Notas de curso
Laboratorio de Optimización

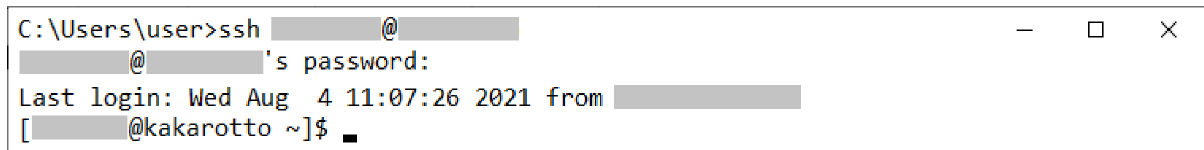
Rodrigo Alexander Castro Campos
UAM Azcapotzalco, División de CBI
`https://racc.mx`
`https://omegaup.com/profile/rcc`

Última revisión: 16 de julio de 2022

1. Acceso al servidor de trabajo y comandos básicos

El Posgrado en Optimización cuenta con servidores con acceso a internet y los cuales permanecen funcionando las 24 horas del día. Los alumnos del Posgrado pueden hacer uso de dichos servidores para ejecutar programas y realizar experimentos computacionales. Los alumnos también pueden solicitar que se instale software especializado que esté disponible gratuitamente para uso académico. Entre el software actualmente instalado se encuentra el solucionador de programación lineal Gurobi, la biblioteca de visión por computadora OpenCV, la biblioteca de geometría computaconal CGAL, entre otros.

El sistema operativo actualmente instalado en los servidores es Fedora 34. Los alumnos pueden conectarse remotamente mediante el protocolo *Secure Shell* (abreviado como SSH). La dirección IP del servidor asignado y las credenciales de acceso (usuario y contraseña) se proporcionan directamente a los alumnos inscritos. La sesión SSH se puede iniciar desde una terminal o línea de comandos que soporte el protocolo. El comando de ingreso es `ssh usuario@ip` y, una vez establecida la conexión con el servidor, el usuario deberá ingresar su contraseña. La contraseña que el usuario teclea no se mostrará en la terminal.



Ejemplo de ingreso al servidor usando la línea de comandos de Windows 10.

El directorio inicial de trabajo es `/home/`*`usuario`* y se abrevia como ~ en la terminal. A continuación se describen algunos de los comandos más comunes en Linux.

Comando	**Acción realizada**
`passwd`	Permite modificar la contraseña del usuario.
`ls`	Lista el contenido del directorio de trabajo.
`cat` *`archivo`*	Lista el contenido del archivo.
`cd` *`carpeta`*	Entra al directorio indicado y lo establece como el directorio de trabajo. La cadena `..` denota el directorio superior.
`gcc` *`código`* `-o` *`ejecutable`*	Compila el código en lenguaje C del archivo indicado, creando un ejecutable con el nombre dado.
`./`*`ejecutable`*	Ejecuta el programa indicado del directorio de trabajo.
`nohup` `comando` *`&`*	Ejecuta el comando dado y lo desvincula de la terminal. El comando continuará su ejecución aunque la terminal se cierre.
`clear`	Limpiar el contenido de la terminal.
`exit`	Cierra la sesión de la terminal actual.

Adicionalmente, la terminal permite redirigir tanto la entrada como la salida estándar de un programa. El comando `./ejecutable < archivo` hará que la entrada estándar se lea del archivo indicado, mientras que el comando `./ejecutable > archivo` hará que la salida se escriba en el archivo indicado. El comando `./ejecutable < entrada > salida` permite combinar ambas redirecciones. Además, el comando `./ejecutable 2> archivo` permite redirigir la salida de errores.

El compilador `gcc` no optimiza por defecto y podemos usar la bandera `-O3` para solicitarle que lo haga. A su vez, `gcc` suele generar instrucciones de un subconjunto muy general y ampliamente soportado de instrucciones, pero se puede usar la bandera `-march=native` para indicarle que puede generar instrucciones especializadas para la computadora actual. El compilador `g++` compila códigos escritos en C++ y se puede indicar la versión del lenguaje con la bandera `-std=c++NN` donde `NN` son los dos últimos dígitos del año de la versión. Por ejemplo, podemos compilar el archivo `programa.cpp` escrito en C++ 2020 con optimización y generando instrucciones especializadas mediante el siguiente comando:

```
g++ -std=c++20 -O3 -march=native programa.cpp -o programa
```

Los ejecutables en Linux no suelen llevar extensión, pero funcionarán correctamente aún con una.

La transferencia de archivos entre el servidor y el equipo local se puede hacer mediante el protocolo *Secure File Transfer Protocol* (abreviado como SFTP). El comando para ingresar al servidor usado este procolo es `sftp usuario@ip`. Habiendo ingresado, los comandos `ls` y `cd` sirven para navegar en el sistema de archivos del servidor, mientras que los comandos `lls` y `lcd` sirven para hacer lo mismo pero en el equipo local. Finalmente, el comando `put archivo` transfiere un archivo del directorio de trabajo local al directorio de trabajo del servidor, mientras que el comando `get archivo` realiza el proceso inverso.

2. Modelado de problemas con programación lineal

Muchos problemas combinatorios y de optimización pueden modelarse mediante *programas lineales*. Un programa lineal consiste en un conjunto de desigualdades lineales que restringen el conjunto de soluciones factibles del problema, así como una función objetivo lineal que determina el valor de las soluciones factibles y cuyo valor se desea minimizar o maximizar.

Las variables de decisión de un programa lineal son las variables que aparecen en los polinomios de dicho programa, pueden ser de distintos tipos (reales, enteras, binarias, etc) y sus valores describen una solución en particular. El término "programación lineal" se suele emplear cuando sólo se usan variables continuas, mientras que los términos "programación lineal entera" y "programación lineal entera mixta" se emplean cuando todas o sólo algunas variables son discretas, respectivamente. A continuación se muestran dos problemas computacionales y se propone un modelo lineal para cada uno de ellos.

Problema: El problema de la mochila (versión discreta).
Entrada: Dos naturales n, c y dos secuencias p, v de n enteros positivos cada una. El entero c denota la capacidad en peso de una mochila, mientras que p, v denotan los pesos y valores de n objetos.
Salida: Un natural r que sea el valor máximo de $\sum_{i \in s}(v_i)$ sujeto a $\sum_{i \in s}(p_i) \leq c$, donde s es una subsecuencia de los índices de 0 a $n-1$ que denota la selección de objetos a meter en la mochila.

Un modelo de programación lineal no permite expresar explícitamente conjuntos o subsecuencias, pero es posible usar variables enteras binarias (es decir, limitadas a tomar valores 0 o 1) para expresar si un objeto se incluirá o no en una selección. Podremos modelar este problema usando variables x_i para $0 \leq i < n$ donde $x_i = 1$ si la solución mete el i-ésimo objeto en la mochila y $x_i = 0$ en caso contrario.

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} v_i x_i \\ \text{sujeto a} \quad & \sum_{i=0}^{n-1} p_i x_i \leq c \\ & x_i \in \mathbb{Z}_2 \qquad \text{para } 0 \leq i < n \end{aligned}$$

Problema: Cálculo de cuadrados mágicos.
Entrada: Un natural n.
Salida: Una matriz de tamaño $n \times n$ cuyas celdas contengan los enteros del 1 al n^2 sin repetir y donde todas las filas, columnas y ambas diagonales sumen lo mismo.

El problema anterior no es de optimización, pero de todos modos se puede plantear un conjunto de restricciones lineales que determinen el conjunto de las soluciones factibles. Usaremos variables binarias $x_{i,j,k}$ para $0 \leq i, j < n$ donde $x_{i,j,k} = 1$ si y sólo si el entero k se anotó en la celda (i, j). El modelo lineal necesita variables familias de restricciones:

- Restricciones que obliguen a que cada celda contenga exactamente un número.

$$\sum_{k=1}^{n^2} x_{i,j,k} = 1 \qquad \text{para } 0 \leq i, j < n$$

- Restricciones que obliguen a que cada número se use exactamente una vez.

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_{i,j,k} = 1 \qquad \text{para } 1 \leq k \leq n^2$$

- Restricciones que obliguen a que cada fila, columna y diagonal sumen lo mismo. Por facilidad, podemos definir variables enteras auxiliares $y_{i,j} = \sum_{k=1}^{n^2} k x_{i,j,k}$ para denotar el valor que está anotado en cada celda. También podemos usar una variable entera auxiliar s para denotar el valor sumado.

$$\sum_{j=0}^{n-1} y_{i,j} = s \qquad \text{para } 0 \leq i < n$$

$$\sum_{i=0}^{n-1} y_{i,j} = s \qquad \text{para } 0 \leq j < n$$

$$\sum_{i=0}^{n-1} y_{i,i} = s$$

$$\sum_{i=0}^{n-1} y_{i,n-1-i} = s$$

3. El solucionador Gurobi y modelos en formato LP

El software Gurobi[1] es un solucionador de programación lineal, lineal entera y cuadrática. Gurobi toma como entrada un modelo lineal y es capaz de de determinar si existen soluciones factibles y, en caso afirmativo, de encontrar la solución óptima o de determinar que dicho óptimo está indefinido. Para resolver el modelo, Gurobi emplea algoritmos de tiempo polinomial cuando todas sus variables son continuas y emplea algoritmos de búsqueda con retroceso cuando alguna variable es discreta.

Gurobi es de uso gratuito para fines académicos y se encuentra disponible en los servidores del Posgrado en Optimización. Aquéllos que deseen instalar Gurobi en sus propios equipos pueden seguir la guía de inicio disponible en `https://www.gurobi.com/documentation/quickstart.html`, donde uno de los pasos es solicitar y activar una licencia académica; ello requiere que el equipo esté conectado a internet usando la red de una institución académica. Una vez instalado, Gurobi puede ejecutarse desde la terminal mediante el comando `gurobi_cl` *`archivo`* donde el archivo debe tener extensión `.lp` y debe contener un modelo escrito en el formato LP que es de uso específico. Aunque este formato no está estandarizado, es de uso común en solucionadores lineales. La variante del formato LP que Gurobi reconoce está documentada

[1] `https://www.gurobi.com/`

en https://www.gurobi.com/documentation/9.0/refman/lp_format.html y contiene algunas características que no se explicarán en este momento, ya que Gurobi proporciona mejores formas de expresar y resolver modelos. A continuación se presenta un modelo lineal que reescribiremos en formato LP.

$$
\begin{aligned}
\text{maximizar} \quad & 5x + y - w - z + 4 \\
\text{sujeto a} \quad & x \leq -3 \\
& -3 \leq w \leq -1 \\
& 0 \leq y \leq 8 \\
& x + y + 1 \geq x - y \\
& x \in \mathbb{R}, w, y \in \mathbb{Z}, z \in \mathbb{Z}_2
\end{aligned}
$$

En el formato LP, el sentido de la optimización se especifica con `minimize` o `maximize`. La línea `subject to` inicia la sección que incluye las restricciones del modelo. Todas las variables del modelo son no negativas por defecto, pero se puede indicar cuáles de ellas son libres mediante la palabra `free` dentro de la sección `bounds`. A su vez, todas las variables del modelo son continuas por defecto, pero se puede indicar cuáles de ellas son enteras o binarias mediante las palabras `generals` y `binaries`, respectivamente. El símbolo \ inicia un comentario de línea y es opcional terminar el modelo con la palabra `end`. Antes de reescribir el modelo, listaremos algunas limitaciones de Gurobi con respecto al formato LP.

