UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Matemática

Tesis de Licenciatura

El problema del viajante: un algoritmo heurístico y una aplicación.

María Lorena Stockdale

Director: Dra. Susana Puddu
Co-Director: Dra. Gabriela Jerónimo

Noviembre de 2011

Agradecimientos

A mi familia por estar siempre a mi lado y apoyarme en todo lo que emprendo.

A la Dra. Susana y al Dr. Fabio por las maravillosas clases a las cuales he tenido el privilegio de asistir y por enseñarme en cada una de ellas la íntima relación entre los problemas reales y la matemática.

A la Dra. Gabriela por el apoyo invalorable que me brindó en el último tramo de este fascinante camino recorrido. Reflejando en cada una de sus palabras, su calidez y en cada una de sus sugerencias, su sabiduría.

A los Dres. Juan Sabia y Matías Graña por honrarme con su presencia en el jurado.

Al Dr. Gustavo Massaccesi por su valiosa colaboración. Le agradezco el tiempo que me ha dedicado y los aportes sugeridos que enriquecieron este trabajo.

A mis queridos compañeros y amigos, por tantos momentos compartidos, no hubiera sido tan divertido todo este tiempo sin ustedes.

Índice general

RESUMEN

En este trabajo abordamos un problema clásico de optimización combinatoria famoso por ser sencillo de enunciar pero complejo de resolver: el problema del viajante, o más conocido como TSP (*traveling salesman problem*).

La importancia del TSP radica en que diversos problemas del mundo real pueden ser formulados como instancias de éste. Tiene variadas aplicaciones prácticas en problemas que aparentemente no están relacionados: se lo aplica en áreas de logística de transporte, en robótica, en control y operación optimizada de semáforos.

Organizamos esta tesis en cuatro capítulos.

El primer capítulo describe los orígenes y la formulación matemática del problema. A continuación, enunciamos las nociones teóricas necesarias para la demostración de que el TSP es un problema $\mathcal{NP}$-Hard.

En el segundo capítulo comentamos distintas clases de algoritmos con que se cuenta para resolver el TSP.

El tercer capítulo se centra en el algoritmo heurístico de Lin y Kernighan, que es uno de los mejores que se conoce hasta el momento. Luego de discutir los aspectos teóricos del algoritmo, exhibimos el código de una implementación en Visual Basic del mismo realizada como parte de esta tesis. Para concluir, comentamos algunos resultados computacionales que resultan de aplicar la implementación del algoritmo a tres problemas clásicos.

En el último capítulo nos dedicamos al estudio de una aplicación en particular: la secuencia de trabajos. En este problema se quiere secuenciar una cierta cantidad de trabajos en una máquina. Para realizar un trabajo tras otro, se debe realizar una transformación a la máquina, lo que implica un costo. Analizamos un método para encontrar el orden en el que deben realizarse los trabajos de forma que se minimice el costo total.

Capítulo 1

El problema del viajante de comercio

1.1. Orígenes del problema

Hay tres aspectos importantes en la historia de cualquier problema matemático: cómo surge, cómo influye su investigación en el desarrollo de las matemáticas y cómo es finalmente resuelto. Esta tesis está dedicada al estudio del siguiente problema:

Un viajante quiere visitar n ciudades una y sólo una vez cada una, empezando por una cualquiera de ellas y regresando al mismo lugar del que partió.

Supongamos que conoce la distancia entre cualquier par de ciudades.

¿De qué forma debe hacer el recorrido si pretende minimizar la distancia total?.

A este problema se lo conoce con el nombre de *problema del viajante* o TSP[1] y resulta ser uno de los más prominentes dentro del campo de la optimización combinatoria pues todavía no ha sido resuelto eficientemente.

Si queremos conocer su historia debemos remontarnos en principio a la de uno anterior que se origina en la teoría de grafos.

Entendemos por **grafo** al par $G = (V, E)$ donde V es un conjunto finito de elementos que llamamos vértices y E es un conjunto de pares de vértices que denominamos ramas.

Un **ciclo** es una sucesión de vértices $u_1, u_2, ..., u_p$ tales que $u_1, u_2, ..., u_{p-1}$ son distintos, $u_p = u_1$ y $(u_i, u_{i+1}) \in E$. Si el ciclo contiene todos los vértices se llama **hamiltoniano** (en honor al matemático irlandés Sir William Rowan Hamilton).

El TSP para un grafo que tiene asignado para cada una de sus ramas un cierto peso es el problema de encontrar un ciclo hamiltoniano de mínimo peso, entendiéndose por peso del ciclo a la suma de los pesos de todas las ramas que pertenecen a él.

[1]TSP: traveling salesman problem

Inversamente, el problema de decidir si un grafo tiene un ciclo hamiltoniano es un caso especial del TSP (si se le asigna a todas las ramas del grafo peso 0 y a las ramas faltantes las agregamos asignándoles peso 1, se tendrá otro grafo que tiene ciclos hamiltonianos. Resolviendo el TSP, el nuevo grafo tiene un ciclo hamiltoniano de mínimo peso = 0 si y sólo si el grafo original contiene un ciclo hamiltoniano).

Previo al análisis de Hamilton, Euler y Vandermonde discutieron el problema del tour del caballo que se trata de encontrar un ciclo hamiltoniano para el grafo cuyos vértices representan los 64 cuadrados del tablero de ajedrez, con dos vértices adyacentes si y sólo si un caballo puede moverse en un paso de un cuadrado a otro.

El reverendo T. P. Kirkman fue el primero en considerar ciclos hamiltonianos en un contexto general. Dio una condición suficiente para que un grafo poliédrico admita un tal ciclo y además mostró que un poliedro con un número impar de vértices en donde cada cara tiene un número par de aristas, no tiene dichos ciclos.

