MA4702. Programación Lineal Mixta. 2020.
Profesor: José Soto
Escriba(s): Sebastian Bustos, Felipe Hernández y Nicolás Toro.
Fecha: 20 de marzo de 2020.

Cátedra 1

Introducción

En esta sesión se estudiarán los conceptos formales de lo que es un problema de optimización en una forma general, se definirán conceptos claves para el trabajo de problemas de optimización con diferentes restricciones y se presentan ejemplos que ilustran lo presentado.

1. Problemas y algoritmos (en optimización)

Comencemos con las definiciones esenciales:

Definición 1 (Problema de optimización e instancias)**.** Un *problema de optimización* $\mathcal{P}$ es un conjunto de *instancias*. Cada *instancia* $\mathcal{I}$ esta definida por:

- Un conjunto factible $S = \text{fact}(\mathcal{I})$.
- Una función a optimizar $f\colon S \to \mathbb{R}$.
- Un objetivo, minimizar o maximizar la función f en dicho conjunto S.

Usualmente las instancias se describen de manera implicita o compacta.

Ejemplo 1 (Árbol cubridor de peso mínimo – minimum spanning tree, MST)**.** Cada instancia del problema (MST) se describe de manera compacta indicando un grafo $G = (V, E)$ con pesos en las aristas $w\colon E \to \mathbb{R}_+$. De estos datos uno puede deducir el conjunto S de todos los árboles cubridores de G, y para cada árbol $T \in \mathcal{S}$, el valor de la función es la suma de los pesos de las aristas $w(T) = \sum_{e \in E[T]} w(e)$. El objetivo es minimizar la función de peso w:

$$\min_{T \in S} \; w(T)$$

Definición 2 (Algoritmo)**.** Un *algoritmo* para un problema de optimización $\mathcal{P}$ es un método que recibe una instancia $(S, f, \text{máx})$ y entrega, en un número finito de pasos:

1. El óptimo en caso de existir. Es decir un elemento $\text{OPT} \in S$ tal que $f(\text{OPT}) = \max_{x \in S} f(x)$.
2. Ó bien certifica que no existe elemento óptimo. Esto puede pasar cuando:
   a) El problema es infactible ($S = \emptyset$).
   b) El problema no es acotado $\left(\max_{x \in S} f(x) = \infty\right)$.
   c) El maximo no se alcanza (máx $\neq$ sup).

Diremos que un algoritmo que siga la estructura anterior resuelve el problema de optimización.

Ejemplo 2 (Kruskal – MST)**.** Un ejemplo de un algoritmo que resuelve el problema MST es Kruskal, modificándolo previamente para que certifique que no hay solución al problema si es que el grafo es disconexo.

Es importante notar que un algoritmo **tiene** que terminar en un tiempo finito (en la practica, el tiempo de ejecución puede ser tan grande, que no hay diferencia entre esto e "infinito").

2. Programas Lineales Mixtos

Definición 3 (Conjunto lineal mixto)**.** Se dice que $S \subseteq \mathbb{Z}^E \times \mathbb{R}^C \subseteq \mathbb{R}^n$ es un conjunto lineal mixto si S pude ser descrito como intersección de un conjunto finito de desigualdades lineales. Es decir S es:

$$S := \{x \in \mathbb{Z}^E \times \mathbb{R}^C : Ax \leq b\} \subseteq \mathbb{R}^n.$$

Con $n, m \in \mathbb{N}$, $E, C \subseteq [n]$ tales que $E \cup C = [n]$, $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ y $b \in \mathbb{R}^m$.

La interpretación y nombres en la definición de un conjunto lineal mixto viene dado por:

1. Las coordenadas en E se llaman *variables enteras*.
2. Las coordenadas en C se llaman *variables continuas*.
3. Las m desigualdades de la forma $a_j^T x \leq b_j$ se llaman *restricciones*.
4. Si $E = [n]$ a S se le llama *conjunto lineal entero*.
5. Si $C = [n]$ a S se le llama *conjunto lineal puro o poliedro*.
6. Si $x \in \{0, 1\}^n$ a S se le llama *conjunto lineal binario*.

Ejemplo 3. En la Figura 1 podemos ver un conjunto lineal puro en $\mathbb{R}^2$, el cual esta determinado por las inecuaciones:

$$\begin{pmatrix} 0 \\ 0 \\ y \end{pmatrix} \leq \begin{pmatrix} y \\ x \\ c - x \end{pmatrix},$$

donde $c > 0$. Mientras que en la Figura 2 observamos el conjunto lineal mixto obtenido añadiendo la restricción adicional $x \in \mathbb{Z}$.

Es importante recordar que sin perder generalidad se puede suponer que $Ax \leq b$ incluye todas las restricciones lineales de un problema de optimización (tanto las inecuaciones como las igualdades).

En la Figura 3 podemos observar un conjunto lineal binario, que cumple la restricción $x \in \{0, 1\}^3$

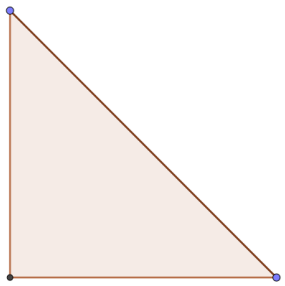


Figura 1: Conjunto lineal puro

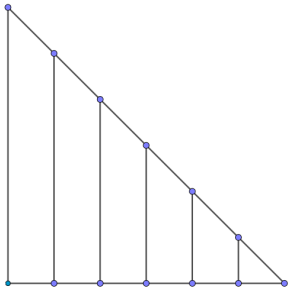


Figura 2: Conjunto lineal mixto
Eje x variable entera
Eje y variable continua

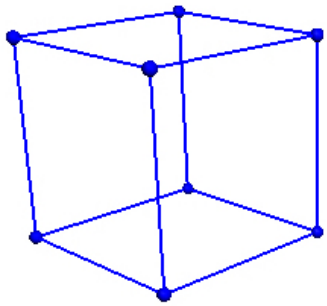


Figura 3: Conjunto lineal binario

Definición 4 (Programas lineales puros / mixtos / enteros / binarios)**.** Diremos que un problema de la forma:

$$\text{máx}\{c^t : x \in S\}$$

es un *programa lineal puro / mixto / entero / binario* si S es un conjunto lineal puro / mixto / entero / binario.

Se abreviaran como *PL / PLM / PLE / PLB* respectivamente.

Observación: A S se le conoce como dominio o conjunto factible del programa.

3. Modelos

En general los problemas de optimización que aparecen en la vida real no están escritos como un PLM. Por ende es necesario reescribir un problema para poder trabajarlo como tal.

Definición 5 (Modelo)**.** Diremos que un PLM es modelo de un problema si

- Cada solución óptima del PLM es solución óptima del problema original.
- Al menos una solución óptima del problema original es factible en el PLM

Es importante notar que a veces se relaja la primera condición y se pide que sea algún tipo de solución óptima en particular. La definición anterior da la oportunidad de que existan soluciones óptimas del problema original que no sean factibles en el PLM, en caso contrario nos encontramos frente a un modelo particular:

Definición 6 (Modelo exacto)**.** Diremos que un PLM es un modelo exacto si las soluciones factibles del problema original están en correspondencia uno a uno, de manera explicita con las soluciones factibles del PLM

Ejemplo 4 (Knapsack/ Problema de la mochila)**.** Dado $n \in \mathbb{N}$ objetos, valores $v_i \geq 0$ y tamaños $s_i \geq 0$. Dada una mochila de capacidad $B \geq 0$. Seleccionar un subconjunto a poner en la mochila maximizando el valor total.

$$\begin{aligned} \text{(Knapsack)} \quad \text{máx} \quad & \sum_{i\in[n]} v_i x_i \\ \text{s.a.} \quad & \sum_{i\in[n]} s_i x_i \leq B \\ & x_i \in \{0,1\} \ \forall i \in [n], \end{aligned}$$

donde

1. $x_i = 1 \iff i$ se encuentra en la mochila
2. $\sum_{i\in[n]} s_i x_i$ es la capacidad ocupada de la mochila.
3. $\sum_{i\in[n]} v_i x_i$ es el valor de la mochila.

Observación 1: Este modelo es exacto.

Observación 2: Es importante notar que a veces, por mucho que se conozcan técnicas para resolver un problema, es conveniente escribir el problema como un PLM, lo anterior permite la libertad de agregar restricciones fácilmente a los problemas de optimización.