- La función objetivo no puede contener términos constantes, ya que el solucionador los interpreta como variables con un nombre numérico. El sumando +4 del modelo anterior debe excluirse de la reescritura.
- La multiplicación de un coeficiente por una variable debe escribirse con un espacio entre ambos. Por ejemplo, escribir `5 x` es correcto, mientras que escribir `5x` no lo es.
- Los operadores binarios de suma y resta deben aparecer separados por espacios de sus operandos. Por ejemplo, escribir `y - z` es correcto, mientras que escribir `y-z` no lo es.
- Una restricción no puede contener términos constantes del lado izquierdo de la desigualdad y no puede contener variables del lado derecho. Además, el lado derecho debe contener exactamente un término constante. Esto nos obligará a despejar desigualdades y a evaluar manualmente términos constantes.

El equivalente en formato LP del modelo anterior es el siguiente:

Código

```
maximize
   5 x + y - w - z      \ excluímos el término constante
subject to
   x <= -3
   w >= -3
   w <= -1
   y <= 8
   2 y >= -1
bounds
   x free
   w free
generals
   y w
binaries
   z
end
```

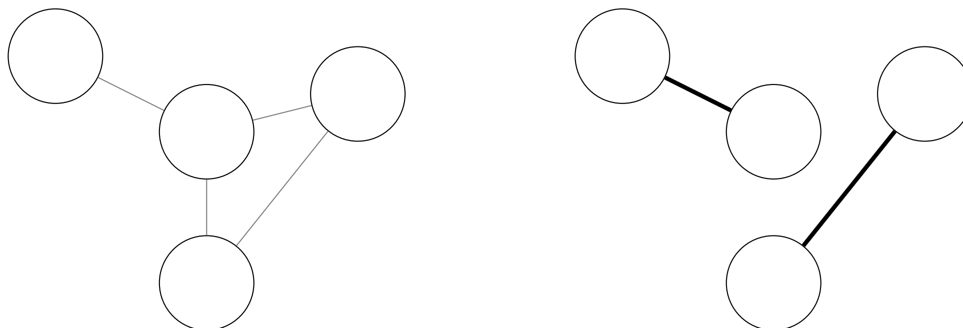
Si escribimos el modelo anterior en un archivo llamado `ejemplo.lp`, podremos resolverlo mediante el comando `gurobi_cl ejemplo.lp`. En este caso, Gurobi nos dirá que sí existe solución y que el valor óptimo es -4, pero no imprimirá el valor que deben tomar las variables para llegar a dicho óptimo. Podemos obtener esta última información mediante el siguiente comando:

`gurobi_cl ResultFile=ejemplo.sol ejemplo.lp`

Con ello, Gurobi escribirá la solución en el archivo indicado. El uso de la extensión `.sol` debe respetarse.

3.1. Ejercicios

1. Un acoplamiento de una gráfica es un conjunto de aristas sin vértices en común.



Ejemplo de gráfica y su acoplamiento de cardinalidad máxima.

Modela el problema de calcular un acoplamiento de cardinalidad máxima mediante un programa lineal y escribe un programa que lea la descripción de una gráfica y produzca el modelo en formato LP. Puedes suponer que la gráfica será descrita mediante dos enteros n y m seguidos de m parejas de enteros, donde n denota la cantidad de vértices de la gráfica, m denota la cantidad de aristas y las m parejas denotan los extremos de cada arista. Los vértices estarán numerados de 0 a $n-1$.

4. Parámetros de interacción de `gurobi_cl`

Gurobi cuenta con decenas de parámetros de configuración que pueden establecerse al momento de ejecutar `gurobi_cl` y que están documentados en `https://www.gurobi.com/documentation/9.0/refman/parameters.html`. Los parámetros más básicos son aquéllos que modifican la interacción que ocurre con el usuario y a continuación se describen los más importantes.

- `TimeLimit=`*`real`*
  Limita el tiempo de ejecución a la cantidad dada en segundos. Por omisión, no hay tiempo límite.

- `ResultFile=`*`archivo`*
  Al terminar la ejecución, escribe en el archivo la mejor solución encontrada. Si el archivo tiene extensión `.sol` se incluirá el valor de la función objetivo y los valores de todas las variables. Si el archivo tiene extensión `.mst` sólo se incluirán los valores de las variables enteras. En dependencia del valor de `IntFeasTol`, Gurobi puede escribir un valor real para denotar el valor de una variable entera.

- `SolFiles=`*`carpeta/archivo`*
  Escribe cada solución intermedia encontrada en `carpeta/archivo_`n`.sol`, donde n es un índice incremental que comienza en 0.

- `InputFile=`*`archivo`*
  Toma una asignación (parcial o completa) de valores para las variables como punto de partida para la resolución del modelo. El archivo puede tener extensión `.sol` o `.mst` y la asignación será potencialmente desechada si no puede producir una solución factible.

- `LogFile=`*`archivo`*
  Determina el archivo de bitácora de Gurobi. Por omisión, todas las ejecuciones de `gurobi_cl` se registran en `gurobi.log`. Una cadena vacía en este parámetro deshabilita la bitácora.

- `LogToConsole=0`
  Deshabilita la impresión de mensajes en consola.

- `OutputFlag=0`
  Deshabilita la impresión de mensajes en consola y también la escritura en la bitácora.

5. El proceso de optimización de Gurobi

El proceso de optimización de Gurobi consta de varias etapas, donde algunas son bastante más complicadas que otras. A continuación se describe cada una de ellas.

```
Academic license - for non-commercial use only

Gurobi Optimizer version 9.0.0 build v9.0.0rc2 (linux64)
Copyright (c) 2019, Gurobi Optimization, LLC

Read LP format model from file modelo.lp
Reading time = 0.00 seconds
: 4 rows, 7 columns, 7 nonzeros
Optimize a model with 4 rows, 7 columns and 7 nonzeros
Model fingerprint: 0xc9c45a1e
Model has 3 general constraints
Variable types: 7 continuous, 4 integer (2 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [0e+00, 0e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 8e+00]
Presolve removed 2 rows and 3 columns
Presolve time: 0.00s
Presolved: 2 rows, 4 columns, 4 nonzeros
Presolved model has 2 SOS constraint(s)
Variable types: 0 continuous, 4 integer (2 binary)

Root relaxation: objective 1.000000e+00, 2 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0    1.000    0     1          -    1.000      -      -    0s
H    0     0                      2.0000000    1.000  100.00%    -    0s
H    0     0                      1.0000000    1.000   0.00%     -    0s

Explored 1 nodes (2 simplex iterations) in 0.02 seconds
Thread count was 24 (of 24 available processors)

Solution count 1: 1

Optimal solution found (tolerance 1.00e-04)
Best objective 1.000000000000e+00, best bound 0.000000000000e+00, gap 0.0000%
```

Lectura de la entrada

Construcción de la representación matricial del modelo

Simplificación del modelo

Resolución de la relajación del modelo

Optimización entera con ramificación y acotamiento

Las etapas del proceso de optimización de Gurobi al resolver un modelo mixto.

1. Lectura de la entrada: En esta etapa, Gurobi abre y procesa el contenido de todos los archivos que el usuario haya indicado. Cuando se ejecuta Gurobi mediante `gurobi_cl`, esta etapa incluye la apertura del archivo con el modelo a resolver, así como su verificación léxica, sintáctica y semántica.

2. Construcción de la representación matricial del modelo: En esta etapa, Gurobi construye una matriz donde cada fila corresponde con una restricción y cada columna corresponde con una variable o un término constante. Si la matriz resultante es dispersa, entonces Gurobi usa una representación especializada para reducir su consumo de memoria. Es común que la matriz sea dispersa porque cada restricción suele mencionar un subconjunto pequeño de las variables del modelo.

3. Simplificación del modelo: En esta etapa, Gurobi intenta simplificar el modelo mediante dos estrategias principales. En la primera, Gurobi intenta determinar si alguna variable del modelo se puede eliminar o reemplazar por alguna combinación lineal de otras variables. En la segunda, Gurobi intenta

determinar qué restricciones del modelo son redundantes y pueden eliminarse. Esta etapa tiene un comportamiento iterativo: una simplificación del modelo suele evidenciar la posibilidad de aplicar simplificaciones adicionales. Ante un modelo sencillo, Gurobi incluso podría encontrar la solución óptima durante la simplificación. Esta etapa suele ser razonablemente eficaz en su tarea de reducir el modelo, lo cual tiene el efecto de acelerar las etapas posteriores.

4. Resolución de la relajación del modelo: En esta etapa, Gurobi intentará calcular una solución óptima de la relajación del modelo, el cual se obtiene ignorando los requerimientos de integralidad de las variables del modelo original. Cuando la relajación no tiene solución o no está acotado, lo mismo ocurrirá con el modelo original (esto no es cierto en general, pero sí cuando todos coeficientes son racionales, como es el caso con Gurobi). Si el modelo original no tiene variables enteras, entonces la solución encontrada es la final. Además, algunas familias específicas de modelos tienen la propiedad de que la solución de su relajación es entera. Un modelo sin variables enteras se puede resolver en tiempo polinomial.

5. Optimización entera con ramificación y acotamiento: En esta etapa, Gurobi intenta encontrar la solución óptima del problema entero. Desafortunadamente, programación entera es un problema NP-Duro, por lo que no se conocen algoritmos subexponenciales para él. Gurobi emplea un algoritmo conocido como ramificación y acotamiento, el cual resulta ser razonablemente rápido para muchos problemas.

El funcionamiento del algoritmo de ramificación y acotamiento se describe a continuación mediante un ejemplo. Supongamos que queremos optimizar el siguiente modelo entero:

$$\begin{aligned} \text{maximizar} \quad & x + 5y \\ \text{sujeto a} \quad & x + y \leq 2.5 \\ & 3x \leq y \\ & x, y \in \{0, 1, 2, 3\} \end{aligned}$$

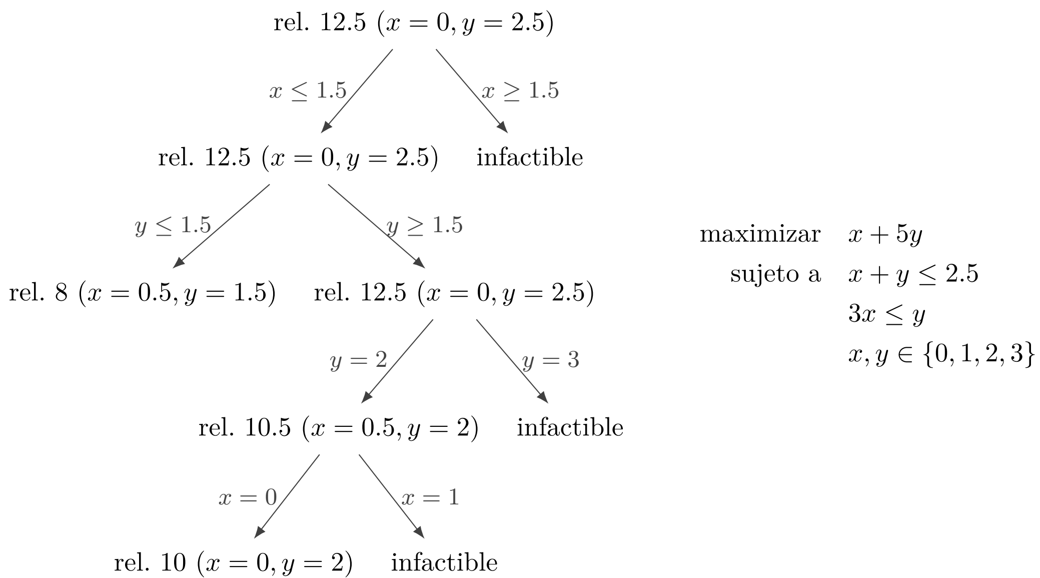
El algoritmo de ramificación y acotamiento construye un árbol de búsqueda de la siguiente forma. La raíz del árbol corresponde con el modelo relajado, mientras que los vértices hijos se construyen copiando el modelo y agregándole restricciones adicionales a cada copia. Por ejemplo, podemos bifurcar el árbol de búsqueda tomando una decisión parcial sobre la variable x de la siguiente forma:

$$\begin{aligned} \text{maximizar} \quad & x + 5y \\ \text{sujeto a} \quad & x + y \leq 2.5 \\ & 3x \leq y \\ & x \leq 1.5 \\ & x, y \in \{0, 1, 2, 3\} \end{aligned} \qquad \begin{aligned} \text{maximizar} \quad & x + 5y \\ \text{sujeto a} \quad & x + y \leq 2.5 \\ & 3x \leq y \\ & x \geq 1.5 \\ & x, y \in \{0, 1, 2, 3\} \end{aligned}$$

En cada vértice del árbol resolveremos la relajación del nuevo modelo. Si el modelo resultante es factible, el valor de dicha relajación nos da una cota del óptimo en el subárbol respectivo. Si el modelo resultante es infactible, podemos descartar el subárbol completo.