Al mismo tiempo en que Kirkman realizaba sus investigaciones, Hamilton inventaba un sistema de álgebra no conmutativa. Hamilton llamó a esta álgebra, cálculo icosaédrico (vértices adyacentes del dodecaedro se corresponden con caras adyacentes del icosaedro). Usó la interpretación gráfica como la base para un juego llamado el juego del icosaedro, éste consistía de varios problemas, como por ejemplo terminar de encontrar un ciclo teniendo prefijadas las 5 primeras posiciones.

Un precursor más directo del TSP, en donde el largo de las ramas jugaba un rol predominante fue, una nueva definición de la medida de una curva que propuso Menger. Él la definió como el supremo del conjunto formado por todos los números que pueden ser obtenidos de tomar cada conjunto finito de puntos sobre la curva y determinar la medida de la menor poligonal que los une. A este problema se lo llamó el del mensajero. La resolución no requiere un ciclo, sólo un camino que contenga todos los vértices.

En 1832 fue impreso en Alemania un libro titulado: **El problema del viajero, cómo debe hacer para obtener éxito en sus negocios**. En su último capítulo se vislumbra la esencia del TSP cuando se comenta que con una elección apropiada del tour, se puede ganar mucho tiempo y que el aspecto más importante es cubrir tantas ciudades como sean posibles sin visitar una de ellas dos veces.

Merrill Flood fue el responsable de divulgar el nombre del TSP. Le habló acerca de él a A. W. Tucker en 1937. Tucker le comentó que recordaba haberlo escuchado de boca de Hassler Whitney de la Universidad de Princeton pero no podía confirmar con certeza esta historia. De ser verídica asegura que ocurrió en los años 1931-1932 pues fue en ese entonces cuando se hallaba terminando su tesis con Lefschetz. Whitney era un compañero que se encontraba en su etapa posdoctoral y estaba trabajando en teoría de grafos, especialmente en planaridad y en el problema de los cuatro colores y Flood era un estudiante recién graduado. El problema del viajante ya tenía un nombre.

John Williams incitó a Flood en 1948 a popularizar el TSP en la corporación RAND, motivado por el propósito de crear talentos intelectuales para modelos fuera de la teoría de juegos. No hay dudas de que la reputación y la autoridad de RAND, que rápidamente se convirtió en el centro intelectual de muchas de las investigaciones sobre esta teoría, amplificó la publicidad de Flood. Otra razón de la popularidad del problema fue su íntima conexión con el problema de la asignación y el del transporte.

La aparición del artículo "Soluciones de un problema del viajante de gran tamaño"de Dantzig, Fulkerson y Johnson en el Journal of the Operations Research Society of America fue uno de los principales eventos en la historia de la optimización. Para entender su importancia necesitamos conocer el estado en que se encontraba la optimización combinatoria en el momento en que el paper apareció y dos problemas en particular de la programación lineal: el problema del transporte y el de asignación.

El problema de la asignación es elegir n elementos, uno por cada fila y columna de una matriz $C = (c_{ij})$ de $n \times n$ tales que la suma de los elementos elegidos sea la menor posible. Hay $n!$ maneras posibles de hacer la elección, por lo tanto, un algoritmo efectivo debería hacer algo diferente que considerar todas las posibilidades. Una alternativa dentro de la programación lineal es considerar el poliedro P en el espacio n^2 definido como el conjunto de todas las matrices $X = (x_{ij})$ que satisfacen las condiciones $x_{ij} \geq 0$, $\sum_j x_{ij} = 1 \ \forall i$, $\sum_i x_{ij} = 1 \ \forall j$ y minimizar $\sum c_{ij}x_{ij}$.

De acuerdo a Birkhoff, los vértices de P son precisamente todas las matrices X en donde cada fila y columna contiene exactamente un 1 y todas las otras entradas son 0. Por lo tanto, los $n!$ vértices de P se corresponden con las $n!$ elecciones posibles y la $\sum c_{ij}x_{ij}$ calcula el valor de cada elección. Como el óptimo de esta función se alcanza en un vértice, se pueden utilizar los algoritmos de la programación lineal.

El problema del tranporte es más general que el problema de la asignación. Es el problema de elegir una matriz $X = (x_{ij})$ de $m \times n$ que satisfaga $x_{ij} \geq 0$, $\sum_j x_{ij} = a_i$ $\forall i$, $\sum_i x_{ij} = b_j \ \forall j$ que minimiza $\sum c_{ij}x_{ij}$ donde a_i y b_j son enteros no negativos que cumplen $\sum_i a_i = \sum_j b_j$. El problema del transporte modela la siguiente situación: se tienen m recursos i $(i = 1, ..., m)$, de cada uno de los cuales hay una cierta cantidad a_i, que se quieren enviar a n destinos j $(j = 1, ..., n)$ sabiendo que cada uno ha pedido una cierta cantidad de mercadería b_j. Si cuesta exactamente c_{ij} enviar una unidad de mercadería i al destino j, ¿cómo podría organizarse la entrega para cumplir con todos los requerimientos y minimizar el costo total?.

Dantzig desarrolló el método simplex para resolver problemas de programación lineal. En 1953, existían códigos de implementación efectivos del método simplex en general y adaptaciones especiales para los casos del problema del transporte y de la asignación.

En 1954 Dantzig, Fulkerson y Johnson hicieron un método que resolvía el problema del horario de los buques. La solución llegó varios años después de que el modelo se volviera obsoleto, pero logró sobrevivir porque el método podía ser usado para estudiar preguntas básicas de la teoría combinatoria. Ford y Fulkerson escribieron su primer reporte sobre sistemas de flujo en 1956, iniciando de este modo un tópico mayor del cual derivó un resultado de Johnson sobre la secuencia de trabajos. El TSP, sin embargo, no estaba relacionado a simple vista con estos desarrollos pero había esperanza de que lo estuviera.