Ejemplo 5 (s-t Camino de largo mínimo)**.** Dado el digrafo $G = (V, \vec{E})$, nodo de origen $s \in V$, nodo destino $t \in V$. Dada además una función de largos no negativos $\ell : \vec{E} \to \mathbb{R}_+$. Determinar un camino de largo mínimo de s a t. Para resolver este problema, necesitamos caracterizar de alguna manera los s-t caminos. Veremos dos maneras de realizar esto:

I.- (Mediante cortes) Sea $x \in \{0,1\}^E$ en donde $\forall e \in E,\ x(e) = 1 \iff$ el arco e esta siendo ocupado.
Recordemos la definición de corte:

Definición 7 (s-t corte)**.** Un subconjunto $U \subseteq V$ es un s-t corte si $s \in U, t \notin U$.

Tambien recordemos las siguientes definiciones:

Definición 8.

- $\delta^+(U)$ son los arcos que salen del conjunto U.
- $\delta^-(U)$ son los arcos que entran del conjunto U.

Para caracterizar que se sale de cada U s-t corte utilizaremos la siguiente ecuación:

$$x(\delta^+(U)) := \sum_{e\in\delta^+(U)} x(e) \geq 1$$

Así el problema lo podemos plantear como sigue:

$$\begin{aligned} \text{(SP-Conector)}\ \text{mín}\quad & \sum_{e\in E} \ell_e x_e \\ \text{s.a.}\quad & x\left(\delta^+(U)\right) \geq 1 \text{ para todo } s\text{-}t \text{ corte } U \\ & x \in \{0,1\}^E \end{aligned}$$

a) Se puede probar que si el conjunto de arcos cruza todos los cortes entonces tiene que haber un s-t camino.

b) Hay $|\mathcal{P}(V \setminus \{s,t\})| = 2^{n-2}$ cortes, y por ende 2^{n-2} restricciones.

Se dejan los siguientes ejercicios propuestos:

Ejercicio 1. Probar que para el problema s-t camino de coste mínimo y $\ell_e > 0\ \forall e$ entonces SP-Conector es un modelo. Y probar que (en general) no es un modelo exacto.

Ejercicio 2. Modificar SP-Conector para que incluso cuando ℓ pueda tomar valores nulos sea un modelo.

Es importante notar que la cantidad de restricciones crece de manera exponencial con el tamaño, luego, si bien el modelo soluciona el problema, en la practica el tiempo en que se ejecute el programa va a ser grande.

Por lo anterior es conveniente plantear el problema de otra manera.

II.- (Mediante flujos) Recordemos la definición de flujo:

Definición 9 (s-t flujo)**.** Una función $f : E \to \mathbb{R}_+$ es un s-t flujo si:

$$f\left(\delta^+(u)\right) = f\left(\delta^-(u)\right) \ \ \forall u \in V \backslash \{t, s\}$$

Luego el problema es posible plantearlo como:

$$\begin{aligned} \text{(SP-Flujo)} \ \text{mín} \quad & \sum_{e \in E} \ell_e x_e \\ \text{s.a.} \quad & x(\delta^+(u)) = x(\delta^-(u)) \ \forall u \in V \backslash \{t, s\} \\ & x(\delta^+(s)) = x(\delta^-(t)) = 1 \\ & x(\delta^-(s)) = x(\delta^+(t)) = 0 \\ & x \in \{0,1\}^E \end{aligned}$$

De esta manera existen $n + 2$ restricciones lo cual es una cantidad lineal de restricciones.

Ejercicio 3. Verifique que SP-Flujo es un modelo del problema cuando $\ell_e > 0 \ \forall e$ y muestre que, en general, no es un modelo exacto.

Ejercicio 4. Modifique SP-Flujo para que sea un modelo incluso cuando ℓ pueda tomar valores nulos.

Ejercicio 5. Pruebe que SP-Flujo es equivalente a

$$\begin{aligned} \text{mín} \quad & \sum_{e \in E} \ell_e x_e \\ \text{s.a.} \quad & x(\delta^+(u)) = x(\delta^-(u)) \ \forall u \in V \backslash \{t, s\} \\ & x(\delta^+(s)) - x(\delta^-(s)) = 1 \\ & x \in \{0,1\}^E \end{aligned}$$

Mostraremos en el curso que el problema es posible relajarse y en vez de considerar $x \in \{0,1\}^E$ se puede transformar al problema lineal puro con la restricción $0 \leq x \leq 1$.

MA4702. Programación Lineal Mixta. 2020.
Profesor: José Soto
Escriba(s): Sebastian Bustos, Felipe Hernández y Nicolás Toro.
Fecha: 20 de marzo de 2020.

Cátedra 1

Introducción

En esta cátedra se verán dos temas principales: se comenzará dando un recorrido por la historia de los algoritmos PL/PLM a modo de motivación, para después retomar el contenido de la clase pasada y estudiar más en detalle el concepto de las *formulaciones*.

4. Historia de Algoritmos PL/PLM

¿Por qué utilizamos algoritmos de PL/PLM sobre programación no lineal o convexa? Algunas razones son:

- Permiten modelar una gran variedad de problemas de la vida real con suficiente generalidad.
- Existen buenos algoritmos (a nivel práctico y teórico) y heurísticas[1] que permiten solucionarlas.
- Han tenido éxito industrial (existen solvers bien implementados).

Orígenes de la Programación Lineal Pura

- **1827 y 1836**: *Fourier* y *Motzkin* (por separado) describen un método **finito** para sistemas de desigualdades (conocido como *esquema de eliminación de Fourier-Motzkin*).
- **1937**:*Kantorovich* (Premio Nobel 1975) desarrolla una teoría de dualidad en programación lineal para temas de economía.
- **1946**: *George Dantzig* desarrolló la base del método Simplex mientras trabajaba para la fuerza aérea de Estados Unidos. Por otro lado, *Jon Von Neumann* desarrolla la teoría de la dualidad en programación lineal. Ambos junto a Kantorovich se consideran los padres de la programación lineal.

Definición 10 (Algoritmo Polinomial)**.** Se define como *algoritmo polinomial* a un algoritmo con entrada de N bits, y que ejecuta sus instrucciones en un orden de $\mathcal{O}(p(N))$, donde p es un polinomio. Es decir, un algoritmo con tiempo de ejecución $N^{\mathcal{O}(1)}$.

Definición 11 (Algoritmo Exponencial)**.** Se define como *algoritmo exponencial* a un algoritmo con entrada de N bits, y que ejecuta sus instrucciones en un orden de $\mathcal{O}(e^N)$. Es decir, por ejemplo, un algoritmo al cual le toma $2^{\mathcal{O}(N)}$ operaciones en ejecutarse.

Ejemplo 6. Un algoritmo que se tome como máximo N^{50} operaciones en ejecutarse, es un algoritmo polinomial[2].

Ejemplo 7. El algoritmo de Kruskal es polinomial.

La gran ventaja de este tipo de algoritmos es que si el tamaño de la instancia crece, por ejemplo, se duplica, entonces el tiempo de ejecución sólo se multiplica por una constante. Por ejemplo, un

[1] Una **heurística** es una técnica diseñada para resolver un problema más rápidamente cuando los métodos clásicos son demasiados lentos, o bien, encontrar una solución cercana al óptimo cuando los métodos clásicos fallan.
[2] Aunque no uno muy eficiente.

algoritmo con cota N^k en su tiempo de ejecución, al ser alimentado por una instancia de tamaño $2N$, su tiempo cambia a $(2N)^k = 2^k N^k$.

En cambio, si se está trabajando con un algoritmo exponencial, al duplicar el tamaño de la instancia, la complejidad del algoritmo podría crecer demasiado. Por ejemplo, para un algoritmo con cota 2^{cN} en su tiempo de ejecución, la cota para una instancia de tamaño $2N$ se eleva al cuadrado (pues $2^{2cN} = (2^{cN})^2$).

Definición 12 (Clase P)**.** La *clase P* contiene a aquellos problemas que son solubles en tiempo polinómico.

Ejemplo 8. El problema de calcular un matching es de clase P, pues existe una solución en tiempo polinomial.