En la práctica, las restricciones que se agregan para construir los nuevos vértices del árbol se determinan de forma heurística. Gurobi generalmente decide primero sobre las variables que más impacto tienen en el valor de la función objetivo, pero esto no es necesariamente cierto para todos los modelos. Tampoco es necesario que todas las decisiones del mismo nivel del árbol se refieran a la misma variable. Eventualmente, el solucionador debe tomar decisiones finales sobre los valores enteros de las variables.

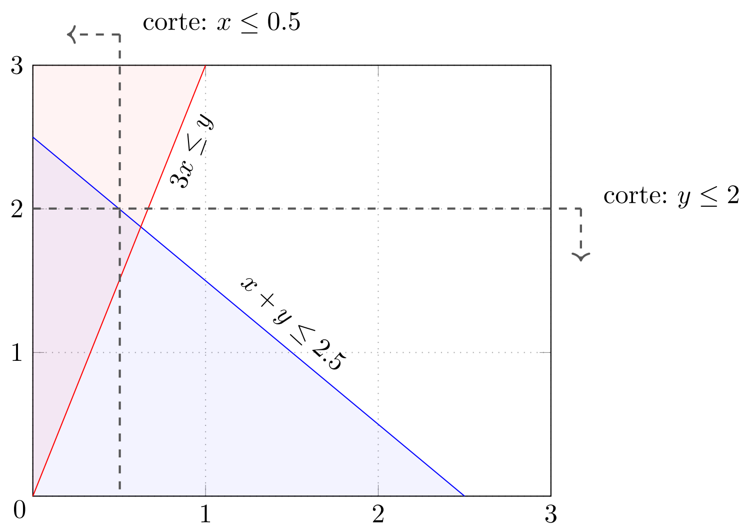
Independientemente de cómo se construye el árbol, la forma de explorarlo suele ser en prioridad: primero se exploran los vértices que sean más prometedores con respecto al valor de la relajación de sus modelos. Hay al menos dos razones que justifican dicho orden de exploración: la primera es que es más probable encontrar buenas soluciones factibles en subárboles prometedores; la segunda es que, una vez encontrada una solución factible, podremos descartar subárboles cuyas relajaciones no sean mejores que el valor de dicha solución. La aceleración producida por la poda del árbol de búsqueda suele ser dramática y es por esto que Gurobi, además de buscar soluciones factibles mediante la exploración del árbol, también busca (desesperadamente) soluciones factibles mediante métodos heurísticos. A la mejor solución factible encontrada hasta el momento se le conoce en inglés como *incumbent*.



El árbol de ramificación y acotamiento para el modelo de ejemplo.

El algoritmo también actualiza el valor global de la relajación más prometedora (*best bound* en inglés) conforme se explora el árbol y se descartan relajaciones inviables debido a la integralidad de las variables, o bien, conforme éstas potencialmente empeoran al bajar por las ramas del árbol. Un valor conocido por la palabra inglesa *gap* cuantifica en términos porcentuales qué tan lejana está la mejor solución encontrada hasta el momento del valor actual de la relajación más prometedora. La resolución del modelo termina cuando el valor de la mejor solución encontrada iguala al valor global de la relajación más prometedora, o bien, cuando la exploración del árbol termina sin haber encontrado una solución factible.

Una variante del algoritmo de ramificación y acotamiento es el algoritmo de ramificación y corte. Este algoritmo, además de agregar restricciones que bifurquen el árbol de búsqueda, también agrega restricciones o planos de corte que descartan soluciones que son infactibles por la integralidad de las variables. Aunque los planos de corte son redundantes desde el punto de vista de la correctitud del modelo, éstos sirven para que la resolución de la relajación se aproxime a ser la resolución del modelo entero. Gurobi conoce varios algoritmos de generación de planos de corte y los aplica automáticamente.



El poliedro de la relajación del modelo y algunos planos de corte, los cuales no deben excluir soluciones enteras factibles.

El poliedro entero de un modelo discreto es uno que incluye todas sus soluciones factibles y además no tiene puntos extremos infactibles. Partiendo del poliedro de la relajación del modelo, es posible obtener el poliedro entero agregando los planos de corte adecuados, tras lo cual bastaría encontrar su punto extremo óptimo. Desafortunadamente, existen problemas que requieren una cantidad exponencial de planos de corte para obtener el poliedro entero de su modelo relajado.

Normalmente, los planos de corte que agrega Gurobi tienden a acelerar el proceso de optimización y nunca ocurre que se generan todos los planos de corte que se necesitarían para obtener el poliedro entero, ya que se suele encontrar la solución óptima mucho antes. Además, agregar una cantidad exagerada de planos de corte tendría el efecto negativo de provocar un aumento en el consumo de memoria y también de ralentizar el cálculo de las relajaciones que la exploración del árbol de búsqueda va requeriendo. Por esta razón, el beneficio que se obtiene de agregar planos de corte varía considerablemente.

5.1. Ejercicios

1. Dado el siguiente modelo entero, dibuja el poliedro de su relajación y encuentra planos de corte que describan un poliedro entero del modelo. Además, encuentra la solución óptima del modelo.

$$\begin{aligned} \text{maximizar} \quad & x + y \\ \text{sujeto a} \quad & y \leq 10 - 4x \\ & y \geq 5 - 4x \\ & y \leq 1.75 + 0.25x \\ & x, y \in \{0, 1, 2, 3\} \end{aligned}$$

6. Parámetros de optimización de `gurobi_cl`

Gurobi cuenta también con decenas de parámetros que permiten configurar el proceso de optimización. Ocasionalmente pueden llegar a tener un enorme impacto en el tiempo de ejecución o en la calidad de la solución, por lo que es útil tenerlos en cuenta. A continuación se describen los más importantes.

- `Presolve=`*`entero`*
  Controla la etapa de simplificación del modelo previo a que Gurobi intente resolverlo. Con `Presolve=0` se deshabilita el proceso, mientras que `Presolve=1` y `Presolve=2` le permiten a Gurobi dedicarle poco o mucho tiempo, respectivamente. Con modelos grandes y una simplificación agresiva, esta etapa puede incluso durar horas; ocasionalmente dicho esfuerzo sí vale la pena.

- `Method=`*`entero`*
  Determina la estrategia con la que se resolverán los modelos continuos. En particular, `Method=3` emplea en paralelo todos los métodos conocidos por Gurobi (simplex primal, simplex dual, método de barrera) y usa la solución del método que acabe primero.

- `MIPGap=`*`real`*
  Cuando el modelo tiene variables enteras, Gurobi normalmente va encontrando paulatinamente mejores soluciones y también va mejorando una cota de la mejor solución. Cuando la distancia entre la mejor solución encontrada hasta el momento y la cota es menor en porcentaje al valor de `MIPGap`, dicha solución se considerará óptima y la optimización se detiene.

- `MIPFocus=`*`entero`*
  Le indica a Gurobi una estrategia de alto nivel a seguir. Con `MIPFocus=1` Gurobi intentará concentrarse más en encontrar nuevas mejores soluciones que en probar optimalidad. Con `MIPFocus=2` le estaremos indicando a Gurobi que creemos que encontrar la solución óptima será fácil, por lo que debería incrementar la fracción del tiempo que le dedica a probar optimalidad. Con `MIPFocus=3` Gurobi dedicará aún más tiempo a mejorar la cota de la solución.

- `Heuristics=`*`real`*
  Determina qué fracción del tiempo de optimización entera debe dedicarle Gurobi a buscar soluciones mediante métodos heurísticos. Por ejemplo, `Heuristics=0.05` establece una fracción del 5 %.

- `Cuts=`*entero*
  Controla la generación de cortes de Gurobi. Con `Cuts=0` se deshabilitan los cortes, mientras que `Cuts=3` establece la política más agresiva de generación de cortes. Los cortes suelen ser efectivos para mejorar la cota de la función objetivo, aunque generarlos implica un gasto de tiempo y memoria. Gurobi también provee parámetros para controlar la generación de tipos específicos de cortes.

- `Threads=`*entero*
  Limita la cantidad de hilos que Gurobi puede emplear. Por omisión, Gurobi buscará aprovechar todos los recursos del hardware. Sin embargo, Gurobi hará una copia independiente del modelo para cada hilo durante la etapa de optimización entera. Si la computadora tiene poca memoria o si el sobrecalentamiento del hardware es un problema, puede convenir limitar el número de hilos.

- `Quad=1`
  Habilita el uso de precisión cuádruple en los cálculos de punto flotante de Gurobi. La exactitud de los cálculos mejora a expensas de tiempo de ejecución. Aunque su uso pareciera ser avanzado, algunas veces el propio Gurobi le sugiere al usuario reiniciar la optimización habilitando este parámetro.

- `IntFeasTol=`*real*
  Define una tolerancia bajo la cual un real se puede considerar entero. Por ejemplo, si `IntFeasTol=0.001` entonces 4.999 se considerará entero, mientras que 4.99 se considerará real.

7. Uso básico de la interfaz de programación de Gurobi

Además de poder resolver modelos expresados en el formato LP, Gurobi también nos permite construir y resolver modelos mediante su interfaz de programación de aplicaciones (API en su abreviatura en inglés). Gurobi proporciona envoltorios de su API para los lenguajes C, C++, Java, Python, R y Matlab, así como para la plataforma .NET de Microsoft. Estas notas usarán C++ como lenguaje de programación.

Dada la instalación disponible en el servidor del Posgrado en Optimización, un programa en C++ que quiera emplear el API de Gurobi podrá compilarse usando el siguiente comando:

```
g++ -std=c++20 -O3 programa.cpp -lgurobi_c++ -lgurobi90 -o programa
```

Los parámetros `-lgurobi_c++` y `-lgurobi90` enlazarán las bibliotecas precompiladas que contienen tanto el envoltorio en C++ del API, como el binario del API en sí. A su vez, el código fuente del programa deberá incluir el archivo de encabezado `gurobi_c++.h` para poder usar el envoltorio.

Para que un programa en C++ pueda construir un modelo con el API de Gurobi, primero debe declarar un ambiente, el cual es una variable de tipo `GRBEnv` que contiene los parámetros de configuración con los cuales se optimizarán los modelos vinculados al mismo. Todos los parámetros de Gurobi tienen valores por omisión, así que no es obligatorio establecer ninguno de ellos. Desafortunadamente, cuando Gurobi se activa con una licencia académica y un programa usa su API, el programa en cuestión emitirá la leyenda "Academic license - for non-commercial use only" aún si se establece el parámetro `OutputFlag` en `0`. Sin embargo, si se escribe la línea `OutputFlag=0` en un archivo llamado `gurobi.env`, entonces los programas que corran en la misma carpeta no emitirán dicha leyenda.