Volviendo al problema de asignación descripto arriba, si c_{ij} es la distancia de la ciudad i a la ciudad j, entonces el TSP guarda cierta similitud con el problema de la asignación. Podemos interpretar que $x_{ij} = 1$ significa que el viajero se mueve de la ciudad i a la j en su tour. Una solución al problema de la asignación, bajo esta interpretación, puede ser un conjunto de subtours disconexos en donde cada ciudad es

visitada exactamente una vez, con lo cual no resolvería el TSP. Deberíamos imponer la condición adicional de no permitir subtours, que se puede expresar matemáticamente mediante 2^{n-1} inecuaciones. Desafortunadamente, el conjunto de vértices del nuevo poliedro Q, a diferencia del conjunto de vértices del poliedro original P, contiene matrices con entradas distintas de 0 y 1, por lo tanto, también debemos imponer la condición que las entradas en X deben ser 0 o 1. Estos cambios producen dificultades. La primera es que en vez de $2n$ ecuaciones de variables no negativas, tenemos un enorme número de inecuaciones extras a considerar. Una dificultad aún mayor es el requerimiento que las variables sean 0 o 1, que es una condición de la programación lineal entera, un tema no estudiado aún en 1950.

Consideremos el conjunto de todas las matrices X que satisfacen los requerimientos del TSP (matrices que en sus entradas sólo tienen ceros y unos y que describen tours) y supongamos que conocemos, además de las inecuaciones que excluyen los subtours, todas las otras inecuaciones necesarias para crear un nuevo poliedro R cuyos vértices son precisamente los tours, así, en principio, se puede aplicar programación lineal. Dantzig, Fulkerson y Johnson especulaban que, empezando de un tour óptimo o tal vez cercano al óptimo, era posible probar optimalidad utilizando pocas ecuaciones adicionales (llamadas cortes). El método parecía funcionar en pequeños problemas, por eso, pasaron a trabajabar sobre uno de 49 ciudades. Dantzig, Fulkerson y Johnson sugirieron la posibilidad de que fuesen necesarios un gran número de cortes. Dantzig, optimista, le apostó a Fulkerson que el número de cortes necesarios eran a lo sumo 25, en cambio Fulkerson más pesimista opinaba que se necesitaban al menos 26. Las predicciones fueron bastante acertadas: la cantidad correcta resultó ser 26 pero en el paper que se publicó se decía que sólo 25 eran necesarios.

Dantzig, Fulkerson y Johnson no sólo resolvieron un TSP de tamaño considerable sino que también demostraron que la complejidad de la estructura de un problema de optimización combinatoria no era un obstáculo insuperable para resolverlo. Ellos utilizaron por primera vez el concepto de branch y bound, que es un método computacional muy popular, en particular, cuando se requiere que sólo algunas de las variables sean enteras. A grandes rasgos se trata de tomar una variable x que debe ser entera pero cuyo valor no lo es y a través del branching se consideran dos casos: x es al menos el mayor valor entero más cercano o x es a lo sumo el menor valor entero más cercano. Esta construcción genera un árbol de búsqueda con nodos correspondientes a programas lineales con varias restricciones en donde no es necesario crecer pasado los nodos donde el bound sobre el valor de la solución indica que el árbol no necesita ser explorado después de esos nodos. Todas estas ideas fueron ingredientes indispensables en la solución de la mayoría de los problemas de optimización combinatoria que surgen en la programación lineal entera.

Es claro que la publicación del paper en cuestión fue un verdadero logro, aun cuando los autores se rehusaron a afirmar el desarrollo de un algoritmo general. Es de remarcar, haciendo una lectura en retrospectiva, cuánto de la filosofía de los últimos avances de la optimización combinatoria fueron imaginados en el exitoso abordaje de una instancia del TSP.

Otro método que se ha aplicado en la resolución del TSP fue la programación dinámi-

ca pero debido a la enorme cantidad de condiciones que incluye puede resolver instancias de problemas relativamente pequeños.

1.2. Formulación

Matemáticamente podemos pensar al TSP como el problema de hallar un ciclo hamiltoniano de mínima distancia en un grafo completo $G = (V, E)$ en donde $V = \{1, 2, ..., n\}$ es el conjunto de nodos y representan las ciudades, E es el conjunto de ramas que denotan la conexión entre ellas y

$$d : E \to \mathbb{R}_+$$

una función que a cada $(i, j) \in E$ le asigna la distancia d_{ij} entre las ciudades i y j.

Algunos casos particulares de este problema son:

- Problema del viajante simétrico o STSP [2], si $d_{ij} = d_{ji}\ \forall (i, j) \in E$.
- Problema del viajante asimétrico o ATSP [3], si $d_{ij} \neq d_{ji}$ para por lo menos una $(i, j) \in E$.
- Problema del viajante triangular o $\triangle TSP$ [4], si $d_{ik} \leq d_{ij} + d_{jk}\ \forall i, j, k$, pues satisface la desigualdad que lleva el mismo nombre.

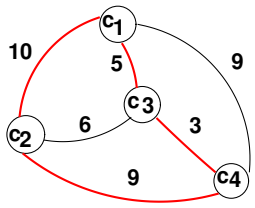
Nuestro primer objetivo es tratar de comprender que resolver este problema no es para nada sencillo, ya que estamos frente a un problema $\mathcal{NP}$-Completo, pero para esto es necesario contar con cierta base teórica que pasaremos a relatar.