Existe una clase de problemas, llamados NP-completos (cuya definición formal no daremos en este curso) que ocurren muchas veces en la práctica. Se conjetura (no ha sido demostrado) que todos estos problemas no están en P.

Observación 1. *La pregunta ¿$P = NP$? es aún un problema abierto y es uno de los problemas del milenio*[3]*.*

Se han definido estos conceptos de complejidad para responder a la siguiente pregunta: ¿Qué clase de problema es PL? la pregunta tiene su importancia, pues Klee y Minty (1973) demostraron que Simplex **no es polinomial**[4], a pesar de que es muy rápido en la mayoría de las instancias.

Entonces, ¿Es PL demasiado general como para no estar en P? La respuesta es: ¡No!, en efecto en el año 1979 se demuestra que **PL está en P**.

Programación Lineal Pura hoy en día

Ha sido trabajo de los matemáticos el de reducir el tiempo de ejecución del PL. En lo que sigue, n corresponde a variables y L al número de bits que se usan para codificar la entrada.

- **1979**: *Khachiyan* desarrolla el método de la elipsoide para PL (Complejidad $\mathcal{O}(n^6 L)$).
- **1984**: *Karmakar* desarrolla el método de punto interior (Barrera) para PL (Complejidad $\mathcal{O}(n^{3,5} L)$).
- **2015**: *Lee y Sidford* desarrollan la primera mejora en 20 años para flujo en redes (Complejidad $\mathcal{O}(n^{2,5} L)$).
- **2019**: *Cohen, Lee y Song* obtienen una complejidad de $\mathcal{O}(n^{2,37} L)$ usando multiplicación rápida de matrices.

Observación 2. *Notar que una complejidad de $\mathcal{O}(n^6 L)$ es bastante alta, pero al menos es polinomial. Por otro lado, $\mathcal{O}(n^{2,37} L)$ ya es una complejidad cercana a una cuadrática, el cuál es un orden más aceptable.*

Observación 3. *En la práctica, ninguno de estos algoritmos se utiliza (ya que son muy difíciles de implementar, y algunos resultan peores que un algoritmo exponencial en regímenes pequeños). Lo que se usa son mezclas de Simplex, Simplex-dual y Barrera.*

Respondiendo a la pregunta de por qué se prefiere PL/PLM, es porque hoy en día tenemos muchos métodos de resolución a estos problemas. De forma que, se ha avanzado tanto en esta área, que programas lineales que antes tomaban 20 años de procesar, hoy en día se pueden resolver en un segundo. Esto permite utilizar millones de variables y restricciones y resolverlos en un tiempo prudente.

[3] Es decir, si es que alguien demuestra o refuta la conjetura, ¡se ganará un millón de dolares!.

[4] Más precisamente, demostraron que hay instancias para los cuales Simplex debe tomar un número superpolinomial de pasos

¿Qué pasa con PLE y PLM?

- En la década de 1960 sale el primer solver comercial.
- Las siguientes décadas se dedican a crear mejoras en la implementación de PL.
- En la década de 1990 aparecen solvers nuevos, como *CPLEX* (ILOG), *MINTO* (Gerogia Tech) y *XpressMP* (comercial).
- En el año 2007 IBM compra ILOG.
- Los creadores originales de *CPLEX* no se involucran con IBM, y en su lugar fundan *Gurobi* en el año 2009.

Hoy en día, los problemas de programación lineal entera se pueden resolver de forma rápida debido a que se han incorporado una serie de ideas (que se verán más adelante en el curso) a los solvers previamente mencionados. Algunas de ellas son:

- Aproximación sucesiva de PLE por PL.
- Agregar planos cortantes, es decir, incluir restricciones sin perder puntos factibles (mejora en la formulación).
- *Branch and Bound*: es la base de los algoritmos PLE. La idea es que si hay un conjunto finito de puntos que pueden ser solución, se pueden enumerar de manera sistemática (como árbol) y es posible darse cuenta que hay ramas del árbol que no pueden ser solución.
- Planos cortantes genéricos: cortes que se generan automáticamente (Gomory, MIR, etc.)
- Presolver (Técnicas de reducción de tamaño).
- Generación de filas y columnas; Simetría; Heurísticas; entre otras.

Probablemente PLE no está en P (se puede probar que es NP-Completo). Sin embargo, los solvers disponibles actualmente son capaces de:

- Realizar buenas aproximaciones de soluciones de PLE con millones de variables y restricciones. Estas soluciones son factibles y pueden reportar que se encuentran cerca del óptimo. (para hacer esto, no se necesita que el usuario sea un experto, sólo basta con saber ocupar dichos solvers).
- Resolver a optimalidad muchas clases de subproblemas.
- Todo eso se hace prácticamente sin intervención del usuario.

5. Formulaciones

Definición 13 (Poliedro)**.** Un *poliedro* P es un conjunto lineal puro, es decir

$$P = P(A, b) = \{x \in \mathbb{R}^n : Ax \leq b\}$$

con $A \subseteq \mathbb{R}^{n \times m}$ y $b \in \mathbb{R}^n$

- Si A y b pueden ser escogidos racionales (es decir, sus componentes pueden ser escogidas racionales): P es un **poliedro racional**.
- Si P es acortado: se llama **polítopo**.

Observación 4. *Dado que los poliedros son intersecciones finitas de semiplanos, y los semiplanos son conjuntos convexos, entonces los poliedros son conjuntos convexos.*

Definición 14 (Formulación)**.** Un poliedro P es una *formulación* para un conjunto $S \subseteq \mathbb{Z}^E \times \mathbb{R}^C$ si:

$$S = P \cap (\mathbb{Z}^E \times \mathbb{R}^C)$$

En otras palabras, una formulación es un poliedro que aproxima externamente bien a algún conjunto.

Ejemplo 9. En la Figura 4, se puede observar un poliedro P (conjunto verde) que es una formulación de un conjunto S (puntos rojos), pues

$$S = P \cap \mathbb{Z}^E$$

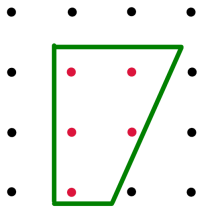


Figura 4: Primer ejemplo de una formulación.

Podemos notar que las formulaciones no son necesariamente únicas, en efecto, en el Ejemplo 9 se puede encontrar otro poliedro P', tal como se presenta en el siguiente ejemplo:

Ejemplo 10. En la imagen 5, se puede observar un poliedro P' (conjunto azul) que es una formulación de un conjunto S (puntos rojos), pues

$$S = P' \cap \mathbb{Z}^E$$

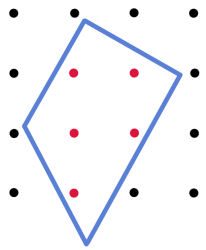


Figura 5: Segundo ejemplo de una formulación.

Dicho esto, nos podemos preguntar: ¿Cuál de todas las formulaciones ocupar? La respuesta que nace naturalmente es: ocupar la formulación minimal para S. Esto se puede lograr ocupando como formulación para S su envoltura convexa, vale decir, conv$S$. Una representación de lo anteriormente dicho se puede observar en la siguiente figura:

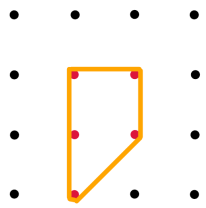


Figura 6: Formulación minimal para S: su envoltura convexa.

Formalizando lo comentado anteriormente en los ejemplos, P es buena formulación de S si es que lo *aproxima* apropiadamente. Para empezar, al menos, sabemos que debemos tener lo siguiente

$$S \subseteq P$$

- Sean $P_1 \subseteq P_2$ dos formulaciones para el mismo conjunto. En este contexto, diremos que P_1 es *mejor* que P_2.

  Posterior a esto, no es difícil pensar que el mejor de todos los casos será tomar como formulación la envoltura convexa (esto puede, y debería, ser demostrado). Justamente por eso tenemos lo siguiente

- Si $P = \text{conv}(S)$ entonces P es *formulación ideal* de S.

Observación 5. *No todos los conjuntos lineales mixtos admiten formulaciones ideales.*

En la siguiente figura se puede apreciar un ejemplo de lo anterior, pues, para la primera restricción (gráficamente, la diagonal) no hay valores que se alcancen en la frontera. Esto, pues habría que solucionar $-\sqrt{2}x + y = 0$ con $x,\ y$ enteros. Sin embargo, por la irracionalidad de $\sqrt{2}$ no habrá un punto (x, y) que resuelva tal ecuación. Lo que sucederá es que iremos tomando una cantidad infinita de puntos que se acerquen, pero eso no define un poliedro. Así, se evidencia que hay conjuntos que no admiten formulación ideal.