Una vez declarado y configurado el ambiente de Gurobi, podremos declarar una variable de tipo `GRBModel` vinculada a dicho ambiente y que representará el modelo matemático que queremos resolver.

Código

```
#include <gurobi_c++.h>

int main( ) {
   GRBEnv ambiente;
   ambiente.set("TimeLimit", "100");    // opcional: parámetro de configuración
   GRBModel modelo(ambiente);
   //...
}
```

El modelo de programación lineal podrá construirse paso a paso antes de resolverlo. La función miembro

`GRBModel::addVar(double ci, double cs, double cf, char t, const std::string& s = "");`

nos permite agregar variables al modelo y sus parámetros denotan lo siguiente:

- El parámetro `double ci` es la cota inferior de la variable.
- El parámetro `double cs` es la cota superior de la variable.
- El parámetro `double cf` es el coeficiente de la variable en la función objetivo. Existe una forma alternativa de establecer la función objetivo, la cual sobreescribe el valor de `cf`.
- El parámetro `char t` es el tipo de la variable y puede ser `GRB_BINARY`, `GRB_INTEGER`, `GRB_CONTINUOUS`, `GRB_SEMICONT` o `GRB_SEMIINT`.
- El parámetro `const std::string& s` es un nombre para la variable. Este parámetro es opcional y su valor se usa en algunas pocas situaciones, como por ejemplo, cuando le solicitamos a Gurobi que imprima un modelo construído con el API.

Los valores `-GRB_INFINITY` y `+GRB_INFINITY` pueden usarse para indicar que la variable no tiene cota inferior o superior, respectivamente. La función `GRBModel::addVar` regresa un manipulador de tipo `GRBVar`, el cual está vinculado con la variable que fue creada del lado del modelo de Gurobi. El API de Gurobi sobrecarga los operadores del lenguaje C++ para poder crear expresiones lineales y restricciones. Por ejemplo, si las variables `x1`, `x2` son de tipo `GRBVar` y `c1`, `c2` son de tipo `double`, entonces la expresión `2 * x1 + x2 + c1` será de tipo `GRBLinExpr`, mientras que `x1 - c1 * x2 + 5 <= c2 * x1 - c1` será de tipo `GRBTempConstr` porque denota una desigualdad. Las debilidades del formato LP (donde no se pueden sumar constantes en la función objetivo y donde hay limitaciones en lo que puede aparecer en cada lado de una restricción) no aplican con el API. La función objetivo se puede establecer con la función

`GRBModel::setObjective(GRBLinExpr e, int s);`

donde el parámetro `s` es el sentido de la optimización y puede ser `GRB_MINIMIZE` o `GRB_MAXIMIZE`. De forma similar, las restricciones del modelo se pueden agregar con la siguiente función miembro:

`GRBModel::addConstr(GRBTempConstr c, const std::string& = "");`

El parámetro `s` es un nombre opcional para la restricción. La función miembro `GRBModel::optimize( )` comienza el proceso de optimización y no regresa hasta que haya terminado. Una vez terminada la optimización, su estado se puede obtener solicitándole a la función miembro `GRBModel::get(`*atributo*`)` el valor del atributo `GRB_IntAttr_Status`. Los valores más comunes que puede tomar dicho atributo son `GRB_OPTIMAL`, `GRB_UNBOUNDED` y `GRB_INFEASIBLE` para denotar que se encontró una solución óptima, que el modelo no está acotado o que el modelo es infactible, respectivamente. De forma similar, el valor de la función objetivo se puede obtener con el atributo `GRB_DoubleAttr_ObjVal`, así como también se puede consultar el valor de cada variable en la solución obtenida usando la función miembro `GRBVar::get(`*atributo*`)` con el atributo `GRB_DoubleAttr_X`, recordando que el valor de la variable será un número en punto flotante con tolerancia, aún si la variable respectiva era entera.

A continuación se reproduce un modelo de programación lineal entera y se muestra cómo puede implementarse usando el API en C++ de Gurobi.

$$\begin{aligned} \text{maximizar} \quad & 2x + y - z + 8 \\ \text{sujeto a} \quad & x + y + z \leq 5 \\ & 2x \leq 0.2z \\ & z - 1 \leq 2 \\ & x + y + 1 \geq x - y \\ & x, y \in \mathbb{Z}_2, z \in \mathbb{R} \end{aligned}$$

Código

```
#include <iostream>
#include <stdio.h>
#include <gurobi_c++.h>

int main( ) try {
   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x = modelo.addVar(0,             1,             0, GRB_BINARY);
   GRBVar y = modelo.addVar(0,             1,             0, GRB_BINARY);
   GRBVar z = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);

   modelo.setObjective(2 * x + y - z + 8, GRB_MAXIMIZE);
   modelo.addConstr(x + y + z <= 5);
   modelo.addConstr(2 * x <= 0.2 * z);
   modelo.addConstr(z - 1 >= 2);

   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      std::cout << modelo.get(GRB_DoubleAttr_ObjVal) << "\n";
      std::cout << "x = " << x.get(GRB_DoubleAttr_X) << "\n";
      std::cout << "y = " << y.get(GRB_DoubleAttr_X) << "\n";
      std::cout << "z = " << z.get(GRB_DoubleAttr_X) << "\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Estado no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```

Cuando algún error ocurre (por ejemplo, si la licencia de uso venció) Gurobi eleva un error en forma de excepción. Es buena costumbre atrapar dicho error y emitir el diagnóstico respectivo.

Es posible declarar una expresión lineal de tipo `GRBLinExpr` inicialmente vacía, y luego usar los operadores `+=` y `-=` para agregarles términos paso a paso. Por ejemplo:

Código

```
GRBVar x = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_INTEGER);
GRBVar y = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_INTEGER);

GRBLinExpr ex;
ex += 2 * x;
ex += 8;
ex -= 5 * y;
modelo.addConstr(ex <= 7);    // modelo.addConstr(2 * x + 8 - 5 * y <= 7);
```

Ambos lados de una desigualdad pueden ser de tipo `GRBLinExpr`. Además, una desigualdad también se puede almacenar en una variable de tipo `GRBConstr`, aunque hacer esto es poco usual.

También es posible declarar un manipulador de tipo `GRBVar` e inicializarlo posteriormente, así como hacer que dos manipuladores estén vinculados a la misma variable del modelo. Es un error intentar usar un manipulador que no esté vinculado a una variable del modelo.

Código

```
GRBVar x;                              // manipulador sin inicializar
x = modelo.addVar(0, 1, 0, GRB_BINARY); // vinculado a una variable del modelo
GRBVar y = x;                          // vinculado a la misma variable
modelo.addConstr(x == 5);              // ok
modelo.addConstr(y == 7);              // ok pero infactible (misma variable)

GRBVar w;                              // manipulador sin inicializar
modelo.addConstr(w == 7);              // error en ejecución (GRBException)
```

7.1. Ejercicios

1. Escribe un programa que use el API de Gurobi para modelar el problema de la mochila en su versión discreta. Puedes suponer que la instancia será descrita mediante dos enteros n y c seguidos de n parejas de enteros p_i y v_i, donde n denota la cantidad de objetos disponibles, c denota la capacidad de la mochila y las n parejas denotan el peso y el valor de los objetos.

8. Funciones de conveniencia de la interfaz de Gurobi

El API de Gurobi incluye funciones miembro adicionales que no son indispensables pero que frecuentemente facilitan la experiencia del usuario. A continuación se listan y se describen las más importantes.

- `GRBModel::update( );`
  Por cuestiones de eficiencia, Gurobi no construye una representación del modelo hasta comenzar la optimización. Esta función le indica a Gurobi que construya anticipadamente dicha representación.

- `GRBModel::computeIIS( );`
  Si el proceso de optimización determinó que el modelo es infactible, esta función calcula un subsistema infactible irreducible, el cual es un subconjunto infactible minimal del modelo. No se garantiza que dicho subconjunto además sea mínimo.

- `GRBModel::write(const std::string& s);`
  El parámetro `s` denota una ruta de archivo. Si el archivo tiene extensión `.lp`, entonces Gurobi escribirá el modelo en dicho archivo y usando tal formato. Si la optimización terminó encontrando soluciones factibles y el archivo tiene extensión `.sol`, entonces Gurobi escribirá la mejor solución encontrada en dicho archivo y usando tal formato. Si el modelo es un modelo infactible para el que ya se calculó el subsistema infactible irreducible y además el archivo tiene extensión `.ilp`, entonces Gurobi escribirá tal subsistema en dicho archivo y usando el formato LP.

Código

```
modelo.update( );          // necesario para escribir el modelo en formato LP
modelo.write("modelo.lp");
modelo.optimize( );
if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
   std::cout << "Modelo resuelto\n";
   modelo.write("modelo.sol");
} else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
   std::cout << "Modelo no acotado\n";
} else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
   std::cout << "Modelo infactible\n";
   modelo.computeIIS( );
   modelo.write("modelo.ilp");
} else {
   std::cout << "Estado no manejado\n";
}
```

Recordemos que las variables creadas con `GRBModel::addVar` pueden recibir un nombre opcional. Gurobi emplea estos nombres al escribir modelos y soluciones mediante `GRBModel::write` y usará nombres arbitrarios para las variables que no tengan nombre explícito, lo cual difícilmente es lo que se quiere. Por ello, es importante especificar nombres explícitos para las variables al usar esta función.

Código

```
#include <string>
#include <gurobi_c++.h>
#include <stdio.h>

template<typename... T>
std::string formato(const char* s, const T&... p) {
   char bufer[32 + 1];     // suposición: suficiente para un nombre de variable
   sprintf(bufer, s, p...);
   return bufer;
}

int main( ) {
   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x[5];
   for (int i = 0; i < 5; ++i) {
      x[i] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d", i));
   }
   //...

