El problema de la ubicación de un aeropuerto
Complejidad y Optimización
Escuela de Ingeniería de Sistemas y Computación
Universidad del Valle

- Gema Elizabeth Díaz Palacios, 0633270
- Dayana Guevara Méndez, 0644152
- Víctor Alberto Romero González, 0632725

1. MODELO

Se busca maximizar la distancia de Manhattan entre el aeropuerto y la ciudad más cercana, dado un conjunto N de variables ubicadas en un plano.

2. VARIABLES DE DECISIÓN

X_a: Coordenada en el eje X del aeropuerto.
Y_a: Coordenada en el eje Y del aeropuerto.
D_{X_i}: Distancia en X entre el aeropuerto y una ciudad i, corresponde a $|X_i - X_a|$
D_{Y_i}: Distancia en Y entre el aeropuerto y una ciudad i, corresponde a $|Y_i - Y_a|$
D_{X_c}: Distancia en X del aeropuerto a la ciudad más cercana
D_{X_c}: Distancia en Y del aeropuerto a la ciudad más cercana

BX_i, BY_i: Variables binarias usadas para modelar las restricciones de valor absoluto.

$$BX_i : \begin{Bmatrix} 1 \; si \; X_i \geq X_a \\ 0 \; si \; X_i \leq X_a \end{Bmatrix}$$

$$BY_i : \begin{Bmatrix} 1 \; si \; Y_i \geq Y_a \\ 0 \; si \; Y_i \leq Y_a \end{Bmatrix}$$

3. FUNCIÓN OBJETIVO

$$\max \ z = D_{X_c} + D_{Y_c}$$

4. RESTRICCIONES

- ✓ Distancia Manhattan entre el aeropuerto y una ciudad i.

$$D_{X_i} \geq X_i - X_a$$
$$D_{X_i} \geq X_a - X_i$$
$$D_{X_i} \leq X_i - X_a + M(1 - BX_i)$$
$$D_{X_i} \leq X_a - X_i + MBX_i$$

$$D_{Y_i} \geq Y_i - Y_a$$
$$D_{Y_i} \geq Y_a - Y_i$$
$$D_{Y_i} \leq Y_i - Y_a + M(1 - BY_i)$$
$$D_{Y_i} \leq Y_a - Y_i + MBXY_i$$

Para garantizar la satisfabilidad del modelo se asigna a M un valor lo suficientemente grande. Inicialmente se escogió $M = n \times n$, pero este alteraba el valor de las restricciones. Por lo que se decidió escoger $M = 1000$.
Además cada distancia tiene asociada una variable binaria BX_i y BY_i que asegura que las distancias sean positivas.

- ✓ Restricción que garantiza que el aeropuerto no esté ubicado en una de las n ciudades.

$$D_{X_c} \geq 1$$
$$D_{Y_c} \geq 1$$

- ✓ Restricción que garantiza que $D_{X_c} + D_{Y_c}$ sea la distancia más cercana del aeropuerto a cada una de las ciudades.

$$D_{X_c} + D_{Y_c} \leq D_{X_i} + D_{Y_i}$$

- ✓ Restricción que garantiza que el aeropuerto este ubicado dentro de la región.

$$X_a \leq n$$
$$Y_a \leq n$$

- ✓ Restricción para garantizar que las variables BX_i y BY_i sean binarias.

$$0 \leq BX_i \leq 1$$

$$0 \leq BY_i \leq 1$$

✓ Restricciones obvias.

$$X_a \geq 0$$
$$Y_a \geq 0$$
$$D_{X_c} \geq 0$$
$$D_{Y_c} \geq 0$$
$$D_{X_i} \geq 0$$
$$D_{Y_i} \geq 0$$
$$BX_i \geq 0$$
$$BY_i \geq 0$$

5. FORMA ESTÁNDAR

$$D_{X_i} + X_a - A_j = X_i$$
$$-D_{X_i} + X_a - A_{j+1} = X_i$$
$$D_{X_i} + X_a + MBX_i + A_{j+2} = X_i + M$$
$$-D_{X_i} + X_a + MBX_i - A_{j+3} = X_i$$

$$D_{Y_i} + Y_a - A_{j+4} = Y_i$$
$$-D_{Y_i} + Y_a - A_{j+5} = Y_i$$
$$D_{Y_i} + Y_a + MBY_i + A_{j+6} = Y_i + M$$
$$-D_{Y_i} + Y_a + MBY_i - A_{j+7} = Y_i$$