1.3. Teoría de problemas

Empezaremos dando la noción de lo que es un problema. Un **problema** será una pregunta general a responder. Usualmente posee varios parámetros o variables libres, cuyos valores no son especificados. Podemos describirlo dando una descripción general de todos sus parámetros y declarando las propiedades que la respuesta o solución debe satisfacer. Una **instancia** se obtiene especificando valores particulares para todos los parámetros del problema. Si por ejemplo tomamos al TSP clásico, los parámetros serán

[2] STSP: symmetric traveling salesman problem
[3] ATSP: asymmetric traveling salesman problem
[4] $\triangle TSP$: triangular traveling salesman problem

el conjunto finito $C = \{c_1, c_2, ..., c_m\}$ de ciudades y para cada par de ciudades c_i, c_j en C, la distancia $d(c_i, c_j)$ entre ellas. Una solución es un orden $< c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(m)} >$ de las ciudades dadas que minimiza $d(c_{\pi(1)}, c_{\pi(2)}) + d(c_{\pi(2)}, c_{\pi(3)}) + ... + d(c_{\pi(m-1)}, c_{\pi(m)}) + d(c_{\pi(m)}, c_{\pi(1)})$. Esta expresión da la longitud total del tour que empieza en $c_{\pi(1)}$, visita cada ciudad en la secuencia, y luego vuelve directamente a $c_{\pi(1)}$ de la última ciudad $c_{\pi(m)}$. Si tomamos $C = \{c_1, c_2, c_3, c_4\}$, $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$ y $d(c_3, c_4) = 3$ tenemos una instancia del TSP. El orden $< c_1, c_2, c_4, c_3 >$ es una solución para esta instancia, y el tour correspondiente tiene distancia mínima de 27.



Una instancia del TSP. La distancia del mínimo tour, en este caso, es 27.

Los problemas pueden ser de dos clases:

- problemas de optimización o
- problemas de decisión.

Un problema de optimización puede describirse como sigue: dado un conjunto finito E, una función $f : E \to \mathbb{R}$ y una familia F de subconjuntos factibles de E, se quiere encontrar un $x \in F$ tal que $f(x)$ sea máximo o mínimo sobre todos los miembros de F.

Los problemas de decisión son aquellos cuya respuesta es sí o no. Cada problema de optimización tiene un problema de decisión correspondiente. Ejemplos:

1. Se define cubrimiento por vértices de un grafo no dirigido $G = (V, E)$ a un subconjunto S de V tal que para cada $(a, b) \in E$, ya sea $a \in S$ o $b \in S$. Dado un grafo G, encontrar el menor número k tal que G tiene un cubrimiento por vértices de tamaño k es el problema de optimización. Puede definirse como un problema de decisión si además del grafo se tiene un número entero positivo K y lo que se busca es saber si existe un cubrimiento por vértices S para G de tamaño menor o igual que K.

2. Supongamos que tenemos n objetos, cada objeto i tiene un peso determinado $0 < p_i \leq P$ y una utilidad o valor asociado v_i también positivo. El problema de optimización de la mochila consiste en llenar una mochila de capacidad P con dichos objetos, maximizando las utilidades sin exceder la capacidad. Podemos transformarlo en un problema de decisión si agregamos un k y nos preguntamos si existe un subconjunto de los objetos que cabe en la mochila y tiene beneficio total por lo menos k.

3. El problema de partición visto como un problema de decisión consiste en decidir si dado un conjunto S de números enteros, puede ser particionado en dos subconjuntos S_1 y S_2 tal que la suma de los elementos en S_1 sea igual a la suma de los elementos en S_2. Los subconjuntos S_1 y S_2 deben formar una partición en el sentido de que son disjuntos y cubren S. La versión de optimización pide la "mejor"partición y se puede plantear como buscar una partición en dos subconjuntos S_1 y S_2 tal que sea mínimo el máx$(sum(S_1), sum(S_2))$ (a veces se agrega la restricción de que los tamaños de los dos subconjuntos deben ser iguales).

4. Dado un grafo completo G, c un costo entero definido sobre cada una de sus ramas y una constante B, hallar un ciclo hamiltoniano de mínimo costo es el problema de optimización TSP, en cambio, responder sólo si el costo mínimo es $\leq B$ es un problema de decisión llamado TSPD.

Cabe destacar que si pudiésemos encontrar un tour de mínima distancia para el TSP entonces podríamos resolver el problema de decisión asociado pues lo que restaría es calcular su peso y compararlo con B. Recíprocamente, veremos más adelante que si podemos resolver el de decisión también podríamos obtener la solución del de optimización. Y es por esto que por el momento nos ocuparemos de los llamados problemas de decisión, convencidos de que podemos extender sus implicancias a los problemas de optimización.

Estamos interesados en la resolución de problemas, es aquí cuando nos toca introducir el concepto de algoritmo. Los **algoritmos** son procedimientos (programas de computadora) para resolver problemas. Se dice que un algoritmo resuelve un problema π si ese algoritmo puede ser aplicado a cualquier instancia I de π y produce siempre una solución para esa instancia I. Llamamos **input** a la descripción de una instancia que le damos a la computadora, lo que podemos hacer a través de una **palabra** que es una secuencia finita de símbolos elegidos de un alfabeto finito. Hay diferentes formas de describir una misma instancia de un problema. El **tamaño del input** para una instancia I de un problema se define como el número de símbolos en la descripción de I. Por ejemplo podríamos describir la instancia del TSP planteada al inicio de esta sección usando el alfabeto $\{c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ a través de la siguiente palabra $c[1]$ $c[2]$ $c[3]$ $c[4]$ // 10/5/9 // 6/9 //3. El tamaño del input en este caso sería 32. Al tamaño del input se lo usa para definir formalmente el tamaño de la instancia.

Es muy importante poder predecir el tiempo que le va a llevar al algoritmo resolver una cierta instancia, que puede ser expresado en términos de una sola variable: el tamaño de la instancia. Por ejemplo, lo que contribuye a la cantidad de información del input en una instancia de m ciudades del TSP será el número de ciudades y los $m(m-1)/2$ números que definen las distancias. La **complejidad** de un algoritmo M para un input x de tamaño n es $C_{M(n)} = \text{máx}_{|x|=n}$ { número de pasos que realiza M para procesar x }.