   modelo.update( );
   modelo.write("modelo.lp");     // variables x0, x1, x2, x3, x4
}
```

- `GRBModel::read(const std::string& s);`
  El parámetro `s` denota una ruta de archivo. Si la representación del modelo ya fue construida, si el proceso de optimización aún no ha comenzado y además el archivo tiene extensión `.sol` o `.mst`, entonces Gurobi leerá del archivo una asignación de valores a algunas o todas las variables del modelo. Antes de comenzar la optimización, Gurobi intentará determinar si la asignación leída del archivo le permite construir una solución factible inicial. Como ya se discutió, Gurobi intenta encontrar soluciones factibles tan pronto como sea posible para estar en posibilidades de podar del árbol de búsqueda durante la etapa de optimización entera. Aún si la asignación de valores no produce una solución factible, Gurobi intentará arreglarla en caso de no encontrar soluciones factibles por otros medios.

- `GRBModel::addGenConstrMin(`
  `GRBVar v, GRBVar arr[], int n, double c = +GRB_INFINITY, const std::string& = "");`
  `GRBModel::addGenConstrMax(`
  `GRBVar v, GRBVar arr[], int n, double c = -GRB_INFINITY, const std::string& = "");`
  Estas funciones agregan restricciones del tipo $v = f(arr_0, arr_1, \ldots, arr_{n-1}, c)$ donde f puede ser mín o máx, respectivamente. Gurobi se encarga de construir un conjunto de desigualdades lineales semánticamente equivalentes usando las variables auxiliares necesarias. Por ejemplo, $v = \text{mín}(a, b)$ se puede escribir con programación lineal entera de la siguiente forma:

$$
\begin{aligned}
v &\leq a \\
v &\leq b \\
v &\geq a + Mt \\
v &\geq b + M(1-t)
\end{aligned}
$$

donde M es un valor constante lo suficientemente grande como para jugar un pseudo-rol de ∞ y $t \in \mathbb{Z}_2$ es una variable auxiliar. De forma similar, $v = \max(a, b)$ se puede modelar como sigue:

$$\begin{aligned} v &\geq a \\ v &\geq b \\ v &\leq a + Mt \\ v &\leq b + M(1 - t) \end{aligned}$$

donde t una variable auxiliar distinta a la del ejemplo anterior.

- `GRBModel::addGenConstrAnd(GRBVar v, GRBVar arr[], int n, const std::string& = "");`
  `GRBModel::addGenConstrOr(GRBVar v, GRBVar arr[], int n, const std::string& = "");`
  Estas funciones agregan restricciones del tipo $v = f(arr_0, arr_1, \ldots, arr_{n-1})$ donde f puede ser $\wedge$ o $\vee$, respectivamente. Gurobi se encarga de construir un conjunto de desigualdades lineales semánticamente equivalentes. Por ejemplo, $v = a \wedge b$ se puede escribir con programación lineal de la siguiente forma:

  $$\begin{aligned} v &\leq a \\ v &\leq b \\ v &\geq a + b - 1 \end{aligned}$$

  De forma similar, $v = a \vee b$ se puede modelar como sigue:

  $$\begin{aligned} v &\geq a \\ v &\geq b \\ v &\leq a + b \end{aligned}$$

- `GRBModel::addGenConstrAbs(GRBVar v, GRBVar a, const std::string& = "");`
  Esta función agrega una restricción $v = |a|$. Gurobi se encarga de construir un conjunto de desigualdades lineales semánticamente equivalentes usando las variables auxiliares necesarias. Por ejemplo, $v = |a|$ se puede escribir con programación lineal entera de la siguiente forma:

  $$\begin{aligned} v &\leq a \\ v &\leq -a \\ v &\geq a + Mt \\ v &\geq -a + M(1 - t) \end{aligned}$$

  donde M es un valor constante lo suficientemente grande como para jugar un pseudo-rol de ∞ y $t \in \mathbb{Z}_2$ es una variable auxiliar. De hecho, el valor absoluto corresponde con $v = \max(a, -a)$.

8.1. Ejercicios

1. Escribe un programa que use el API de Gurobi para construir y resolver el siguiente modelo.

$$\begin{aligned} \text{maximizar} \quad & x - 2y + z \\ \text{sujeto a} \quad & |x| \leq 5 \\ & |x + y| \leq 8 \\ & z = \min(x - 1, y) \\ & x, y, z \in \mathbb{R} \end{aligned}$$

9. Uso de múltiples funciones objetivo

Gurobi permite especificar múltiples funciones objetivo, aunque todas deben tener el mismo sentido de optimización. Esto último no representa un problema porque, por ejemplo, la minimización de un objetivo

negado equivale a la maximización del objetivo no negado. El sentido de la optimización de todas las funciones se puede establecer con el atributo `GRB_IntAttr_ModelSense` de `GRBModel` y por defecto es minimización. Cada objetivo se debe especificar de forma individual con la siguiente función miembro:

```
GRBModel::setObjectiveN(GRBLinExpr e, int i, int p, double w = 1, /* otros */);
```

El parámetro `e` es la expresión lineal, el parámetro `i` denota el índice del objetivo (deben estar numerados a partir de 0) y los parámetros `p` y `w` denotan la prioridad y el peso del objetivo, respectivamente. Cuando existan objetivos con distinta prioridad, Gurobi optimizará primero el objetivo de mayor prioridad y luego optimizará los demás (también en orden de prioridad) bajo la condición de que no deberá empeorar los objetivos previamente optimizados; a esto se le conoce como optimización jerárquica. Cuando existan objetivos con la misma prioridad, Gurobi construirá y optimizará un objetivo que sea la suma de éstos multiplicándolos por su respectivo peso `w`; a esto se le conoce como optimización ponderada. Es posible combinar el uso de objetivos jerárquicos y ponderados en el mismo modelo. Por ejemplo, dos objetivos pueden tener prioridad p_1 y dos objetivos pueden tener prioridad $p_2 < p_1$. En este caso, Gurobi optimizará jerárquicamente dos objetivos ponderados: primero optimizará la suma ponderada de los objetivos con prioridad p_1 y luego optimizará la suma ponderada de los objetivos con prioridad p_2.

El valor del atributo `GRB_DoubleAttr_ObjVal` de `GRBModel` siempre corresponderá con el valor del primer objetivo del modelo. Para examinar el valor de los demás objetivos una vez terminada la optimización, se debe sobreescribir el atributo `GRB_IntParam_ObjNumber` con el índice del objetivo deseado y luego consultar el valor del atributo `GRB_DoubleAttr_ObjNVal`. A su vez, el atributo `GRB_DoubleAttr_NumObj` permite examinar el número de objetivos establecidos en el modelo.

Código

```
GRBVar x = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
GRBVar y = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
modelo.addConstr(x <= 5);
modelo.addConstr(y <= 5);
modelo.addConstr(x + y <= 5);

modelo.set(GRB_IntAttr_ModelSense, GRB_MAXIMIZE);
modelo.setObjectiveN(x + y, 0, +0);
modelo.setObjectiveN(x,     1, -1, 5);
modelo.setObjectiveN(y,     2, -1, 5);

modelo.optimize( );
if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
   for (int i = 0; i < modelo.get(GRB_IntAttr_NumObj); ++i) {
      modelo.set(GRB_IntParam_ObjNumber, i);
      std::cout << modelo.get(GRB_DoubleAttr_ObjNVal) << "\n";
   }
   std::cout << "x" << " = " << x.get(GRB_DoubleAttr_X) << "\n";
   std::cout << "y" << " = " << y.get(GRB_DoubleAttr_X) << "\n";
} else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
   std::cout << "Modelo no acotado\n";
} else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
   std::cout << "Modelo infactible\n";
} else {
   std::cout << "Estado no manejado\n";
}
```

Es posible indicarle a Gurobi que permita que objetivos de menor prioridad degraden un poco los valores objetivo de mayor prioridad. También ocurre que la optimización multiobjetivo de modelos sin variables enteras se comporta de forma especial cuando se permite degradación. Para más información, consultar `https://www.gurobi.com/documentation/9.1/refman/working_with_multiple_obje.html`.

9.1. Ejercicios

1. El problema del agente viajero es un problema NP-Duro que es clásico en optimización. Este ejercicio trata de resolver la versión original del problema y una de sus variantes, la cual también es NP-Dura.

   Problema: El agente viajero.
   Entrada: Un natural n y una gráfica completa g de n vértices con pesos en sus aristas.
   Salida: Un camino cerrado de costo mínimo sobre g que visite cada vértice exactamente una vez. El costo del camino está dado por la suma de los costos de sus aristas.

   Problema: El agente viajero con cuello de botella.
   Entrada: Un natural n y una gráfica completa g de n vértices con pesos en sus aristas.
   Salida: Un camino cerrado sobre g que visite cada vértice exactamente una vez y que minimice el costo de su arista más cara.

   Escribe un programa que use el API de Gurobi para resolver el agente viajero con dos objetivos: el principal deberá ser minimizar la suma de los costos de las aristas del camino; el secundario deberá ser minimizar el costo de la arista más cara usada. Puedes suponer que la instancia será descrita mediante un entero n seguido de $\binom{n}{k}$ tripletas de enteros (v_1, v_2, w). Los vértices de la gráfica estarán numerados de 0 a $n-1$ y cada tripleta denota una arista de peso w que conecta los vértices v_1 y v_2.

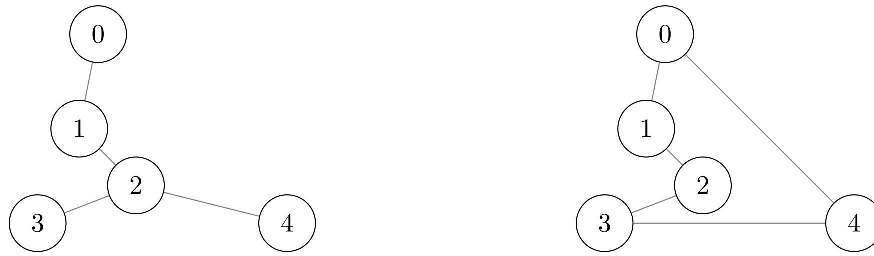
10. Algoritmos de aproximación y soluciones iniciales

Gurobi suele ser muy eficaz encontrando soluciones factibles de buena calidad aún para problemas NP-Duros, por lo que proporcionarle a Gurobi cualquier solución inicial no resultará en una aceleración notable del proceso de optimización. Sin embargo, algunos problemas NP-Duros cuentan con algoritmos polinomiales que calculan soluciones cuyos valores están cercanos al óptimo (¡aún si no se sabe cuánto vale el óptimo!). A estos algoritmos se les denomina algoritmos de aproximación y una solución con este tipo de garantías puede constituir una excelente solución inicial para Gurobi.

El ejercicio de la sección anterior trata del problema del agente viajero sobre una gráfica general. Este problema tiene una versión euclidiana que sigue siendo NP-Dura pero que cuenta con algoritmos de aproximación que generan soluciones que están a lo mucho un factor constante por encima del óptima.

Problema: El agente viajero (versión euclidiana).
Entrada: Un natural n y una gráfica completa g de n vértices con pesos en sus aristas. Los vértices tienen coordenadas en $\mathbb{R}^2$ y las aristas tienen por peso la distancia euclidiana de los vértices unidos.
Salida: Un camino cerrado de costo mínimo que visite cada vértice exactamente una vez. El costo del camino está dado por la suma de los costos de sus aristas.

Un árbol abarcador de costo mínimo de la gráfica g tiene un costo c_t menor al valor c de la solución óptima del agente viajero, ya que dicho árbol conecta todos los vértices a costo mínimo y además no tiene ciclos. Además, el árbol se puede calcular en tiempo $O(n^2)\log_2(n)$ usando el algoritmo de Prim. Claramente, el árbol abarcador de costo mínimo no es una solución factible al problema del agente viajero euclidiano. Sin embargo, podemos construir una solución a partir de recorrer dicho árbol en profundidad y anotar en qué orden se visitaron los vértices. Como la gráfica es completa, dicho orden es un orden válido en el que se pueden visitar los vértices de g en una solución factible (no necesariamente óptima) del problema agente viajero. Supondremos que esta solución tiene costo $c_r \geq c$ y sabemos que $c_t < c$, por lo que sólo falta por determinar la relación entre c_r y c_t. Un recorrido en profundidad del árbol abarcador pasa dos veces por cada arista: una vez de ida y una vez de regreso. Una solución factible del agente viajero no repite vértices ni tampoco aristas, pero si la gráfica es completa entonces siempre hay una arista directa del vértice actual al siguiente del orden. Como además estamos en $\mathbb{R}^2$, dicha arista constituye la forma más barata de llegar al siguiente vértice. Esto significa que $c \leq c_r \leq 2c_t < 2c$ y entonces el algoritmo calcula soluciones con un valor menor al doble del valor óptimo. Se dice que el algoritmo es un algoritmo de aproximación con factor constante 2.



A la izquierda, un ejemplo de árbol abarcador de costo mínimo y un recorrido en profundidad que comienza en el vértice 0 y etiqueta el resto de los vértices conforme se visitan. A la derecha, la solución del problema del agente viajero construída a partir de dicho recorrido.

Gurobi puede tomar una solución inicial leyendo un archivo con extensión `.mst` y entonces, si algún programa escribe dicho archivo, nosotros podremos pedirle a Gurobi que lo lea mediante una llamada a `GRBModel::update` seguida de una llamada a `GRBModel::read`. Si por el contrario, el programa que calcula la solución inicial es el mismo que después interactuará con Gurobi, es posible evitar el uso del archivo intermedio si establecemos un valor para las variables mediante el parámetro `GRB_DoubleAttr_Start` de `GRBVar`. Por otra parte, recordemos que una solución inicial no necesita especificar el valor de todas las variables del modelo y también que los potenciales del modelo para el agente viajero corresponden con la permutación y determinan la solución, por lo que basta escribir dichos potenciales en el archivo.

10.1. Ejercicios

1. Escribe un programa que resuelva el problema del agente viajero en su versión euclidiana. Tu programa deberá generar una solución inicial mediante algún algoritmo de aproximación y después deberá usar el API de Gurobi para entregar dicha solución, construir el modelo y encontrar el óptimo. Puedes suponer que la instancia será descrita mediante un entero n seguido de n parejas de reales (x, y) que denotan las coordenadas en $\mathbb{R}^2$ de los n vértices de la gráfica.

11. Uso de retrollamadas durante la optimización

El API de programación de Gurobi provee herramientas para trabajar con Gurobi de forma cooperativa durante la resolución de un modelo. Por ejemplo, podremos examinar cada nueva solución justo en el momento en el que ésta se encuentra y también podremos entregarle a Gurobi nuevas soluciones que provengan de mejorar heurísticamente las soluciones que él mismo encontró. Incluso podremos rechazar soluciones factibles encontradas por Gurobi, así como agregar planos de corte personalizados. La interacción descrita se podrá llevar a cabo mediante la instalación de una retrollamada o *callback*.

En Gurobi, una retrollamada es una variable de un tipo `struct` que herede del tipo `GRBCallback` y se debe instalar previo a la optimización mediante la función `GRBModel::setCallback`. Cada vez que un evento relevante ocurra, Gurobi invocará la función miembro `callback` de la retrollamada.

Código