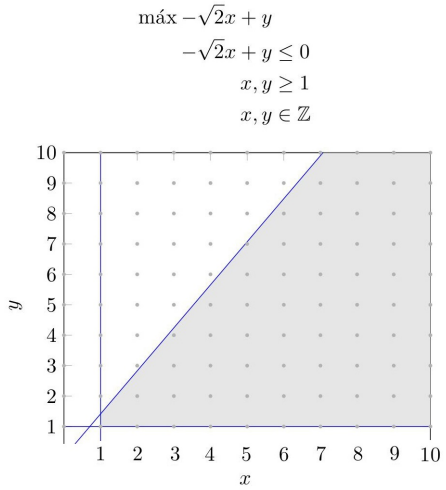


Figura 7: Ejemplo de PLM donde no existe formulación ideal.

Notemos que lo anterior no sucede para un PL puro, recordando del curso de optimización, que todo PL factible y con objetivo acotado alcanza su óptimo.

Teorema 1. *Sea (M) un PLM con datos racionales. Si su relajación lineal (P) es acotado en la dirección de optimización, entonces o bien (M) tiene óptimo finito racional y alcanzable, o bien (M) es infactible.*

Ejemplo 11 (Uncapacitated Facility Location)**.** Un problema clásico en Investigación de Operaciones es el de *Uncapacitated Facility Location*. Definimos ahora lo que se recibe como información y lo que busca hallar el problema.

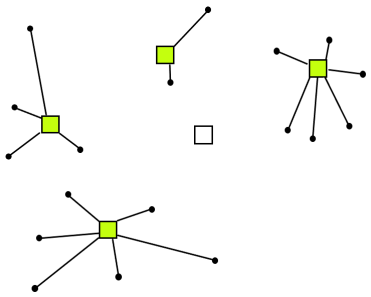


Figura 8: Visualización del problema *Uncapacitated Facility Location.*

Entrada: n clientes y m ubicaciones de bodegas. Existe un costo fijo $c_j \geq 0,\ \forall j \in [m]$ asociado a la apertura de una bodega y un costo de conexión $d_{ij} \geq 0$ por conectar a un cliente i con una bodega j, $\forall i \in [n]$, $\forall j \in [m]$.

Objetivo: Abrir bodegas y conectar clientes a bodegas abiertas incurriendo en un costo total mínimo.

Construiremos una formulación para el problema:

Para ello haremos uso de dos variables binarias:

$$y_j = \begin{cases} 1 \text{ si la bodega } j \text{ abre.} \\ 0 \text{ si no lo hace} \end{cases} \qquad x_{ij} = \begin{cases} 1 \text{ si el cliente } i \text{ conecta con la bodega } j \\ 0 \text{ si no lo hace} \end{cases}$$

De esta forma, procederemos a plantear la formulación. Queremos minimizar el costo total, así, el objetivo será:

$$\text{(FL1)} \qquad \text{mín} \sum_{j \in [m]} c_j y_j + \sum_{i \in [n]} \sum_{j \in [m]} x_{ij} d_{ij}$$

El primer término será el asociado a los costos fijos de las bodegas y el de la derecha el costo de conexión cliente-bodega total.

Ahora, las restricciones serán las siguientes:

$$\begin{aligned} s.a. \quad & \sum_{j \in [m]} x_{ij} = 1 && \forall i \in [n] \\ & \sum_{i \in [n]} x_{ij} \leq n \cdot y_j && \forall j \in [m] \\ & x_{ij},\ y_j \in \{0,1\} && \forall i \in [n]\ \forall j \in [m] \end{aligned}$$

La primera restricción es la que indica que de todas las bodegas, el cliente *debe* estar asociado a una. (Podría ser reemplazado por la restricción de estar asociado al menos con una). La segunda viene de la siguiente situación:

- Si la bodega no abre ($y_j = 0$) no pueden haber clientes asociados a esa bodega, luego, como los x_{ij} son no negativos, deben ser todos cero.
- Si la bodega abre ($y_j = 1$) entonces pueden haber como máximo n clientes asociados a esa bodega (pues ese es el número total de clientes)

La tercera restricción viene del hecho de que definimos x_{ij} y y_j como variables binarias.

El objetivo junto a las tres restricciones definen la formulación que hemos construido.

Ejemplo 12 (Problema de Asignación)**.** Un problema al que probablemente usted ha podido enfrentarse en Algoritmos Combinatoriales es el *problema de asignación.*

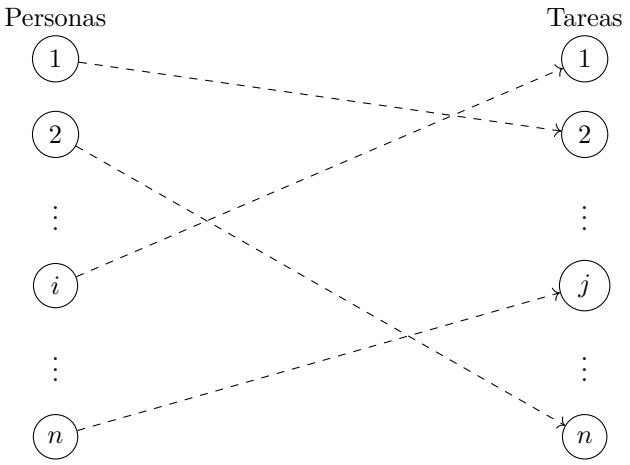


Figura 9: Problema de Asignacion.

Entrada: n tareas y n personas. Cada persona debe realizar *exactamente* una tarea. c_{ij} es el costo incurrido por la persona i al realizar la tarea j.

Objetivo: Encontrar la asignación de costo total mínimo.

Construiremos una formulación para el problema:

Para ello haremos uso de una variables binaria:

$$x_{ij} = \begin{cases} 1 \text{ si la persona } i \text{ realiza la tarea } j \\ 0 \text{ si no lo hace} \end{cases}$$

Así, procedemos. Queremos minimizar el costo total, así, el objetivo será:

$$(A) \qquad \text{mín} \sum_{i \in [n]} \sum_{j \in [n]} c_{ij} x_{ij}$$

El término proviene de sumar los costos de las personas (suma sobre i) asociadas a las tareas (suma sobre j), los cuales contaremos solo si la persona i realiza la tarea j.

Ahora, las restricciones serán las siguientes:

$$\begin{aligned} s.a. \quad & \sum_{j \in [n]} x_{ij} = 1 \quad \forall i \in [n] \\ & \sum_{i \in [n]} x_{ij} = 1 \quad \forall j \in [n] \\ & x_{ij}, \in \{0,1\} \quad \forall i \in [n] \; \forall j \in [n] \end{aligned}$$

El objetivo junto a las tres restricciones definen la formulación que hemos construido.

Ahora, definiremos lo siguiente:

- S: Dominio de (A)
- D: Relajación de (A)

Gracias a un importante resultado de Birkhoff y Von Neumann sabemos que todo punto en P es combinación convexa de puntos enteros (en S), Luego, P es ideal.

Para finalizar la clase, se presenta un caso interesante.
Sea (M) un PLE (totalmente) acotado, con dominio S. Luego S es finito. ¿Tiene formulación ideal?

$$S = \{s_i\}_{i=1}^{K}$$
$$\text{conv}(S) = \{x = \sum_{i=1}^{K} \lambda_i s_i : \lambda_i \geq 0,\ \sum_{i=1}^{K} \lambda_i = 1\}$$

Surge así la pregunta... ¿La envoltura convexa de un conjunto finito de puntos es un poliedro? (esto puede ser respondido a través del estudio de teoría poliedral).

MA4702. Programación Lineal Mixta. 2020.
Profesor: José Soto
Fecha: 20 de marzo de 2020.

Cátedra 1

6. Descripción de Branch and Bound

Uno de los métodos más usados para resolver PLM y otros problemas de optimización es **branch and bound (BnB)**. Se basa en la siguiente idea:

> Sea $\{S_1, \ldots, S_k\}$ una partición del conjunto factible S de un problema de optimización $z^* = \max\{c^T x \colon x \in S\}$. Si $z_i^* = \max\{c^T x \colon x \in S_i\}$ entonces $z^* = \max_{i \in [k]} z_i^*$.