$$D_{X_c} + A_{j+8} = 1$$
$$D_{Y_c} + A_{j+9} = 1$$

$$D_{X_c} + D_{Y_c} - D_{X_i} - D_{Y_i} + A_{j+10}$$

$$X_a + A_{j+11} \geq n$$

$$Y_a + A_{j+12} \geq n$$

$$Y_a + A_{j+12} \geq n$$

$$BX_i + A_{j+13} \geq n$$

$$BY_i + A_{j+13} \geq n$$

$$M = 1000$$

DETALLES DE IMPLEMENTACIÓN

Para el desarrollo del proyecto se utilizó la librería

LPSolve integrado con **Java.**

El programa consta de seis archivos .java:

- **GUI.java:** En esta clase está implementada la Interfaz Gráfica que permite cargar un archivo con el formato especificado en el enunciado del proyecto y muestra las coordenadas de las ciudades. Con el botón ejecutar se visualiza la solución óptima obtenida por el simplex.

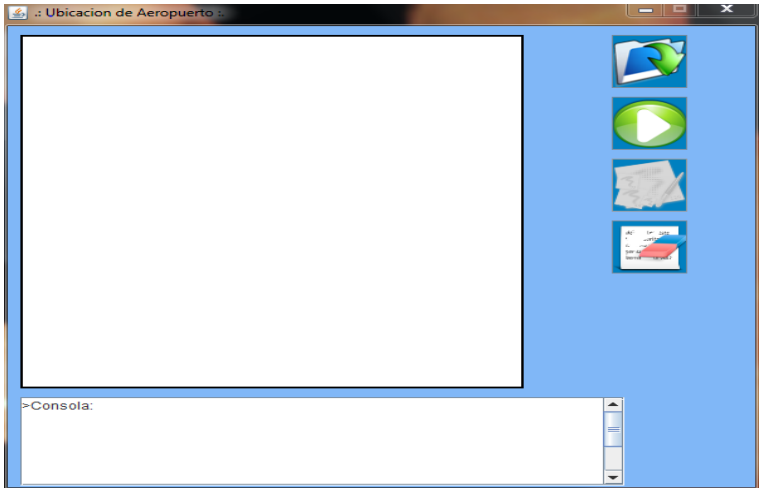


Figura 1. Interfaz gráfica

- **LeerArchivo.java:** En esta clase se lee el archivo con el formato especificado y se almacena en la variable `ubicacionCiudades[][]`, que permitirá su manipulación para encontrar la solución óptima.

- **MiCanvas.java:** Esta clase permite dibujar las coordenadas de las ciudades y la coordenada óptima para la ubicación del aeropuerto.

- **ProblemaOptimizar.java:** La clase consta de siete atributos, los cuales se describen brevemente a continuación:

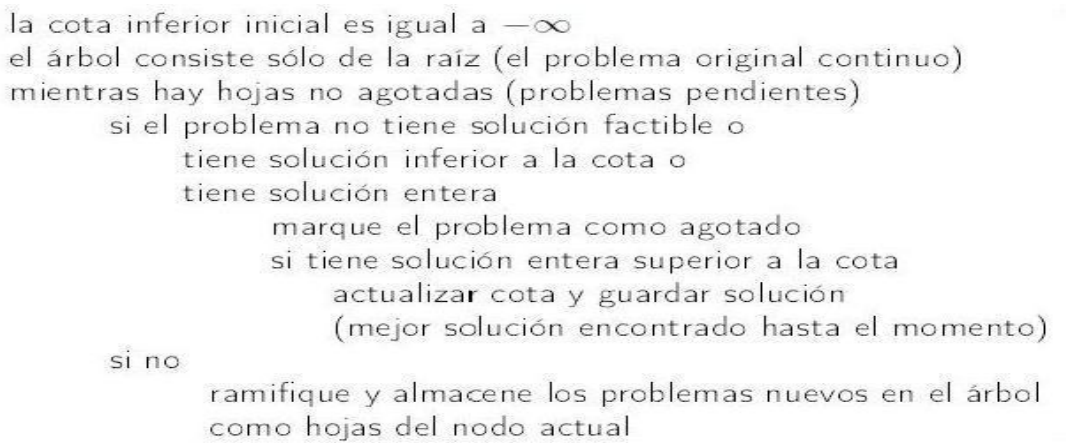
    - `M`: Atributo tipo `double` que almacena un valor lo suficientemente grande para que las restricciones sigan siendo satisfactibles.
    - `tamaño`: Atributo tipo `double` que guarda la cantidad de ciudades con las que se va a construir el modelo.
    - `rightHand`: Atributo tipo `double` que almacena el valor del coeficiente al lado derecho de la desigualdad.
    - `N`: Atributo tipo `int` que contiene el tamaño de la región.
    - `ubicacionCiudades[][]`: Atributo tipo `double` que guarda la ubicación de las ciudades leída desde el archivo.