```
struct retrollamada : GRBCallback {
   void callback( ) {
      std::cout << "evento notificado por Gurobi\n";
   }
};

int main( ) {
   GRBEnv ambiente;
   GRBModel modelo(ambiente);
   retrollamada r;
   modelo.setCallback(&r);
   modelo.optimize( );
}
```

Dentro de la función miembro `callback`, el programa podrá examinar el valor de la variable heredada `where` que indica el tipo de evento que Gurobi está notificando. Los dos tipos más importantes ocurren cuando `where` vale `GRB_CB_MIPSOL` o cuando vale `GRB_CB_MIPNODE`. En el primer caso, Gurobi acaba de encontrar una nueva solución factible; en el segundo caso, Gurobi acaba de moverse en el árbol de búsqueda y estará en posibilidades de aceptar tanto soluciones calculadas por el usuario como planos de corte personalizados (no podremos entregarle a Gurobi ninguna de las dos cosas en otro momento).

Cuando `where` valga `GRB_CB_MIPSOL`, podremos consultar el valor de la solución encontrada llamando a la función `GRBCallback::getDoubleInfo` con el parámetro `GRB_CB_MIPSOL_OBJ`. A su vez, podremos consultar el valor de las variables del modelo llamando a la función `GRBCallback::getSolution` con la variable de interés como parámetro. Eso significa que la retrollamada debe tener acceso a las `GRBVar`.

Código

```
struct retrollamada : GRBCallback {
   GRBVar x;
   retrollamada(GRBVar v)       // constructor para inicializar el manipulador
   : x(v) {
   }
   void callback( ) {
      if (where == GRB_CB_MIPSOL) {
         // nueva solución factible encontrada (no necesariamente óptima)
         std::cout << getDoubleInfo(GRB_CB_MIPSOL_OBJ) << "\n";
         std::cout << "x: " << getSolution(x) << "\n";
      }
   }
};

int main( ) {
   GRBEnv ambiente;
   GRBModel modelo(ambiente);
   GRBVar x = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_INTEGER);
   modelo.addConstr(2 * x <= 10);
   modelo.setObjective(x + 3, GRB_MAXIMIZE);

   retrollamada r(x);           // guardar una copia del manipulador
   modelo.setCallback(&r);
   modelo.optimize( );
}
```

Durante la retrollamada es posible rechazar una solución encontrada por Gurobi, aunque es necesario haber habilitado el parámetro `LazyConstraints` de `GRBEnv`. El rechazo de la solución se logra usando `GRBCallback::addLazy` para agregar una restricción que invalida la solución entregada por Gurobi.

Código

```
struct retrollamada : GRBCallback {
   //...
   void callback( ) {
      if (where == GRB_CB_MIPSOL) {
         if (getSolution(x) > 0) {
            // olvidamos decirle a Gurobi que no queríamos x positivas
            addLazy(x <= 0);
         }
      }
   }
};

// en main: ambiente.set("LazyConstraints", "1");
```

Las restricciones perezosas, que son precisamente las restricciones fabricadas durante una retrollamada, no se usan en ejemplos tontos como el anterior, sino que se usan con modelos que tienen una cantidad exhorbitante y prohibitiva de restricciones. Usando restricciones perezosas, el programa está en la posibilidad de incorporar restricciones conforme éstas se vuelven relevantes para el proceso de optimización. No hay garantía de que Gurobi recuerde restricciones perezosas agregadas previamente, por lo que el programa debe estar preparado para emitirlas nuevamente si fuera necesario. Esto le permite a Gurobi minimizar el uso de memoria y de todos modos se garantiza que el proceso de resolución terminará, ya que la exploración del árbol de búsqueda siempre avanzará.

Cuando `where` valga `GRB_CB_MIPNODE`, podremos proporcionarle a Gurobi una solución que hayamos calculado por nuestra cuenta. La función `GRBCallback::setSolution` se usa para específicar el valor que toma alguna de las variables del modelo en nuestra solución. La función se puede llamar repetidamente y, al igual que al entregar soluciones iniciales, no es necesario asignarle valores a todas las variables del modelo; Gurobi intentará determinar el valor de las variables sin un valor asignado en la solución.

Código

```
struct retrollamada : GRBCallback {
   //...
   void callback( ) {
      if (where == GRB_CB_MIPNODE) {
         setSolution(x, 5);
      }
   }
};
```

También cuando `where` valga `GRB_CB_MIPNODE`, podremos agregar planos de corte usando la función `GRBCallback::addCut`. Recordemos que un plano de corte en realidad simplemente es una restricción lineal. Sin embargo, los planos de corte no tienen permitido eliminar soluciones factibles del modelo. Si el usuario agrega un plano de corte inválido, Gurobi podría calcular soluciones erroneas. Se recomienda habilitar el parámetro `PreCrush` de `GRBEnv` para evitar que Gurobi simplifique de más y termine ignorando silenciosamente algunos planos de cortes del usuario. Esto último se debe a que los planos cortes son, de hecho, restricciones redundantes desde el punto de vista de la correctitud del modelo.

Otros tipos de eventos se documentan en `https://www.gurobi.com/documentation/9.1/refman/cb_codes.html`. A su vez, el tipo de dato `GRBCallback` que define la interfaz de retrollamadas de Gurobi se documenta en `https://www.gurobi.com/documentation/9.1/refman/cpp_cb_.html`.

11.1. Ejercicios

1. Para los problemas cuyas soluciones factibles corresponden con permutaciones, podemos definir una heurística de búsqueda local como sigue. Definiremos la vecindad de una permutación como el conjunto de todas las permutaciones que se pueden obtener intercambiando alguna pareja de elementos de la permutacion original. El objetivo de la heurística será encontrar la mejor solución que se puede obtener examinando la permutación actual y sus vecinos.

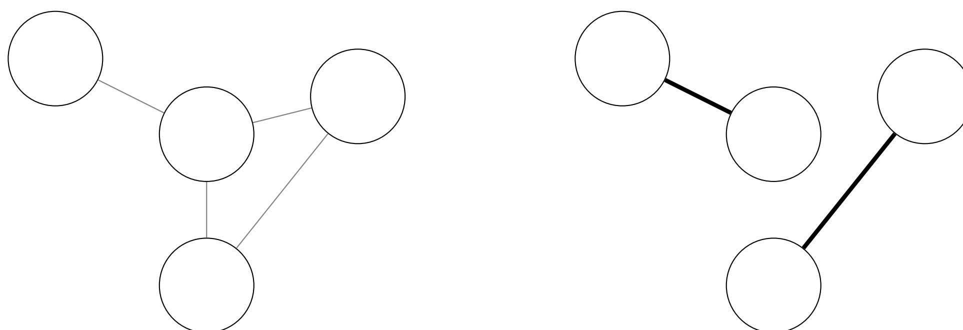
   Escribe un programa que resuelva el problema del agente viajero en su versión euclidiana. Tu programa deberá usar la heurística descrita arriba para mejorar las soluciones intermedias que el solucionador encuentra. Si la heurística determina que una permutación vecina es mejor que la solución actual, entonces tu programa deberá entregar dicha permutación mejorada como nueva mejor solución.

A. Soluciones de ejercicios

En general, siempre hay más de una posible respuesta correcta para cada pregunta. Razonen cuidadosamente las respuestas que propongo para que decidan si las suyas son equivalentes.

Soluciones de 3.1

1. Un acoplamiento de una gráfica es un conjunto de aristas sin vértices en común.



Ejemplo de gráfica y su acoplamiento de cardinalidad máxima.

Modela el problema de calcular un acoplamiento de cardinalidad máxima mediante un programa lineal y escribe un programa que lea la descripción de una gráfica y produzca el modelo en formato LP. Puedes suponer que la gráfica será descrita mediante dos enteros n y m seguidos de m parejas de enteros, donde n denota la cantidad de vértices de la gráfica, m denota la cantidad de aristas y las m parejas denotan los extremos de cada arista. Los vértices estarán numerados de 0 a $n-1$.

Solución: Aunque lo más natural sea definir variables binarias $x_{i,j}$ que denoten si la arista que conecta los vértices i, j se seleccionará en el acoplamiento, definiremos variables binarias $x_{i,j}$ que denoten arcos. Esto se hará así para facilitar la implementación del generador del modelo en formato LP; cuando se vean mejores métodos de construcción de modelos, podremos usar estrategias más naturales. Diremos que $x_{i,j} = x_{j,i}$, lo que en realidad significa que cada arista aparecerá en la forma de ambos arcos. Si una arista no existe en la gráfica entonces $x_{i,j} = 0$. La función objetivo maximizará la suma de las $x_{i,j}$, pero hay que recordar que cada arista en realidad se expresa con dos variables. Finalmente, restringiremos a que se elija a lo mucho un arco saliente por vértice (recordando que el arco entrante provendrá de la misma arista).

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 0.5x_{i,j} & \\ \text{sujeto a} \quad & x_{i,j} = x_{j,i} & \text{para } 0 \le i \ne j < n \\ & x_{i,j} = 0 & \text{para } (i,j) \notin E(G) \\ & \sum_{j=0}^{j-1} x_{i,j} \le 1 & \text{para } 0 \le i < n \\ & x_{i,j} \in \mathbb{Z}_2 & \text{para } 0 \le i,j < n \end{aligned}$$

En cuanto a la generación del modelo en formato LP, hay que recordar que no se pueden emplear términos constantes en la función objetivo, lo que nos obliga a cuidar la generación de su último sumando, ya que la función no puede terminar en `+` y tampoco en `+ 0`. También debemos recordar que no se permiten variables del lado derecho de una desigualdad. A su vez, es recomendable usar guiones bajos para separar los subíndices de una variable; la ausencia de un separador provocaría que las variables $x_{1,23}$ y $x_{12,3}$ sean indistinguibles en el modelo, lo cual está mal.

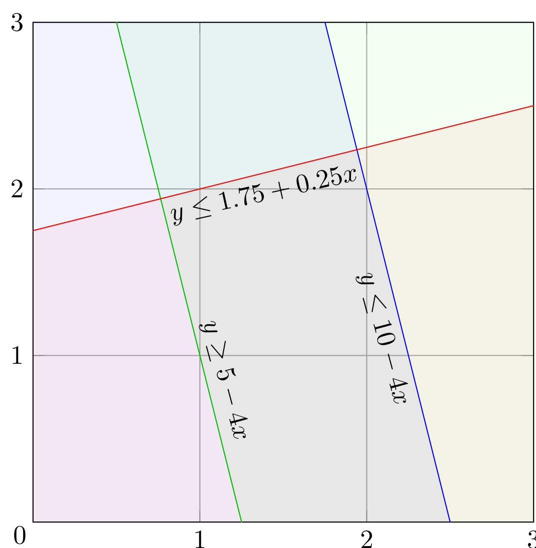
```
#include <iostream>
#include <stdio.h>