BnB consiste en particionar el conjunto factible S del problema original en conjuntos más pequeños y resolver luego $\max c^T x$ en cada subconjunto de manera recursiva. Si la recursión se pudiera llevar completamente, al final enumeraríamos todos los puntos factibles del problema. Esta idea tiene dos problemas. Primero, si el dominio es infinito entonces esto no es posible. Segundo, incluso si el dominio es finito, esto podría ser extremadamente lento y no difiere en nada de simplemente de simple fuerza bruta.

La idea es que BnB no explore todo el árbol de recursión sino que guarde cotas para los subproblemas que ya ha resuelto y, usando estas cotas determinar que no necesitamos resolver ciertos subproblemas.

En lo que sigue nos enfocaremos en PLM de la forma

$$(M) \quad \max c^T x$$
$$S\colon \begin{cases} Ax & \leq b \\ x & \in \mathbb{Z}^E \times \mathbb{R}^C \end{cases}$$

donde A, b, c son racionales. Llamemos $P = \{x \in \mathbb{R}^{E \times C}, Ax \leq b\}$ a la relajación lineal natural de S, luego $S \subseteq P$. Como los datos son racionales tenemos que el programa lineal relajado

$$(L) \quad \max c^T x$$
$$P\colon \begin{cases} Ax & \leq b \\ x & \in \mathbb{R}^{E \cup C} \end{cases}$$

es o bien **infactible**, o bien **factible no acotado** o bien **factible acotado** con solución óptima $\bar{x} \in P$ racional. En este caso llamamos $\bar{z} = c^T \bar{x}$ a su valor.

Llamemos (M_0) al problema inicial dado y **pediremos que** (P_0) **sea factible acotado**. Hacemos esto pues BnB *a secas* no es capaz de lidiar con problemas con relajación no acotada en la dirección de optimización.

Para resolver (M_0) se construye **iterativamente** un árbol $\mathcal{T}$ cuyos nodos son de la forma (M, B, s) con (M) un subproblema de (M_0), B una cota superior (optimista) del valor de (M_0), y s un **status** que puede ser **activo**, **ramificado**, **infactible**, **dominado** o **entero**. Tanto la cota como el status de un nodo puede cambiar a lo largo del algoritmo.

En el árbol siempre se tendrá que **la unión de los dominios de las hojas** es el **dominio** de la raíz. Además se mantiene globalmente una solución factible (inicialmente nula) x^* llamada **incumbente** que resulta ser la mejor solución encontrada hasta ahora y $z^* = c^T x^*$ su valor (también llamada

mejor cota inferior). Inicialmente, el nodo raíz es $(M_0, B_0, \text{activo})$ donde M_0 es el PLM original y B_0 es el valor $\bar{z}$ de su relajación. Además, x^* es nulo y $z^* = -\infty$. Durante el algoritmo BnB hay 2 procesos importantes: la **creación de un nodo** y la **ramificación de un nodo activo**.

Algorithm 1 Creación de un nodo (M)

1: Resolver la relajación lineal (P) de (M).
2: **if** (P) es infactible **then return** $(M, -\infty, \text{infactible})$.
3: **if** el óptimo $\bar{x}$ de (P) es factible para (M) **then**
4: **if** $\bar{z} \leq z^*$ **then return** $(M, \bar{z}, \text{entero})$
5: **if** $\bar{z} > z^*$ **then** $x^* \leftarrow \bar{x}$, $z^* \leftarrow \bar{z}$, **return** $(M, \bar{z}, \text{entero})$ ▷ actualizar incumbente
6: **if** el óptimo $\bar{x}$ de (P) es infactible para (M) **then**
7: **if** $\bar{z} \leq z^*$ **then return** $(M, \bar{z}, \text{dominado})$
8: **if** $\bar{z} > z^*$ **then return** $(M, \bar{z}, \text{activo})$

Notamos que si un nodo se declara entero, entonces conocemos su mejor valor factible. Si un nodo se declara dominado en su dominio S no pueden haber soluciones enteras mejores que el incumbente, por lo que no es necesario seguir procesándolo, al igual que si el nodo se declara infactible. Los únicos problemas que podrían tener soluciones enteras mejores que el incumbente actual son aquellos que están activos.

Por otro lado, ramificar un nodo activo $(M, B(M), \text{activo})$ consiste en:

Algorithm 2 Ramificar nodo $(M, B(M), \text{activo})$

1: Determinar $k \geq 2$ subproblemas (M_i) tales que la unión de sus dominios es el dominio de M.
2: Crear un nodo (M_i) para cada subproblema y colgarlo como hijo de (M).
3: Declarar el status de (M) como ramificado.

Hay varias formas de ramificar un nodo. Una forma estándar y simple es hacer **branching en una variable dada**.

Branching en una variable x_k Como el óptimo $\bar{x} \in P$ de (P) no es factible en (M) debe haber una coordenada $k \in E$ tal que la variable $\bar{x}_k$ es fraccional (pero que debería ser entera en (M)). Podemos **elegir** una coordenada y definir entonces dos PLM nuevos (M_1) y (M_2) como sigue:

$$S_1 = S \cap \{x\colon x_k \leq \lfloor \bar{x}_k \rfloor\}. \qquad S_2 = S \cap \{x\colon x_k \geq \lceil \bar{x}_k \rceil\}.$$
$$P_1 = P \cap \{x\colon x_k \leq \lfloor \bar{x}_k \rfloor\}. \qquad P_2 = P \cap \{x\colon x_k \geq \lceil \bar{x}_k \rceil\}.$$
$$(M_1)\colon \text{máx}\{c^T x\colon x \in S_1\}. \qquad (M_2)\colon \text{máx}\{c^T x\colon x \in S_2\}.$$
$$(L_1)\colon \text{máx}\{c^T x\colon x \in P_1\}. \qquad (L_2)\colon \text{máx}\{c^T x\colon x \in P_2\}.$$

Esto satisface las condiciones de la ramificación anterior. Ahora estamos listos para escribir el algoritmo de BnB completo. Como BnB es un método genérico hay algunas instrucciones (en rojo) que no están completamente descritas.

Algorithm 3 BnB

Ensure: Un PLM (M_0) **racional** con relajación (L_0) factible acotada.

$x^* \leftarrow$ NULL, Crear nodo (M_0) como raíz del árbol $\mathcal{T}$.
while existan nodos activos en $\mathcal{T}$ **do**
    Si se ha alcanzado un criterio de terminación temprana parar
    Elegir un nodo activo (M, B, activo) y ramificarlo.
    Actualizar las cotas $B(M')$ para cada nodo (M') en el camino entre (M) y la raíz (M_0), i.e.
        $B(M') \leftarrow \min\{B(M'), \max\{B(\tilde{M}) \colon (\tilde{M}) \text{ es hijo de } (M')\}\}$
    Si el incumbente cambió, **actualizar** todos los nodos activos que ahora estén dominados, i.e.
        Declarar todo (M', B', activo) con $B' \leq z^*$ como dominado.
Return x^*.

Discutiremos luego criterios de terminación temprana. Pero si en algún minuto necesitamos terminar, entonces observamos que el valor óptimo de M_0 siempre está en $[z^*, B(M_0)]$. La razón $\frac{B(M_0)-z^*}{z^*}$ se suele llamar el **GAP** de resolución (en dicho momento). Hagamos un ejemplo concreto de BnB usando solo branchings por variables.

$$M_0: \quad \max 3x + 5y$$

$$S_0: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases}$$

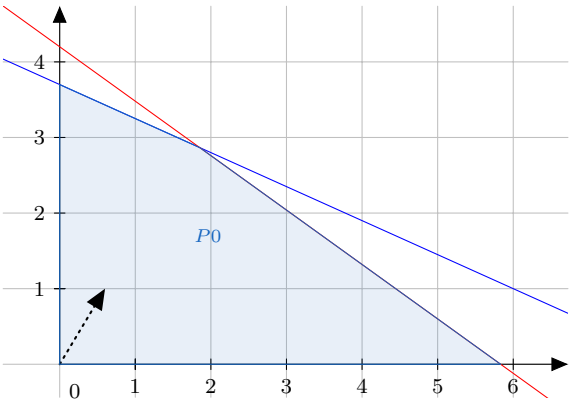
```
modelo= Model()
set_optimizer(modelo, Gurobi.Optimizer)
set_optimizer_attributes(modelo, "Presolve" => 0,
"OutputFlag" => 0)
@variable(modelo, x>=0, base_name="x")
@variable(modelo, y>=0, base_name="y")
@constraint(modelo, rest1, 20y + 9x<=74)
@constraint(modelo, rest2, 25y+ 18x<=105)
@objective(modelo, Max, 3x+5y)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo)," x=",
  value(modelo[:x])," y=", value(modelo[:y]))

z=19.799999999999997 x=2.0 y=2.76
```

Al resolver la relajación lineal anterior (por ejemplo con el solver GUROBI mediante JULIA, abajo), encontramos que el óptimo del PL asociado es $(x_0, y_0) \approx (1.8518, 2.866)$, $z \approx 19.888$ que no es entero. Luego el nodo raíz M_0 queda activo, con cota superior $B = 19.888$.