Ahora que ya tenemos todos estos conceptos podemos definir matemáticamente lo que es un problema y un algoritmo.

Un problema Π es un subconjunto de palabras formadas con un alfabeto. Los símbolos que utilizaremos para referirnos al alfabeto y al conjunto de todas las palabras que se

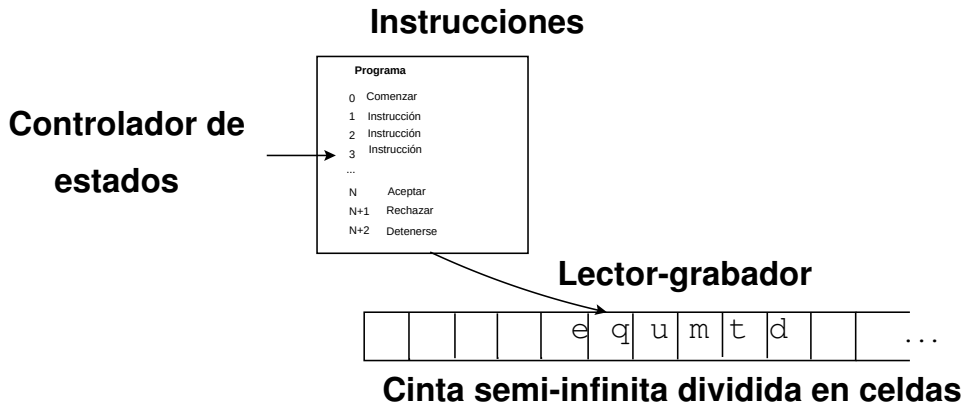
pueden formar con sus símbolos serán: $\sum$ y $\sum_*$ respectivamente. Cualquier subconjunto $\Pi \subseteq \sum_*$ es un problema por definición.

Por algoritmo utilizaremos un modelo de cálculo inventado por Alan Turing llamado la máquina de Turing.

Máquina de Turing

La máquina de Turing consta de tres partes:

- Una lista finita de instrucciones (programa) y un controlador de estados (que nos indica qué instrucción se está ejecutando)
- Una cinta semi-infinita dividida en celdas
- Un cursor o lector/ grabador de las celdas



Los caracteres grabados en la cinta son símbolos de un alfabeto $\sum = \{\alpha, \beta, ..., \omega\}$. Hay un caracter adicional b, blanco, que representa la ausencia de símbolo.

Las instrucciones o estados están dados por una función $\phi(n, \alpha)$ del número de estado n y caracter α siendo leído por el cursor, y el valor de la función es la terna (m, β, δ) donde m es el número de la instrucción siguiente, β es el caracter grabado en la cinta sobre α y $\delta \in \{\leftarrow, 0, \rightarrow\}$ indica si el cursor retrocede a la celda anterior, no se mueve o avanza a la próxima celda.

La ejecución de una instrucción constituye un paso del programa.

Inicialmente, los caracteres que figuran en la cinta desde la primera celda no vacía hasta la última no vacía es el input. El programa se inicia en el estado inicial 0 leyendo el input y procediendo de acuerdo con el programa; este proceso se detiene si alcanza uno de los estados Aceptar, Rechazar o Detenerse o puede seguir funcionando indefinidamente.

El número de pasos hasta que el programa se detiene (si se detiene) mide el tiempo de ejecución.

Si el programa alcanza Aceptar decimos que la máquina ha aceptado la palabra del input o que la respuesta al problema de decisión es "sí". El conjunto de palabras que acepta se llama el lenguaje que acepta la máquina.

Si la máquina se detiene, a la palabra que figura en la cinta en ese momento la llamamos el output.

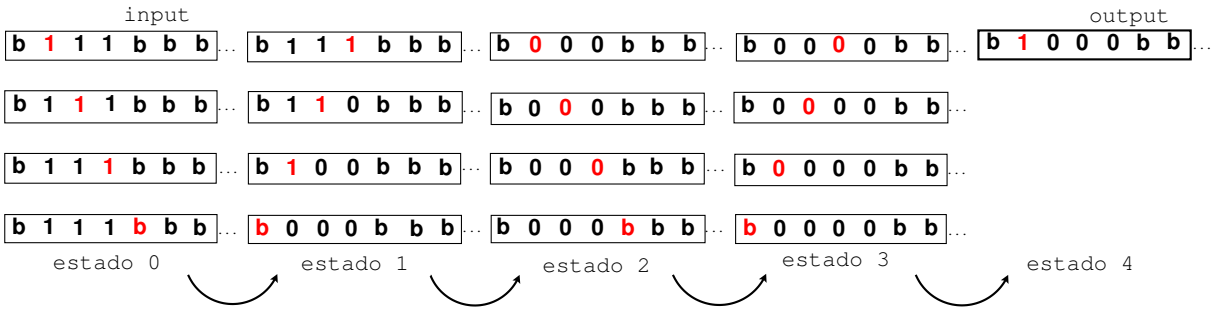
Para entender mejor cómo trabaja esta máquina plantearemos un ejemplo.

Supongamos que dado un número natural x escrito en binario queremos un algoritmo que nos devuelva $x+1$. Por ejemplo nos gustaría que al ingresar 7= 111 (en base 2) nos arroje como resultado 8= 1000 (en base 2) o que al computar 74= 1001010 (en base 2) nos retorne el 75 = 1001011 (en base 2).