int main( ) {
   int n, m;
   std::cin >> n >> m;

   bool adyacencia[n][n] = { };
   for (int i = 0; i < m; ++i) {
      int x, y;
      std::cin >> x >> y;
      adyacencia[x][y] = true;
      adyacencia[y][x] = true;
   }

   printf("maximize\n");
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         printf("0.5 x%d_%d %s", i, j, (i == j && j == n - 1 ? "\n" : " + "));
      }
   }

   printf("subject to\n");
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         if (i != j) {
            printf("x%d_%d - x%d_%d = 0\n", i, j, j, i);
         }
      }
   }
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         if (!adyacencia[i][j]) {
            printf("x%d_%d = 0\n", i, j);
         }
      }
   }
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         printf("x%d_%d %s", i, j, (j == n - 1 ? " <= 1\n" : " + "));
      }
   }

   printf("binaries\n");
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         printf("x%d_%d\n", i, j);
      }
   }
   printf("end\n");
}
```

Soluciones de 5.1

1. Dado el siguiente modelo entero, dibuja el poliedro de su relajación y encuentra planos de corte que describan un poliedro entero del modelo. Además, encuentra la solución óptima del modelo.

$$\begin{aligned} \text{maximizar} \quad & x + y \\ \text{sujeto a} \quad & y \leq 10 - 4x \\ & y \geq 5 - 4x \\ & y \leq 1.75 + 0.25x \\ & x, y \in \{0, 1, 2, 3\} \end{aligned}$$

Solución: A continuación se muestra el poliedro de la relajación del modelo.



Las soluciones factibles son $x = 1, y = 1$; $x = 1, y = 2$; $x = 2, y = 0$; $x = 2, y = 1$; $x = 2, y = 2$. Por lo tanto, los planos de corte que describen el poliedro entero son $x \geq 1$; $x \leq 2$; $y \geq 2 - x$; $y \leq 2$. Finalmente, el óptimo está en el punto crítico $x = 2, y = 2$.

Soluciones de 7.1

1. Escribe un programa que use el API de Gurobi para modelar el problema de la mochila en su versión discreta. Puedes suponer que la instancia será descrita mediante dos enteros n y c seguidos de n parejas de enteros p_i y v_i, donde n denota la cantidad de objetos disponibles, c denota la capacidad de la mochila y las n parejas denotan el peso y el valor de los objetos.

Solución: Implementaremos el modelo matemático que se describió en las notas para el mismo.

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} v_i x_i \\ \text{sujeto a} \quad & \sum_{i=0}^{n-1} p_i x_i \leq c \\ & x_i \in \mathbb{Z}_2 && \text{para } 0 \leq i < n \end{aligned}$$

Una vez leída la instancia, procederemos a declarar los manipuladores de las variables y a crear éstas del lado de Gurobi. La única restricción del modelo tendrá que construirse con un ciclo, ya que desconocemos de antemano el valor de n. Lo mismo ocurrirá con la función objetivo.

```
#include <iostream>
#include <gurobi_c++.h>

struct objeto {
   int peso, valor;
};

int main( ) try {
   // lectura de la entrada

   int n, c;
   std::cin >> n >> c;

   objeto objetos[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> objetos[i].peso >> objetos[i].valor;
   }

   // construcción del modelo

   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x[n];
   for (int i = 0; i < n; ++i) {
      x[i] = modelo.addVar(0, 1, 0, GRB_BINARY);
   }

   GRBLinExpr restriccion;
   for (int i = 0; i < n; ++i) {
      restriccion += objetos[i].peso * x[i];
   }
   modelo.addConstr(restriccion <= c);

   GRBLinExpr objetivo;
   for (int i = 0; i < n; ++i) {
      objetivo += objetos[i].valor * x[i];
   }
   modelo.setObjective(objetivo, GRB_MAXIMIZE);

   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      std::cout << modelo.get(GRB_DoubleAttr_ObjVal) << "\n";
      for (int i = 0; i < n; ++i) {
         std::cout << "x" << i << " = " << x[i].get(GRB_DoubleAttr_X) << "\n";
      }
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
```

```
   } else {
      std::cout << "Estado no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```

Soluciones de 8.1

1. Escribe un programa que use el API de Gurobi para construir y resolver el siguiente modelo.

$$\begin{aligned} \text{maximizar} \quad & x - 2y + z \\ \text{sujeto a} \quad & |x| \leq 5 \\ & |x + y| \leq 8 \\ & z = \text{mín}(x - 1, y) \\ & x, y, z \in \mathbb{R} \end{aligned}$$

Solución: Usaremos las funciones de conveniencia del API de Gurobi para implementar el modelo de forma literal. Se debe recordar que las restricciones generales de Gurobi trabajan sobre variables, por lo que puede ser necesario crear variables auxiliares para las expresiones intermedias.

```
#include <array>
#include <iostream>
#include <gurobi_c++.h>

int main( ) try {
   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
   GRBVar y = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
   GRBVar z = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);

   GRBVar xabs = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
   modelo.addGenConstrAbs(xabs, x);
   modelo.addConstr(xabs <= 5);

   GRBVar xy = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
   modelo.addConstr(xy == x + y);
   GRBVar xyabs = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
   modelo.addGenConstrAbs(xyabs, xy);
   modelo.addConstr(xyabs <= 8);

   GRBVar x_1 = modelo.addVar(-GRB_INFINITY, +GRB_INFINITY, 0, GRB_CONTINUOUS);
   modelo.addConstr(x_1 == x - 1);
   modelo.addGenConstrMin(z, std::array{ x_1, y }.data( ), 2);

   modelo.setObjective(x - 2 * y + z, GRB_MAXIMIZE);
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      std::cout << "x" << " = " << x.get(GRB_DoubleAttr_X) << "\n";
```

```
      std::cout << "y" << " = " << y.get(GRB_DoubleAttr_X) << "\n";
      std::cout << "z" << " = " << z.get(GRB_DoubleAttr_X) << "\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Estado no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```

Soluciones de 9.1

1. El problema del agente viajero es un problema NP-Duro que es clásico en optimización. Este ejercicio trata de resolver la versión original del problema y una de sus variantes, la cual también es NP-Dura.

   Problema: El agente viajero.
   Entrada: Un natural n y una gráfica completa g de n vértices con pesos en sus aristas.
   Salida: Un camino cerrado de costo mínimo sobre g que visite cada vértice exactamente una vez. El costo del camino está dado por la suma de los costos de sus aristas.

   Problema: El agente viajero con cuello de botella.
   Entrada: Un natural n y una gráfica completa g de n vértices con pesos en sus aristas.
   Salida: Un camino cerrado sobre g que visite cada vértice exactamente una vez y que minimice el costo de su arista más cara.

   Escribe un programa que use el API de Gurobi para resolver el agente viajero con dos objetivos: el principal deberá ser minimizar la suma de los costos de las aristas del camino; el secundario deberá ser minimizar el costo de la arista más cara usada. Puedes suponer que la instancia será descrita mediante un entero n seguido de $\binom{n}{k}$ tripletas de enteros (v_1, v_2, w). Los vértices de la gráfica estarán numerados de 0 a $n-1$ y cada tripleta denota una arista de peso w que conecta los vértices v_1 y v_2.

   Solución: En la solución que se propone, descompondremos cada arista de la gráfica en dos arcos con direcciones opuestas. Por tal razón, definiremos variables binarias $x_{i,j}$ que denoten si el arco que conecta los vértices i, j formará parte del camino cerrado. La funcion objetivo principal es la minimización de la suma de las $w_{i,j}x_{i,j}$ donde $w_{i,j}$ es el peso del arco respectivo, mientras que la función objetivo secundaria se puede expresar como la minimización de una variable auxiliar t tal que t a su vez deba ser mayor o igual que todas las $w_{i,j}x_{i,j}$.

   Con respecto a las restricciones del modelo, en un camino cerrado que visita cada vértice una vez hay exactamente un arco saliente y un arco entrante por vértice. Sin embargo, se debe evitar que el solucionador construya varios caminos cerrados disjuntos. Para obligar al solucionador a construir un único camino, haremos lo siguiente. Primero supondremos que el camino cerrado comienza en el vértice 0 (lo cual es válido, porque en un camino cerrado que incluya todos los vértices, cualquiera puede considerarse el vértice inicial). Después asignaremos un potencial entero p_i entre 1 y $n-1$ al resto de los vértices, bajo la restricción de que el potencial del vértice destino de un arco debe ser mayor que el potencial del vértice origen del mismo. Cualquier camino cerrado que no incluya al vértice 0 será infactible porque será imposible asignar potenciales crecientes a

sus vértices. Entonces, sólo se construirá un camino que incluya al vértice 0 y a todos los demás.

$$\text{minimizar} \quad \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} w_{i,j}x_{i,j}$$

$$\text{minimizar} \quad t$$

$$\begin{aligned} \text{sujeto a} \quad & t \geq w_{i,j}x_{i,j} && \text{para } 0 \leq i,j < n \\ & \sum_{j=0}^{n-1} x_{i,j} = 1 && \text{para } 0 \leq i < n \\ & \sum_{i=0}^{n-1} x_{i,j} = 1 && \text{para } 0 \leq j < n \\ & p_i - p_j + (n * x_{i,j}) \leq n-1 && \text{para } 1 \leq i,j < n \end{aligned}$$

```
#include <algorithm>
#include <iostream>
#include <string>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

struct arista {
   int v0, v1, w;
};

int main( ) try {
   // lectura de la entrada

   int n;
   std::cin >> n;

   int adyacencia[n][n] = { };
   for (int i = 0; i < (n * n - n) / 2; ++i) {
      int v0, v1, w;
      std::cin >> v0 >> v1 >> w;
      adyacencia[v0][v1] = w;
      adyacencia[v1][v0] = w;
   }

   // construcción del modelo

   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x[n][n];                    // ¿un arco conecta a i con j?
   for (int i = 0; i < n; ++i) {
```

```
   for (int j = 0; j < n; ++j) {
      x[i][j] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%d", i, j));
   }
}

for (int i = 0; i < n; ++i) {     // del vértice i sólo sale un arco
   GRBLinExpr ex;
   for (int j = 0; j < n; ++j) {
      ex += x[i][j];
   }
   modelo.addConstr(ex == 1);
}

for (int j = 0; j < n; ++j) {     // al vértice j sólo llega un arco
   GRBLinExpr ex;
   for (int i = 0;  i < n; ++i) {
      ex += x[i][j];
   }
   modelo.addConstr(ex == 1);
}

GRBVar potencial[n];              // potencial del vértice i (excepto el 0)
for (int i = 1; i < n; ++i) {
   potencial[i] = modelo.addVar(1, n - 1, 0, GRB_INTEGER, formato("p%d", i));
}

for (int i = 1; i < n; ++i) {     // restricción de potencial
   for (int j = 1; j < n; ++j) {
      modelo.addConstr(potencial[i] - potencial[j] + (n * x[i][j]) <= n - 1);
   }
}

GRBVar t = modelo.addVar(0, GRB_INFINITY, 0, GRB_CONTINUOUS, "t");
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      modelo.addConstr(t >= adyacencia[i][j] * x[i][j]);
   }
}

GRBLinExpr obj0, obj1 = t;
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      obj0 += adyacencia[i][j] * x[i][j];
   }
}
modelo.setObjectiveN(obj0, 0, 0);
modelo.setObjectiveN(obj1, 1, -1);

modelo.optimize( );
if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
   modelo.write("solucion.sol");
} else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
   std::cout << "Modelo no acotado\n";
} else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
```

```
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Estado no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```

Soluciones de 10.1

1. Escribe un programa que resuelva el problema del agente viajero en su versión euclidiana. Tu programa deberá generar una solución inicial mediante algún algoritmo de aproximación y después deberá usar el API de Gurobi para entregar dicha solución, construir el modelo y encontrar el óptimo. Puedes suponer que la instancia será descrita mediante un entero n seguido de n parejas de reales (x, y) que denotan las coordenadas en $\mathbb{R}^2$ de los n vértices de la gráfica.