Estudiemos el árbol que BnB produce, por simplicidad anotemos en cada nodo además el punto fraccional óptimo de la relajación. Escribamos además sobre la raíz los datos actuales del incumbente.

$(x^*, y^*) = \text{NULL}, z^* = -\infty$

$(M_0, B = 19.888, \text{ACT})$
$(1.8518, 2.866), z_0 = 19.888$



Como ambas coordenadas de (x_0, y_0) son fraccionales **podemos elegir una**, digamos x y **ramificar** de acuerdo a dicha variable, creando dos problemas M_1 y M_2, con conjuntos factibles

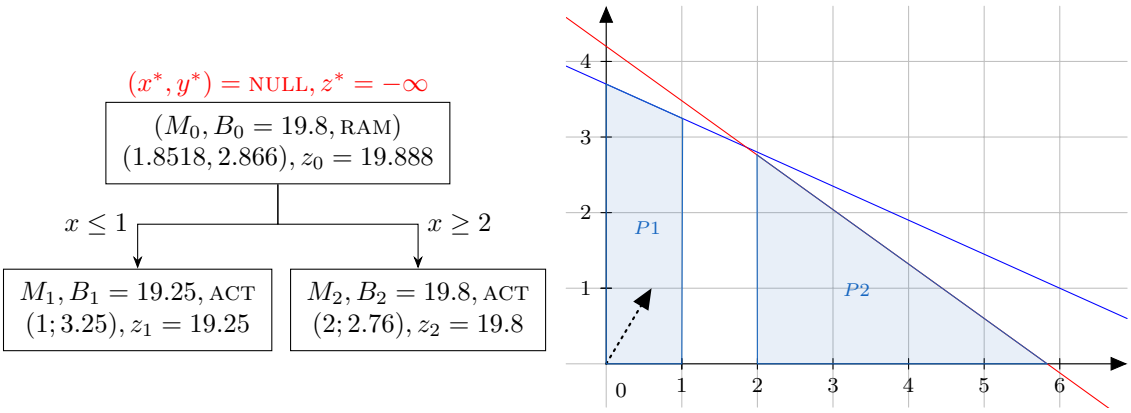
$$S_1 = S_0 \cap \{x \leq \lfloor x_0 \rfloor\} \text{ y } S_1 = S_0 \cap \{x \geq \lceil x_0 \rceil\}$$

$$M_1: \quad \text{máx } 3x + 5y \qquad M_2: \quad \text{máx } 3x + 5y$$

$$S_1: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x & \leq 1 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases} \qquad S_2: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x & \geq 2 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases}$$

```
#modelo M1
set_upper_bound(modelo[:x], 1)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo),
 " x=", value(modelo[:x]),
 " y=", value(modelo[:y]))

z=19.25 x=1.0 y=3.25

#modelo M2
delete_upper_bound(modelo[:x])
set_lower_bound(modelo[:x], 2)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo),
 " x=", value(modelo[:x]),
 " y=", value(modelo[:y]))

z=19.799999999999997 x=2.0 y=2.76
```

La solución óptima de la relajación de M_1 es $(x_1, y_1) = (1; 3.25)$ de valor $z_1 = 19.25$, y la solución óptima de la relajación de M_2 es $(x_2, y_2) = (2; 2.76)$ de valor $z_2 = 19.8$. Luego ambos nodos se crean activos. Más aún la cota de M_0 se actualiza a $\text{máx}\{19.25, 19.8\} = 19.8$. Por lo que el nuevo árbol de BnB (y su diagrama actual) se ven asi:



Resulta útil anotar en las aristas del árbol que restricciones hemos agregado en cada problema. Ahora hay 2 nodos activos en el árbol y podemos elegir cualquiera para ramificar. La solución (x_1, y_1) tiene variable y fraccional. Ramifiquemos M_1 en dos problemas M_3 y M_4 de acuerdo a si $y \leq 3$ o si $y \geq 4$. Obtenemos:

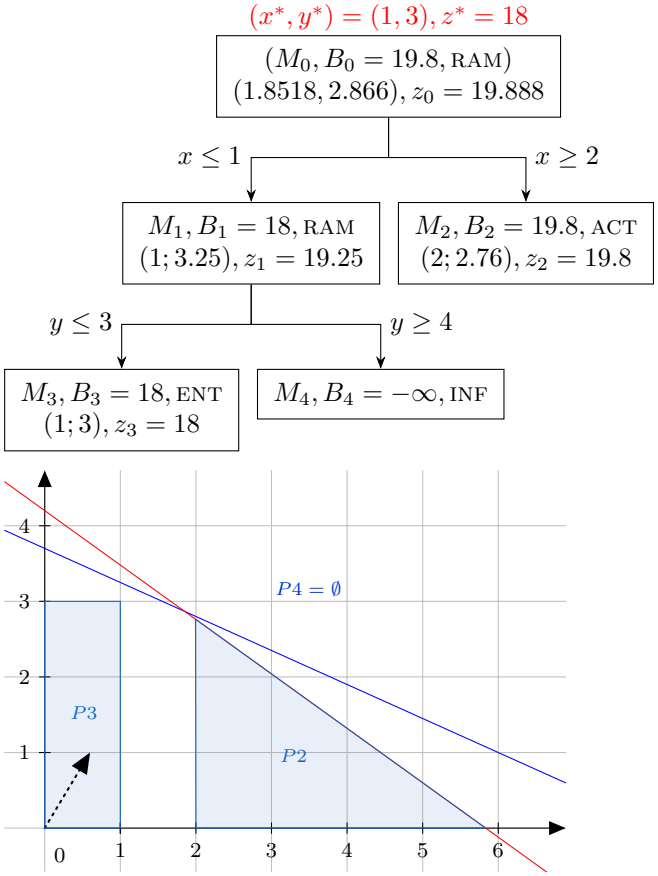
$$M_3: \quad \text{máx } 3x + 5y \qquad M_4: \quad \text{máx } 3x + 5y$$

$$S_3: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x & \leq 1 \\ y & \leq 3 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases} \qquad S_4: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x & \leq 1 \\ y & \geq 4 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases}$$

```
#Modelo M3
set_lower_bound(modelo[:x],0)
set_upper_bound(modelo[:x],1)
set_upper_bound(modelo[:y],3)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo),
 " x=", value(modelo[:x]),
 " y=", value(modelo[:y]))

z=18.0 x=1.0 y=3.0

#Modelo M4
delete_upper_bound(modelo[:y])
set_lower_bound(modelo[:y],4)
optimize!(modelo)
termination_status(modelo)

INFEASIBLE::TerminationStatusCode = 2
```

La solución de la relajación de M_3 es entera $(x_3, y_3) = (1, 2)$ con $z_3 = 18$. Por lo que M_3 se declara entero y además, (x_3, y_3) se transforma en incumbente (actualizando también z^*). Mientras tanto la relajación de M_4 es infactible, por lo que su cota es $-\infty$. Actualizamos la cota superior de M_1 a 18, mientras que la cota de M_0 se mantiene.

Nuestro **árbol de BnB** se ve actualmente así.