La idea sería pararnos en el estado 0, por eso inicialmente leemos el input hasta llegar al primer blanco y luego retrocedemos una celda para ubicarnos en el último caracter del input. Si en el estado 0 hay un 0 en la cinta, simplemente lo cambiamos por un 1 y obtendríamos el resultado deseado. Si en cambio, hay un 1, al sumar otro 1 se obtiene 0 pues estamos en base 2, luego tendríamos que ir recorriendo el número a izquierda y cambiar todos los unos que encontremos por ceros; si en algún momento nos topamos con un 0 lo cambiamos por un 1 y nos detenemos y, si esto no sucediera vamos a tener que cambiar el blanco del final por un cero y volver a la primera celda cuyo valor es 0 y cambiarlo por un 1 para que el output quede escrito a partir de la primer celda de la cinta.

n	α		m	β	δ
0	0		0	0	$\rightarrow$
0	1		0	1	$\rightarrow$
0	b		1	b	$\leftarrow$
1	1		1	0	$\leftarrow$
1	0		D	1	0
1	b		2	b	$\rightarrow$
2	0		2	0	$\rightarrow$
2	b		3	0	$\leftarrow$
3	0		3	0	$\leftarrow$
3	b		4	b	$\rightarrow$
4	0		D	1	0

Podemos pensar a esta tabla como el programa o la lista de instrucciones. Los estados son $\{0, 1, 2, 3, 4, D\}$ donde D es detenerse.



Ahora estamos en condiciones de definir lo que es un algoritmo.

Definición: Un algoritmo es una máquina de Turing que para con cualquier input (alcanza el estado detenerse).

1.3.1. Clase P

Definición: Un algoritmo es de tiempo polinomial si existe un polinomio P tal que $C_M(n) \leq P(n)\ \forall n$ (recordemos que $C_M(n)$ hace referencia a la complejidad de un algoritmo M para un input de tamaño n y es el número de pasos que como máximo realiza M con cualquier entrada de tamaño n).

Diremos que una función $f(n)$ es del orden de $g(n)$, y escribiremos $O(g(n)))$, si existe una constante c tal que $\mid f(n) \mid \leq c \mid g(n) \mid \forall n \geq 0$. Según la definición que dimos anteriormente se puede decir que un algoritmo de tiempo polinomial es aquél cuya función de complejidad es del $O(P(n))$ para algún polinomio P.

Hay tres categorías de problemas en lo que se refiere a la complejidad:

- **problemas polinomiales**: si se conoce un algoritmo polinomial que lo resuelva.

  Por ejemplo, supongamos que tenemos n números y queremos ordenarlos de menor a mayor. Hay distintos algoritmos para hacerlo, uno de ellos es el **MergeSort** que es del $O(n \log n)$. Este algoritmo usa la estrategia "dividir y conquistar", ya que divide al conjunto $x_1, x_2, ..., x_n$ en dos partes iguales si n es par, o en dos partes que difieren en uno si n es impar. Llamemos A y B a estas dos partes. Observemos que si A y B estuvieran ordenadas entonces necesitamos a lo sumo n-1 comparaciones para ordenar A $\cup$ B. Definamos $f(n)$ como el número de comparaciones para ordenar un conjunto de n números partiendo el conjunto en dos partes, ordenando cada parte y luego ordenando la mezcla de ambas. Podemos observar que si n es par vale la recurrencia $f(n) = 2f(n/2) + n - 1$; además tenemos como condición de borde $f(2) = 1$ y $f(1) = 0$. Veamos que $f(n) = O(n \log n)$.

Si $n = 2^k$, aplicando la relación sucesivamente obtenemos:

$f(n) = 2f(n/2) + n - 1$

$2f(n/2) = 2^2 f(n/2^2) + n - 2$

$2^2 f(n/2^2) = 2^3 f(n/2^3) + n - 2^2$

...

$2^{k-1} f(n/2^{k-1}) = 2^k f(n/2^k) + n - 2^{k-1}$

por lo tanto $f(n) = 2^k f(n/2^k) + nk - (2^k - 1)$. Como $k = \log_2 n$ podemos escribir $f(n) = n \log_2 n - (n-1)$ para $n = 1, 2, 2^2, 2^3, ...$ de lo que se deduce que si n es una potencia de 2 vale que $f(n) = O(n \log n)$.

Consideremos ahora un n cualquiera. Existe k tal que $2^{k-1} \leq n < 2^k$. La función $f(n)$ es no decreciente así que $f(n) \leq f(2^k) \leq M 2^k \log_2 2^k = 2M 2^{k-1} \log_2 2^k \leq 2Mn \log_2 2^k = 2Mnk = 2Mn(k-1+1) \leq 2Mn(\log_2 n + 1) = O(n \log n)$.

Un grafo se dice *conexo* si dados u, $v \in V$ hay un camino (sucesión de vértices $u_1, ..., u_p$ tales que $(u_i, u_{i+1}) \in E$) que une u con v. El problema de ver si un grafo es conexo es polinomial. Un algoritmo que lo resuelve es el **Search** que es del $O(n^2)$ donde n es el número de vértices.

- **problemas intratables**: si no se pueden resolver en tiempo polinomial. Hay dos tipos,

  $*$ aquellos que requieren una salida de tamaño no polinomial.

  Por ejemplo, si queremos hallar los ciclos hamiltonianos en un grafo completo de n vértices podemos partir de uno cualquiera de ellos, luego tendremos $n-1$ opciones para elegir el segundo vértice y siguiendo con este razonamiento se llega a que los ciclos hamiltonianos son en total $(n-1)!$

  $*$ aquellos que no requieren salidas no polinomiales pero podemos probar que no se pueden resolver en tiempo polinomial.

- **problemas que no se han demostrado intratables, pero para los cuales no se ha encontrado un algoritmo polinomial**.

  Por ejemplo, el problema de Knapsack, en donde suponemos que tenemos una mochila que puede soportar como peso máximo P(entero no negativo) y además contamos con n items cuyos pesos son $p_1, p_2, ..., p_n$ (enteros no negativos) y sus valores son $v_1, v_2, ..., v_n$, y nos preguntamos si existe un subconjunto de items cuyo peso no supere a P de forma que su valor sea máximo.