- `nuevasRestricciones`: Atributo tipo `Vector` que contiene en cada una de sus posiciones un `Vector`, es decir un vector de vectores donde se almacenan las nuevas restricciones que surgen cuando una de las variables no es entera.
- `vectorVariablesBinarias`: Atributo tipo `Vector` que contiene en cada una de sus posiciones un `Vector`, donde se almacenan las variables binarias por cada una de las ciudades.

Además, contiene tres métodos:

- `ProblemaOptimizar()`: Método constructor de la clase que inicializa los vectores `nuevasRestricciones` y `vectorVariables Binarias` creando un vector tipo `Double` en cada posición. Además inicializa la variable `M` asignándole el valor de `1000.`
- `execute(double ubicacionCiudades[][],int N)` :Método tipo `int` que recibe como parámetros las coordenadas de las ciudades y el tamaño de la región, donde se adicionan las restricciones al `LPSolve` y se obtienen las variables binarias y su valor. Retorna `0` si se encontró una solución óptima y `1` sino.
- `executeConRestricciones(double ubicacionCiudades [][],int N, Vector<Vector<Double>> nuevas Restricciones`: Método tipo `int` que recibe como parámetros la ubicación de las ciudades, el tamaño de la región y el vector de vectores que contiene las restricciones que surgen cuando una variable no es entera. Se adicionan estás nuevas restricciones al LPSolve y se calcula una solución. Retorna `0` si se encontró una solución óptima y `1` sino.

- **BranchAndBound.java:** En esta clase se implementa el algoritmo Branch & Bound aprendido en el curso.



La ramificación del árbol se realizo haciendo búsqueda en profundidad, para ello se utilizo la estructura de dato conocida como “pila” representándose de esta forma la búsqueda del optimo entero.

Cada dato dentro de la pila es un nodo que contiene las nuevas restricciones con las que se debe ejecutar el simplex para obtener el óptimo con las variables binarias enteras.

El algoritmo termina cuando no hay ningún elemento dentro de la pila, esto quiere decir que el árbol tiene todas sus hojas agotadas.

Nota: La implementación entregada funciona gracias al método `setBinary()`, de la librería `LPSolve`, debido a que cuando se realizaba el recorrido del árbol las variables binarias nunca adquirían valores de `0` ó `1`.

PRUEBAS REALIZADAS

Las pruebas se realizaron aumentando la cantidad de ciudades para observar el aumento de restricciones, variables y tiempo de ejecución.

ANÁLISIS DE LOS RESULTADOS

N (tamaño de la región)	Cantidad de ciudades	Número de restricciones	Número de variables
12	10	136	44
20	20	266	84
20	25	331	104
180	50	656	204
250	100	1306	404
250	200	2606	804

Tabla 1. Número de veredas vs. Duración

Con la tabla anterior se construyeron las siguientes gráficas:

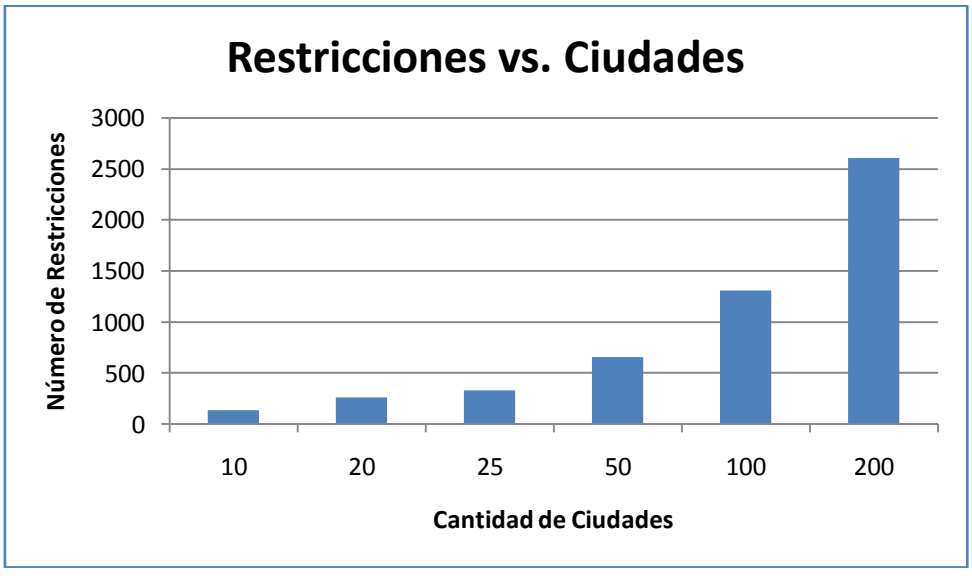


Gráfico 1. Restricciones vs. Ciudades.

En el Grafico 1 se observa que existe una relación proporcional entre el número de ciudades y el número de restricciones, esto se debe a que por cada ciudad aumenta el número de variables definidas en el modelo.

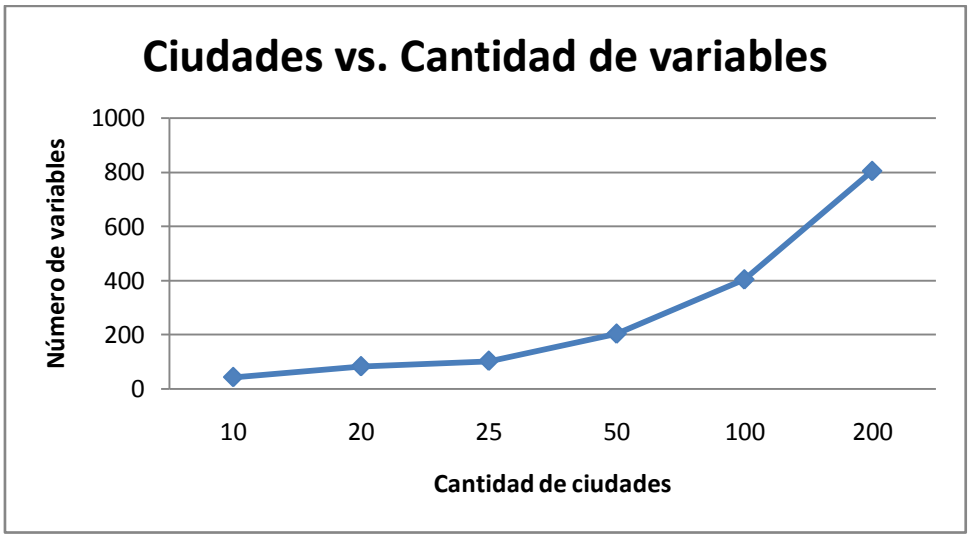


Gráfico 2. Número de varibales vs. Ciudades.

En la gráfica 2, se observa como a medida que el número de ciudades crece las variables se aumentan significativamente.

CONCLUSIONES

- El tiempo de ejecución depende del tamaño de la instancia, pues éste crece proporcionalmente con el número de ciudades que se adicionen al problema, agregándose nuevas restricciones y variables binarias. El aumento de estas últimas se ve reflejado en el tiempo de ejecución ya que el árbol al aplicar el Branch and Bound se ramifica dependiendo de la cantidad de estas variables.

- El diseño de un modelo requiere de habilidades y practica para formular problemas por parte de las personas que los abordan. Se menciona esto, porque a pesar de que existen pasos para diseñar un modelo, la metodología para su creación no siempre es la misma y puede variar mucho de un caso a otro.

- Una debilidad del programa, es el incorrecto funcionamiento del Branch & Bound, dado que no se presenta la solución óptima, pues el algoritmo se queda estancado en un ciclo infinito cuando se está recorriendo el árbol, error que hasta el momento de terminación del informe no fue posible resolver, es por eso que se utilizó el método `setBinary()` de la librería `LPSolve`, el problema radica en que no se está implementando correctamente la correspondencia recursiva de las variables no enteras.

- Aunque parezca fácil la implementación de la solución del problema, debemos tener en cuenta que sin las adecuadas restricciones, es decir, sin el modelo apropiado encontrar una solución óptima satisfactible es casi imposible. Esto nos permite concluir que el análisis es muy importante antes de programar, pensar en una solución o implementar la solución no es suficiente, se necesita analizar sus posibles casos para mejorar en tiempo, espacio y eficiencia.