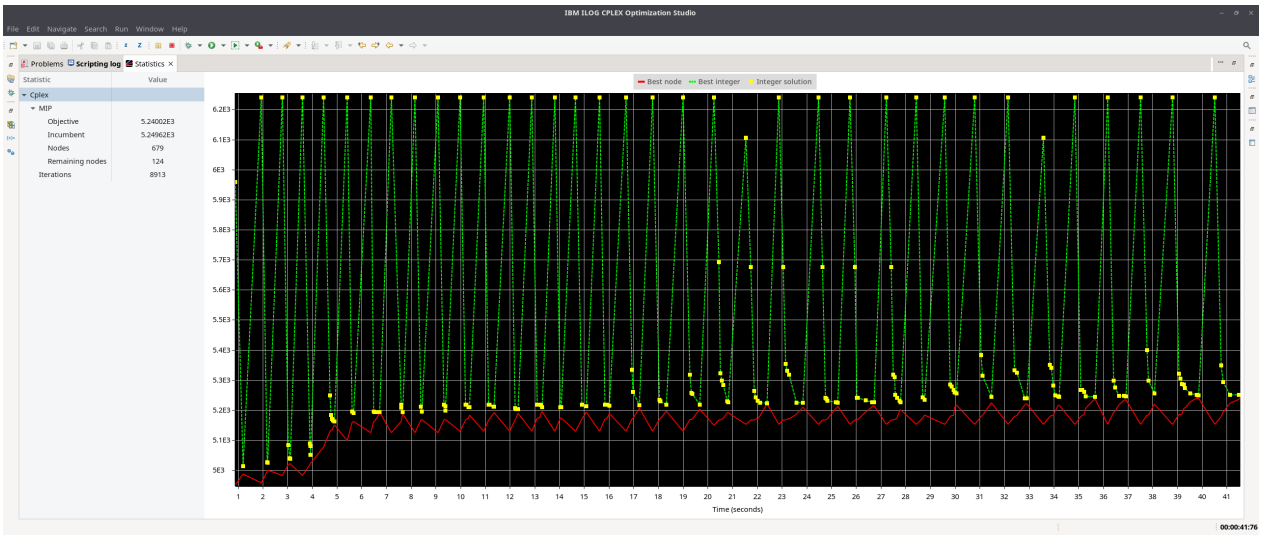
Luego, se modifica el código de CPLEX para insertar una solución inicial desde la cual ambos modelos parten, la cual proviene de correr el código de `main.py`

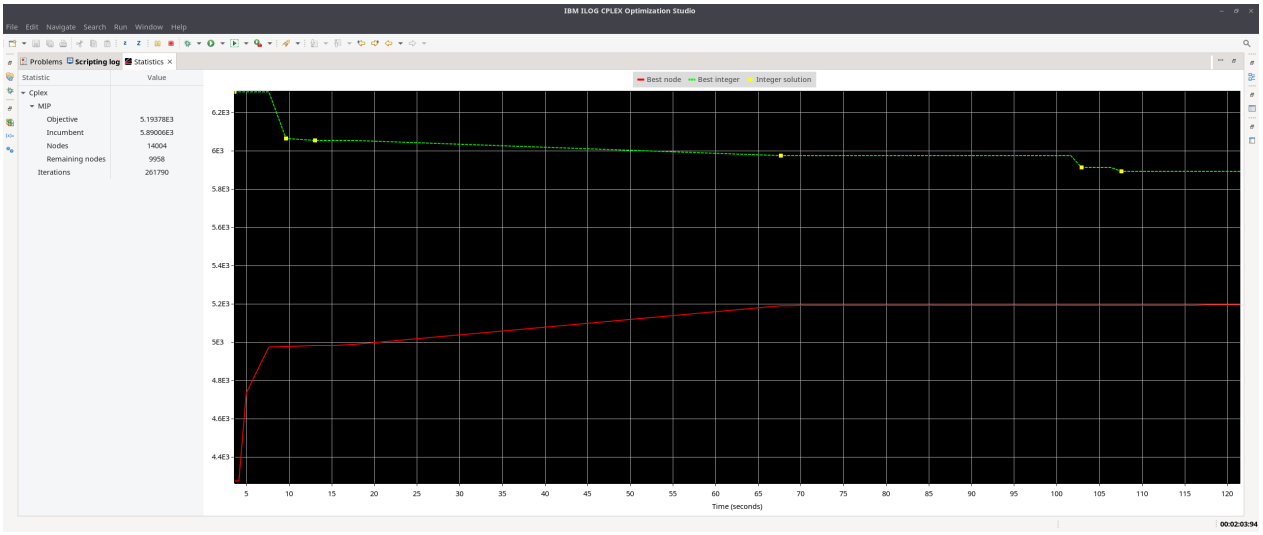
```
[1, 98, 87, 76, 73, 48, 63, 30, 84, 7, 8, 89, 96, 35, 93, 52, 33, 92, 54, 46, 90, 56,
↪  26, 75, 18, 85, 65, 55, 58, 50, 70, 86, 29, 81, 25, 20, 51, 43, 67, 32, 23, 38, 77,
↪  14, 80, 15, 78, 59, 16, 79, 88, 94, 10, 3, 62, 22, 4, 45, 71, 44, 64, 72, 49, 31,
↪  27, 41, 57, 39, 60, 66, 17, 11, 61, 36, 69, 24, 12, 53, 40, 42, 9, 28, 6, 37, 2,
↪  19, 99, 47, 83, 97, 100, 5, 95, 82, 34, 21, 68, 91, 13, 74]
```