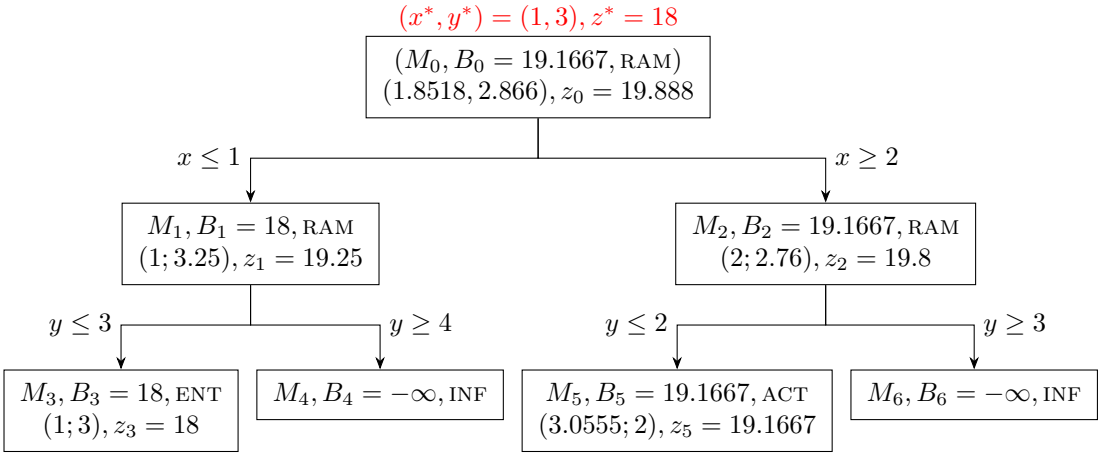
Como tenemos incumbente y cota, el **gap** de nuestra solución actual es $\frac{19.8-18}{18} = \frac{1.8}{18} = 0.1 = 10\,\%$.
Solo queda M_2 activo, lo ramificamos en M_5 y M_6, donde $y \leq$ o $y \geq 3$ respectivamente.

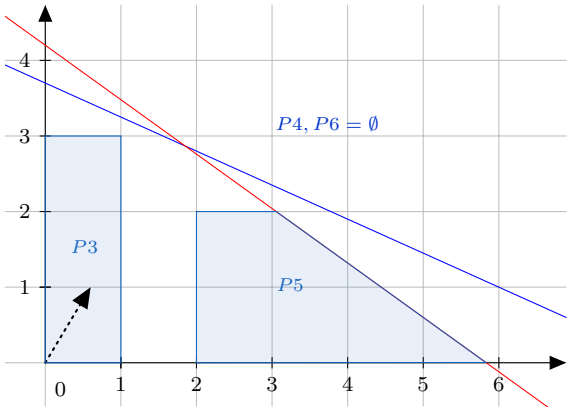
$$M_5: \quad \text{máx } 3x + 5y \qquad M_6: \quad \text{máx } 3x + 5y$$

$$S_5: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x & \geq 2 \\ y & \leq 2 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases} \qquad S_6: \begin{cases} 20y + 9x & \leq 74 \\ 25y + 18x & \leq 105 \\ x & \geq 2 \\ y & \geq 3 \\ x, y & \geq 0 \\ x, y & \in \mathbb{Z} \end{cases}$$

```
#Modelo M5
set_lower_bound(modelo[:x],2)
delete_upper_bound(modelo[:x])
set_upper_bound(modelo[:y],2)
set_lower_bound(modelo[:y],0)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo),
 " x=", value(modelo[:x]),
 " y=", value(modelo[:y]))

z=19.166666666666664 x=3.0555555555555554 y=2.0

#Modelo M6
delete_upper_bound(modelo[:y])
set_lower_bound(modelo[:y],3)
optimize!(modelo)
termination_status(modelo)

INFEASIBLE::TerminationStatusCode = 2
```

Notamos que M_6 es infactible, y la relajación de M_5 tiene solución $(3.0555; 2)$ con valor $z_5 = 19.1667$ por lo que queda activo (y podemos actualizar las cotas). Actualizamos el árbol BnB como sigue:





Nuestro gap mejoró a $(19.1667 - 18)/18 = 6.48\,\%$. Nuevamente tenemos solo un nodo activo, M_5. Lo ramificamos en x, definiendo los problemas M_7 y M_8 con $x \leq 3$ y $x \geq 4$ respectivamente:

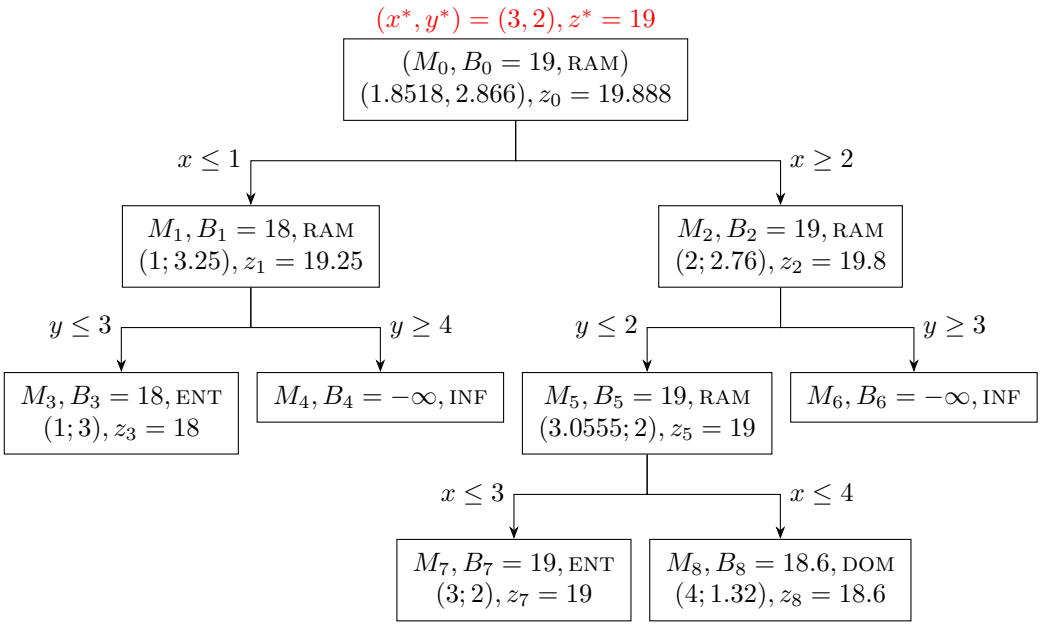
$$M_7: \quad \max 3x+5y \qquad M_8: \quad \max 3x+5y$$

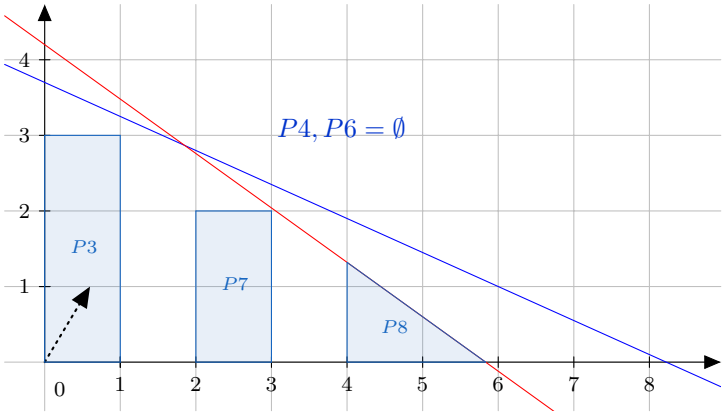
$$S_7: \begin{cases} 20y+9x & \leq 74 \\ 25y+18x & \leq 105 \\ x & \geq 2 \\ x & \leq 3 \\ y & \leq 2 \\ x,y & \geq 0 \\ x,y & \in \mathbb{Z} \end{cases} \qquad S_8: \begin{cases} 20y+9x & \leq 74 \\ 25y+18x & \leq 105 \\ x & \geq 4 \\ y & \geq 2 \\ x,y & \geq 0 \\ x,y & \in \mathbb{Z} \end{cases}$$

```
#Modelo M7
set_lower_bound(modelo[:x],0)
set_upper_bound(modelo[:x],3)
set_upper_bound(modelo[:y],2)
set_lower_bound(modelo[:y],0)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo),
 " x=", value(modelo[:x]),
 " y=", value(modelo[:y]))

z=19.0 x=3.0 y=2.0

#Modelo M8
delete_upper_bound(modelo[:x])
set_lower_bound(modelo[:x],4)
optimize!(modelo)
termination_status(modelo)

OPTIMAL::TerminationStatusCode = 1

println("z=",objective_value(modelo),
 " x=", value(modelo[:x]),
 " y=", value(modelo[:y]))

z=18.6 x=4.0 y=1.32
```

Al resolver la relajación de M_7 se obtiene una solución **integral** $(x_7, y_7) = (3, 2)$ con $z_7 = 19$. Como 19 es mayor que nuestro valor incumbente actual, éste se actualiza. Por otro lado al resolver la relajación de M_8 se obtiene una solución fraccional $(x_8, y_8) = (4; 1.32)$ con $z_8 = 18.6$. Pero esta vez z_8 es peor que el valor incumbente (19), por lo que se declara **dominado** (o podado por cota).