Solución: Usaremos el algoritmo de Prim para construir el árbol abarcador de costo mínimo. En este algoritmo tendremos una cola de prioridad de aristas, la cual colocará la arista más barata al frente. Un vértice se considerará cubierto cuando agreguemos las aristas adyacentes a ese vértice en la cola. Comenzaremos el proceso cubriendo cualquier vértice (por ejemplo, el 0) y repetiremos el proceso a partir del extremo de la arista más barata identificada por la cola. Si la arista más barata de la cola conecta dos vértices cubiertos, ignoraremos dicha arista y revisaremos la siguiente. El algoritmo termina cuando todos los vértices alcanzables quedan cubiertos.

El árbol abarcador de costo mínimo se obtiene al anotar, para cada vértice, la arista más barata con la que llegamos a dicho vértice para cubrirlo. Luego recorreremos el árbol en profundidad para anotar el orden de visita de los vértices y construiremos una tabla que relacione el identificador del vértice con su posición en el orden (que será su potencial). Finalmente, construiremos el modelo matemático y le proporcionaremos la solución inicial a Gurobi.

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
#include <math.h>
#include <stdio.h>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

struct punto {
   double x, y;
};

double distancia(punto p1, punto p2) {
   return hypot(p1.x - p2.x, p1.y - p2.y);
}
```

```
struct arista {
   int origen, destino;
   double costo;
   bool operator<(const arista& a) const {
      return costo > a.costo;
   }
};

void recorre(int v, std::vector<int> adyacencia[], std::vector<int>& w) {
   w.push_back(v);
   for (int destino : adyacencia[v]) {
      recorre(destino, adyacencia, w);
   }
}

int main( ) try {
   // lectura de la entrada

   int n;
   std::cin >> n;

   punto arr[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i].x >> arr[i].y;
   }

   // cálculo de la solución inicial

   bool cubiertos[n] = { };
   std::priority_queue<arista> cola;
   cola.push({ -1, 0, 0 });
   std::vector<int> arbol[n];

   do {
      arista a = cola.top( );
      cola.pop( );
      if (!cubiertos[a.destino]) {
         cubiertos[a.destino] = true;
         if (a.origen != -1) {
            arbol[a.origen].push_back(a.destino);
         }
         for (int i = 0; i < n; ++i) {
            cola.push({ a.destino, i, distancia(arr[a.destino], arr[i]) });
         }
      }
   } while (!cola.empty( ));

   std::vector<int> orden, potencial_inicial(n, 0);
   recorre(0, arbol, orden);
   for (int i = 0; i < orden.size( ); ++i) {
      potencial_inicial[orden[i]] = i;
   }

   // construcción del modelo
```

```
GRBEnv ambiente;
GRBModel modelo(ambiente);

GRBVar x[n][n];                   // ¿un arco conecta a i con j?
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      x[i][j] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%d", i, j));
   }
}

for (int i = 0; i < n; ++i) {    // del vértice i sólo sale un arco
   GRBLinExpr ex;
   for (int j = 0; j < n; ++j) {
      ex += x[i][j];
   }
   modelo.addConstr(ex == 1);
}

for (int j = 0; j < n; ++j) {    // al vértice j sólo llega un arco
   GRBLinExpr ex;
   for (int i = 0;  i < n; ++i) {
      ex += x[i][j];
   }
   modelo.addConstr(ex == 1);
}

GRBVar potencial[n];              // potencial del vértice i (excepto el 0)
for (int i = 1; i < n; ++i) {
   potencial[i] = modelo.addVar(1, n - 1, 0, GRB_INTEGER, formato("p%d", i));
}

for (int i = 1; i < n; ++i) {    // restricción de potencial
   for (int j = 1; j < n; ++j) {
      modelo.addConstr(potencial[i] - potencial[j] + (n * x[i][j]) <= n - 1);
   }
}

GRBLinExpr objetivo;
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
       objetivo += distancia(arr[i], arr[j]) * x[i][j];
   }
}
modelo.setObjective(objetivo, GRB_MINIMIZE);

for (int i = 1; i < n; ++i) {    // establecimiento de la solución inicial
   potencial[i].set(GRB_DoubleAttr_Start, potencial_inicial[i]);
}

modelo.optimize( );
if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
   modelo.write("solucion.sol");
} else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
```

```
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Estado no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```

Soluciones de 11.1

1. Para los problemas cuyas soluciones factibles corresponden con permutaciones, podemos definir una heurística de búsqueda local como sigue. Definiremos la vecindad de una permutación como el conjunto de todas las permutaciones que se pueden obtener intercambiando alguna pareja de elementos de la permutacion original. El objetivo de la heurística será encontrar la mejor solución que se puede obtener examinando la permutación actual y sus vecinos.

   Escribe un programa que resuelva el problema del agente viajero en su versión euclidiana. Tu programa deberá usar la heurística descrita arriba para mejorar las soluciones intermedias que el solucionador encuentra. Si la heurística determina que una permutación vecina es mejor que la solución actual, entonces tu programa deberá entregar dicha permutación mejorada como nueva mejor solución.

Solución: La manera más sencilla (aunque no la más eficiente) de implementar la heurística descrita es definir una función que toma una permutación y que le aplica los $\binom{n}{2}$ intercambios posibles, evaluando la función objetivo para cada permutación resultante. De esta forma, la función podrá identificar la mejor de las permutaciones vistas (incluyendo a la original) y devolver su valor.

La implementación de la retrollamada se complica ligeramente porque no podremos recibir una solución y entregar su mejora en el mismo evento. Cuando Gurobi nos entregue una solución que sí logramos mejorar, tendremos que guardar la solución mejorada en el `struct`, en espera de poder entregarla en un evento de tipo `GRB_CB_MIPNODE`. Este desfase abre la posibilidad de que Gurobi notifique sobre una nueva solución antes de que hayamos podido entregar la mejora de una solución previa. Por esta razón, conviene almacenar no solo la solución mejorada sino también su valor, para así poder compararla con potenciales nuevas soluciones por parte de Gurobi. Es posible que incluso debamos tirar mejoras de soluciones previas que terminan siendo superadas en calidad por una nueva solución de Gurobi, o bien, por la mejora respectiva de esta última.

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <math.h>
#include <stdio.h>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[32 + 1];
   sprintf(bufer, s, v...);
   return std::string(bufer);
}

struct punto {
```

```
    double x, y;
};

double distancia(punto p1, punto p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);
}

double funcion_objetivo(int n, punto* arr, const std::vector<int>& orden) {
    double res = 0;
    for (int i = 0; i < n; ++i) {
        res += distancia(arr[orden[i]], arr[orden[(i + 1) % n]]);
    }
    return res;
}

double intenta_mejora(int n, punto* arr, std::vector<int>& orden, double valor) {
    std::vector<int> mejor = orden;
    for (int i = 1; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            std::swap(orden[i], orden[j]);
            double nuevo_valor = funcion_objetivo(n, arr, orden);
            if (nuevo_valor < valor) {
                valor = nuevo_valor;
                mejor = orden;
            }
            std::swap(orden[i], orden[j]);
        }
    }
    orden = mejor;
    return valor;
}

struct retrollamada : GRBCallback {
    int n;
    punto* arr;
    GRBVar* potencial;
    std::vector<int> orden_mejorado;
    double objetivo_mejorado = INFINITY;

    retrollamada(int t, punto* a, GRBVar* p)
    : n(t), arr(a), potencial(p) {
    }

    void callback( ) try {
        if (where == GRB_CB_MIPSOL) {
            double objetivo = getDoubleInfo(GRB_CB_MIPSOL_OBJ);
            if (objetivo < objetivo_mejorado) {
                objetivo_mejorado = INFINITY;
            }

            std::vector<int> orden(n, 0);
            for (int i = 1; i < n; ++i) {
                orden[int(std::round(getSolution(potencial[i])))] = i;
            }
```

```
         double nuevo_objetivo = intenta_mejora(n, arr, orden, objetivo);
         if (nuevo_objetivo < std::min(objetivo, objetivo_mejorado)) {
            objetivo_mejorado = nuevo_objetivo;
            orden_mejorado = orden;
         }
      } else if (where == GRB_CB_MIPNODE) {
         if (objetivo_mejorado != INFINITY) {
            std::cout << "Entregamos solucion de " << objetivo_mejorado << "\n";
            objetivo_mejorado = INFINITY;
            for (int i = 1; i < n; ++i) {
               setSolution(potencial[orden_mejorado[i]], i);
            }
         }
      }
   } catch (const GRBException& e) {
      std::cout << e.getMessage( ) << "\n";
   }
};

int main( ) try {
   // lectura de la entrada

   int n;
   std::cin >> n;

   punto arr[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i].x >> arr[i].y;
   }

   // construcción del modelo

   GRBEnv ambiente;
   GRBModel modelo(ambiente);

   GRBVar x[n][n];                    // ¿un arco conecta a i con j?
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         x[i][j] = modelo.addVar(0, 1, 0, GRB_BINARY, formato("x%d_%d", i, j));
      }
   }

   for (int i = 0; i < n; ++i) {     // del vértice i sólo sale un arco
      GRBLinExpr ex;
      for (int j = 0; j < n; ++j) {
         ex += x[i][j];
      }
      modelo.addConstr(ex == 1);
   }

   for (int j = 0; j < n; ++j) {     // al vértice j sólo llega un arco
      GRBLinExpr ex;
      for (int i = 0;  i < n; ++i) {
```

```
        ex += x[i][j];
      }
      modelo.addConstr(ex == 1);
   }

   GRBVar potencial[n];            // potencial del vértice i (excepto el 0)
   for (int i = 1; i < n; ++i) {
      potencial[i] = modelo.addVar(1, n - 1, 0, GRB_INTEGER, formato("p%d", i));
   }

   for (int i = 1; i < n; ++i) {    // restricción de potencial
      for (int j = 1; j < n; ++j) {
         modelo.addConstr(potencial[i] - potencial[j] + (n * x[i][j]) <= n - 1);
      }
   }

   GRBLinExpr objetivo;
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
          objetivo += distancia(arr[i], arr[j]) * x[i][j];
      }
   }
   modelo.setObjective(objetivo, GRB_MINIMIZE);

   retrollamada r(n, &arr[0], &potencial[0]);
   modelo.setCallback(&r);
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL) {
      modelo.write("tsp.sol");
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED) {
      std::cout << "Modelo no acotado\n";
   } else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE) {
      std::cout << "Modelo infactible\n";
   } else {
      std::cout << "Estado no manejado\n";
   }
} catch (const GRBException& e) {
   std::cout << e.getMessage( ) << "\n";
}
```