Esta solución se inserta en ambos modelos

```
int ordenInicial[Cities] = [1, 98, 87, 76, 73, 48, 63, 30, 84, 7, 8, 89, 96, 35, 93,
↪  52, 33, 92, 54, 46, 90, 56, 26, 75, 18, 85, 65, 55, 58, 50, 70, 86, 29, 81, 25, 20,
↪  51, 43, 67, 32, 23, 38, 77, 14, 80, 15, 78, 59, 16, 79, 88, 94, 10, 3, 62, 22, 4,
↪  45, 71, 44, 64, 72, 49, 31, 27, 41, 57, 39, 60, 66, 17, 11, 61, 36, 69, 24, 12, 53,
↪  40, 42, 9, 28, 6, 37, 2, 19, 99, 47, 83, 97, 100, 5, 95, 82, 34, 21, 68, 91, 13,
↪  74];
int values[Edges];
execute {
  for ( var e in Edges) {
    values[e] = 0;
  }
  var ciudadAnterior = ordenInicial[n];
  for ( var i in Cities) {
    var ciudad = ordenInicial[i];
    if (ciudadAnterior < ciudad) {
      values[Edges.find(ciudadAnterior, ciudad)] = 1;
    } else {
      values[Edges.find(ciudad, ciudadAnterior)] = 1;
    }
    ciudadAnterior = ciudad;
  }
}
```

```
int ordenInicial[cities] = [1, 98, 87, 76, 73, 48, 63, 30, 84, 7, 8, 89, 96, 35, 93,
↪  52, 33, 92, 54, 46, 90, 56, 26, 75, 18, 85, 65, 55, 58, 50, 70, 86, 29, 81, 25, 20,
↪  51, 43, 67, 32, 23, 38, 77, 14, 80, 15, 78, 59, 16, 79, 88, 94, 10, 3, 62, 22, 4,
↪  45, 71, 44, 64, 72, 49, 31, 27, 41, 57, 39, 60, 66, 17, 11, 61, 36, 69, 24, 12, 53,
↪  40, 42, 9, 28, 6, 37, 2, 19, 99, 47, 83, 97, 100, 5, 95, 82, 34, 21, 68, 91, 13,
↪  74];
int values[edges];
execute {
  for ( var e in edges) {
    values[e] = 0;
  }
  var ciudadAnterior = ordenInicial[n];
  for ( var i in cities) {
    var ciudad = ordenInicial[i];
    values[edges.find(ciudadAnterior, ciudad)] = 1;
    ciudadAnterior = ciudad;
  }
}
```





Ambos modelos deben agregar restricciones para la eliminación de subtours.

Las restricciones añadidas en `TSP_MTZ` son las vistas en clase

```
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;
```

$$U_i - U_j + nY_{i,j} \leq n - 1 \quad \forall i, j \in \{1...n\} \quad i \neq j$$

Por otro lado, `TSP_subtours` es un modelo iterativo que agrega restricciones para que en cada iteración actual se evite entrar en los subtours encontrados en la iteración anterior. El ciclo de iteraciones termina cuando todos los subtours se eliminan.

```
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;
```

Ideas de la primera entrega

- Vamos a solucionarlo de manera greedy. Armo una matriz de distancias que me diga cuan alejado estoy de cada ciudad. Partiendo de la ciudad de origen voy a la que sea mas cercana, de ahí a la más cercana, etc.
- Como todavía el problema es de pocos elementos, puedo darme el lujo de no elegir ciudad de origen. Simplemente hago fuerza bruta. Parto de todas las ciudades (con demanda positiva), aplico mi solucionador de tsp greedy, y después me quedo con el que mejor (menor) score tenga.
- El único cambio introducido es que me confundi y no permití tener demanda = 0 ni demanda = capacidad, algo que en el problema si se puede. Ironicamente tuve peor resultado con este cambio, pero lo mantengo porque si no el modelo no sería correcto.

Ideas de la segunda entrega

- El problema de la solución actual no es tanto el algoritmo para una ciudad en particular, si no la elección de la ciudad de origen. Lo que haciamos en la primera instancia del problema era iterar todas las ciudades, correr el algoritmo para cada una, y elegir la que mejor score dé.
- La solución actual, en la instancia nueva, corre en 1 minuto y medio por cada ciudad de origen, lo cual no esta nada mal. Pero al ser casi 10 mil ciudades para probar, termina siendo una solución que corre en más de 200 horas en total, lo cual son casi 10 días de cómputo.
- Entonces, en vez de correr el algoritmo para todas las ciudades con fuerza bruta, simplemente lo corro para una, ya que la diferencia cambiando la ciudad de inicio es mínima (pensar que el algoritmo es determinístico para cada paso a tomar, entonces cambiar la ciudad de inicio solo podría, a lo sumo, minimizarnos un viaje en todo el recorrido).
- Para no volvernos locos y confirmar que el programa esta corriendo, agregamos la libreria `tqdm` que nos proporciona una linda barra de progreso para ir trackeando el avance en cada paso.
- Una optimización que se intentó agregar es la de pre-filtrar las ciudades a las que ya se que no puedo ir desde mi ciudad actual, viendo la demanda de plata que traen. Sin embargo, después de varios intentos y corridas se descartó, porque el pre-filtrado terminó siendo más lento que el ordenamiento directo de las ciudades. De todas formas esto sería una optimización sobre el tiempo de ejecución, y no sobre la solución en si, por lo que no es necesaria.

Ideas de la tercera entrega

- Se modifica el código para dejar de considerar las demandas de cada sucursal, y solo resolver el problema del viajante.