  Decimos que un grafo es *k-colorable* si podemos pintar cada uno de sus vértices con uno de los $k \geq 2$ colores de forma tal que los extremos de cada rama tengan distintos colores. Ver si un cierto grafo dado G es k-colorable es otro problema de este tipo al que se lo conoce con el nombre de colorabilidad.

  El problema del viajante también se encuentra dentro de esta clase de problemas.

El objetivo es tratar de resolver problemas pero si éstos se encontraran dentro de las últimas dos clases expuestas con anterioridad tendríamos serias dificultades, ya que no tienen algoritmos polinomiales, por lo que un algoritmo que los resuelva en forma exacta puede tardar un tiempo prohibitivo. Así que vamos a tener que usar algoritmos polinomiales que arrojen soluciones aproximadas. Hay dos categorías de tales algoritmos: heurísticos y de aproximación. Nos dedicaremos a toda esta parte de la teoría en el capítulo 2.

Definición: La clase $\mathcal{P}$ es el conjunto de todos los problemas de decisión que pueden ser resueltos por un algoritmo polinomial.

Es decir, que para cada problema $\Pi \in \mathcal{P}$, existe un algoritmo y un polinomio p de forma que una instancia de Π cuyo input es de tamaño w puede ser resuelto por ese algoritmo en a lo sumo $p(w)$ pasos.

1.3.2. Clase NP

Empezaremos dando una idea intuitiva de esta nueva clase de problemas. Consideremos el TSPD descripto anteriormente. Como comentamos, no se conoce un algoritmo de tiempo polinomial que lo resuelva. Sin embargo, supongamos que una persona viniera diciendo que para una instancia particular del problema, la respuesta a esa instancia es sí y nos da un ciclo asegurando que es hamiltoniano de costo $\leq B$. Sería fácil para nosotros verificar la veracidad o falsedad de sus dichos, deberíamos ver inicialmente que es un tour, calcular su costo y compararlo con B. Podríamos hacer el proceso de verificación como un algoritmo de complejidad polinomial en el tamaño de la instancia.

Lo mismo ocurriría si para una instancia del problema de Knapsack con respuesta positiva nos entregaran como información una lista con los items por los que debiéramos optar para que su peso total no supere el que puede soportar la mochila (P). Sólo tendríamos que calcular la suma de sus pesos y constatar que efectivamente lo es. Caso contrario, para averiguar si existe un subconjunto de items con dicha propiedad, nos correspondería, en el peor de los casos hacer $\binom{n}{1} + \binom{n}{2} + ... + \binom{n}{n}$ verificaciones.

Como último ejemplo, y en relación al problema de colorabilidad, imaginemos que nos dan una instancia positiva, adjuntándonos una asignación de colores para cada uno de los nodos del grafo. Podríamos comprobar la afirmación precedente tomando cada una de las ramas del grafo y viendo que sus vértices tienen distinto color.

A esa información extra, que codificada es representada mediante una palabra con los símbolos del alfabeto, se la llama **certificado**.

Observamos que dada una instancia positiva del problema (es decir, una instancia cuya respuesta es sí) si nos entregan un certificado (en nuestros ejemplos: un ciclo hamiltoniano de costo $\leq B$, una lista con los ítem cuyo peso total es P, una coloración para los vértices) entonces se puede demostrar en tiempo polinomial el hecho de que efectivamente es una instancia positiva.

Definición: Decimos que π pertenece a la clase $\mathcal{NP}$ si para cualquier instancia positiva con input w del problema π existe una palabra $v(w)$ cuyo tamaño es a lo sumo $p(\mid w \mid)$ y un algoritmo que con input $(v(w), w)$ demuestra en tiempo $\leq p(\mid w \mid)$ que $w \in \pi$.

Nos referimos a $v(w)$ como el certificado, es decir, se trata de la información que agregamos a la descripción del problema π que nos permite verificar que w es una instancia positiva. Más aún, decimos que el certificado es sucinto porque el tamaño de $v(w)$ es $\leq p(\mid w \mid)$.

Observación: $\mathcal{P} \subset \mathcal{NP}$ pues suponiendo que $\pi \in \mathcal{P}$, por definición hay un algoritmo que dada cualquier instancia $w \in \pi$, si es positiva lo demuestra en tiempo polinomial. En la definición de $\mathcal{NP}$ podemos tomar $v(w)=$ vacío.

1.3.3. Clase NP - Completos

Reducibilidad polinomial

Supongamos que queremos resolver el problema de decisión A y que tenemos un algoritmo que resuelve el problema de decisión B.

Supongamos también que tenemos un algoritmo que construye una instancia $\mathbf{y}$ de B para cada instancia $\mathbf{x}$ de A de tal forma que un algoritmo para B responde sí para $\mathbf{y} \iff$ la respuesta al problema A para $\mathbf{x}$ es sí. Dicho algoritmo se denomina algoritmo de transformación. El algoritmo de transformación combinado con el algoritmo para B nos da un algoritmo para A. Cuando el algoritmo de transformación es polinomial decimos que se trata de una transformación polinómica.

Definición: si existe un algoritmo de transformación polinomial del problema de decisión A en el problema de decisión B, el problema A es *reducible polinomialmente* al problema B. Se nota $A \propto B$. En caso que exista un tal algoritmo decimos que B es un problema más duro que A.

Enunciaremos y demostraremos algunos ejemplo de reducibilidad polinomial.