Con esto hemos completado el árbol de BnB y obtenemos que la solución óptima es $(3, 2)$ de valor 19.





Hay muchos solvers comerciales que realizan BnB de manera muy eficiente (eligen buenos nodos activos para ramificar, hacen branching en buenas variables, etc.) y de hecho aplican varias heurísticas y otros algoritmos que aceleran aún más su ejecución. Por completitud, abajo anotamos una serie de instrucciones para ejecutar GUROBI sobre nuestro modelo (y dando directrices de modo que el orden de ramificado, etc. sea el mismo que nosotros usamos en el ejemplo anterior):

```
modelonuevo= Model()
@variable(modelonuevo, x>=0)
@variable(modelonuevo, y>=0)
@constraint(modelonuevo, rest1, 20y + 9x<=74)
@constraint(modelonuevo, rest2, 25y+ 18x<=105)

#función objetivo
@objective(modelonuevo, Max, 3x+5y)

#declaramos las variables como enteras
set_integer(modelonuevo[:x])
set_integer(modelonuevo[:y])

#Le indicamos a JuMP que el solver a utilizar es Gurobi y eliminamos presolver
set_optimizer(modelonuevo, Gurobi.Optimizer)
set_optimizer_attributes(modelonuevo, "Presolve" => 0, "Heuristics"=> 0,
 "Cuts"=> 0, "Threads" => 1, "BranchDir" => -1)

#declaramos atributos (branch primero en x luego en y)
MOI.set(modelonuevo, Gurobi.VariableAttribute("BranchPriority"), x, 20)
```

```
MOI.set(modelonuevo, Gurobi.VariableAttribute("BranchPriority"), y, 1)

#resolver
optimize!(modelonuevo)
```

Esto produce:

```
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.1 build v9.0.1rc0 (win64)
Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0x71254e4e
Variable types: 0 continuous, 2 integer (0 binary)
Coefficient statistics:
  Matrix range     [9e+00, 3e+01]
  Objective range  [3e+00, 5e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [7e+01, 1e+02]
Variable types: 0 continuous, 2 integer (0 binary)

Root relaxation: objective 1.988889e+01, 2 iterations, 0.00 seconds

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

     0     0   19.88889    0    2          -   19.88889      -     -    0s
     0     0   19.88889    0    2          -   19.88889      -     -    0s
     0     2   19.80000    0    2          -   19.80000      -     -    0s
*    1     1               1      18.0000000   19.80000  10.0%   2.0    0s
*    2     0               1      19.0000000   19.00000  0.00%   2.0    0s


Explored 3 nodes (6 simplex iterations) in 0.03 seconds
Thread count was 1 (of 8 available processors)

Solution count 2: 19 18

Optimal solution found (tolerance 1.00e-04)
Best objective 1.900000000000e+01, best bound 1.900000000000e+01, gap 0.0000%
```

Gurobi realiza un árbol de BnB parecido al nuestro, pero es más eficiente en su implementación: cada vez que un nodo se ramifica en 2 y uno de ellos es infactible o dominado, continúa bajando por el árbol (es decir no crea nuestros nodos 1, 4, 2, 6, 8) hasta que deja un par de nodos activos. Otros solvers realizan otras variantes de BnB.

7. Algunas consideraciones para hacer BnB eficiente

Resolución de relajación lineal en cada nodo. Un detalle interesante a considerar es que en general el PL (P) asociado a un nodo es muy similar al PL asociado al de su padre (son solo restricciones extras) por lo que podemos resolver (P) de manera más eficiente a partir de la solución óptima del padre y mediante **simplex dual**.

Selección de nodos activos El rendimiento de BnB depende fuertemente de la manera como se elige el nodo activo para ramificar (cuando hay varios de estos). Aquí hay dos objetivos que compiten: (1) Encontrar rápidamente un incumbente (2) Acotar rápidamente la cota B_0 del nodo raíz.

Algunas estrategias estándar para seleccionar nodos son:

1. Búsqueda en profundidad (DFS). *Ventajas*: apunta a encontrar un incumbente rápidamente con pocos nodos. *Desventajas*: puede explorar un área *mala* del árbol con nodos sin buenas soluciones.
2. Desarrollar el mejor nodo (best-node). Consiste en buscar el nodo activo (M) cuya cota (B) es la más alta posible. *Ventajas*: Las mejores soluciones integrales se deben encontrar en nodos con cotas altas, por lo que esta estrategia apunta a encontrar rápidamente buenas soluciones

(o acotar rápidamente B_0) *Desventajas*: Muchos nodos deben permanecer activos por largo tiempo, provocando que se deba usar una gran cantidad de memoria.

Hay otras estrategias más avanzadas que involucran crear un estimador de cuanto debería *degradarse* el valor de la cota B en un nodo dado de acuerdo a su PL y luego elegir aquel con mayor valor estimado para la cota. En la práctica se ocupan estrategias mixtas como por ejemplo hacer DFS hasta que se encuentre un incumbente y luego seleccionar mejor nodo. Gurobi hace esto automáticamente (Cplex permite un poco más de control respecto a esto).

Selección de variable a ramificar (Ver por ejemplo Parámetro VarBranch de Gurobi

Puede que la solución fraccional $\bar{x}$ del nodo que ha sido elegido tenga múltiples variables fraccionales. Una forma estándar es elegir la variable x_j con $j \in E$ más fraccional (la más cercana a 0.5). Otra alternativa llamada *strong branching* involucra resolver pequeñas variaciones del PL original para determinar cual es la variable x_j cuya ramificación produce el mayor decrecimiento en la cota superior B del nodo a ramificar. Esta solución es más cara (involucra resolver un número de PL proporcional a las variables fraccionales) pero en la práctica resulta ser útil.

Criterios de término (Ver lista de parámetros de Gurobi)

BnB puede tomar un tiempo prohibitivo pero podemos detener el proceso en cualquier momento y, si para entonces tenemos un incumbente, podemos retornar una solución factible y una estimación (GAP relativo) de cuan cerca de ser óptima es la solución. Criterios típicos de término temprano incluyen detener la ejecución si: El tiempo de reloj excede un máximo establecido, si el número de nodos procesados excede un máximo, si la memoria necesaria excede un umbral, si el gap relativo o si el gap absoluto ($B - z^*$) es menor que una tolerancia preestablecida.

Heurísticas En muchos casos es posible determinar buenas soluciones factibles (incumbentes) en un nodo cualquiera mediante heurísticas. Por ejemplo, podemos aplicar un algoritmo simple (como glotón o programación dinámica) para encontrar una buena solución factible antes de comenzar BnB. De este modo muchas ramas pueden ser podadas rápidamente por cota al estar dominadas. Otras heurísticas típicas pueden ser usadas en cada nodo. Por ejemplo, redondear (de alguna forma inteligente) una buena solución fraccional puede producir una buena solución factible o bien, usar las soluciones enteras que podrían aparecer mientras se ejecuta SIMPLEX en un nodo. Esto se puede incorporar fácilmente al algoritmo genérico de BnB que ya vimos.

Usar presolver El presolver típicamente es capaz de reducir la complejidad del modelo de manera automática (eliminando variables, restricciones, cotas innecesarias, etc.). El presolver de Gurobi está activado por defecto.

Usar Cortes (branch and cut) En realidad BnB no es un buen método por si solo. Lo ideal es mejorar la formulación en cada nodo. Esto se puede hacer automáticamente mediante el uso de planos cortantes, como lo veremos más adelante.