Teorema: Ciclo Hamiltoniano $\propto$ TSPD

Demostración:

Dado un grafo $G = (V, E)$, queremos saber si tiene un ciclo hamiltoniano. Construimos $G^* = (V, E \cup E^*)$ el grafo completo correspondiente a G. Definimos un costo en cada rama de G^* que es 0 si la rama pertenece a E y vale 1 si pertenece a E^*. Entonces G tiene un ciclo hamiltoniano si y sólo si el costo óptimo es ≤ 0 para el TSPD de G^*.$\square$

Teorema: TSP $\propto$ TSPD

Demostración:

Supongamos que queremos resolver el TSP con un algoritmo que resuelve el TSPD. Sea $c_{\text{máx}}$ el costo máximo de las ramas de G y sea c^* el costo mínimo de un ciclo hamiltoniano. Entonces $0 \leq c^* \leq nc_{\text{máx}}$ donde n es el número de ramas de cualquier ciclo hamiltoniano. Usando el algoritmo para TSPD con $B = nc_{\text{máx}}/2$ podemos determinar a qué semi-intervalo $(0, nc_{\text{máx}}/2)$ o $(nc_{\text{máx}}/2, nc_{\text{máx}})$ de $(0, nc_{\text{máx}})$ pertenece c^*. Usando la búsqueda binaria determinamos cuanto vale c^* el valor del óptimo ciclo de G. Cambiemos el costo c_u de una rama u de G y pongámoslo igual a $c_u = nc_{\text{máx}}+1$. Apliquemos el algoritmo para TSPD con $B = c^*$ entonces el algoritmo responderá que el costo mínimo es $\leq c^*$ si y sólo si la rama u no pertenece a un ciclo óptimo. Si u no pertenece a un ciclo óptimo suprimimos u. De esta manera podemos repetir el procedimiento hasta que las ramas que queden sean las de un ciclo óptimo. □

Definición: Un problema B es $\mathcal{NP}$- Completo si
∗ está en $\mathcal{NP}$ y
∗ para cualquier otro problema A de $\mathcal{NP}$, $A \propto B$.

Problema SAT-FNC

∗ Una variable lógica es una variable que puede tomar los valores verdadero o falso. Si x es una variable lógica, $\overline{x}$ es la negación de x.
∗ Un literal es una variable lógica o su negación.
∗ Una cláusula es una secuencia de literales separados por el operador lógico "$\vee$".
∗ Una expresión lógica en forma normal conjuntiva (FNC) es una secuencia de cláusulas separadas por el operador lógico "$\wedge$".

El problema de decisión de satisfactibilidad FNC (SAT-FNC) consiste en determinar, dada una expresión lógica en FNC, si existe una asignación de valores verdaderos y falsos a las variables que haga la expresión verdadera.

Stephen Cook en el teorema que lleva su nombre demuestra que SAT-FNC es $\mathcal{NP}$-Completo [1].

Teorema: Un problema C es $\mathcal{NP}$-Completo si:
∗ está en $\mathcal{NP}$ y
∗ para algún problema $\mathcal{NP}$-Completo B, $B \propto C$

Demostración:

Por ser B $\mathcal{NP}$-Completo, para cualquier problema A en $\mathcal{NP}$, $A \propto B$. La reducibilidad es transitiva entonces, $A \propto C$. Y como C está en $\mathcal{NP}$ se deduce que C es $\mathcal{NP}$-Completo. □

Entonces para verificar que un problema es $\mathcal{NP}$-Completo tenemos que ver que

está en $\mathcal{NP}$ y que algún problema $\mathcal{NP}$-Completo se reduce a él. Demostraremos que los problemas 3-SAT, cubrimiento por vértices y circuito hamiltoniano son $\mathcal{NP}$-Completos [2].

El problema 3-SAT es sólo una restricción del problema de satisfabilidad en donde todas las instancias tienen exactamente tres literales por cláusula.

Teorema: 3-SAT es $\mathcal{NP}$-Completo.

Demostración:

$*$ 3-SAT $\in \mathcal{NP}$ pues dada una asignación para las variables se puede verificar en tiempo polinomial si se satisfacen todas las cláusulas dadas, o sea, si la expresión es verdadera.

$*$ Vamos a transformar SAT en 3-SAT.

Sea $U = \{u_1, u_2, ..., u_n\}$ un conjunto de variables y $C = \{c_1, c_2, ..., c_m\}$ un conjunto de cláusulas de una instancia arbitraria del SAT. Construiremos una colección $C^{'}$ de cláusulas con tres literales sobre un conjunto $U^{'}$ de variables de forma que $C^{'}$ se satisface si y sólo si C se satisface.

En la construcción de $C^{'}$ se reemplaza cada cláusula $c_j \in C$ por una colección equivalente $C_j^{'}$ de cláusulas con tres literales, basadas en las variables originales U y en otras adicionales $U_j^{'}$ cuyo uso estará limitado a las cláusulas $C_j^{'}$, o sea,
$U^{'} = U \cup (\bigcup_{1 \leq j \leq m} U_j^{'})$ y $C^{'} = \bigcup_{1 \leq j \leq m} C_j^{'}$. Sólo necesitamos mostrar cómo se construyen $C_j^{'}$ y $U_j^{'}$ a partir de los c_j.

Supongamos que c_j está dada por $\{z_1, z_2, ..., z_k\}$ donde los z_i son todos literales que derivan de las variables en U. La forma en que se construyen $C_j^{'}$ y $U_j^{'}$ depende del valor de k.

- **Caso 1** $k = 1$

  $U_j^{'} = \{y_j^1, y_j^2\}$

  $C_j^{'} = \{\{z_1, y_j^1, y_j^2\}, \{z_1, y_j^1, \overline{y}_j^2\}, \{z_1, \overline{y}_j^1, y_j^2\}, \{z_1, \overline{y}_j^1, \overline{y}_j^2\}\}$

- **Caso 2** $k = 2$

  $U_j^{'} = \{y_j^1\}$

  $C_j^{'} = \{\{z_1, z_2, y_j^1\}, \{z_1, z_2, \overline{y}_j^1\}\}$

- **Caso 3** $k = 3$

  $U_j^{'} = \phi$

  $C_j^{'} = \{\{c_j\}\}$

- **Caso 4** $k > 3$

  $U_j^{'} = \{y_j^i : 1 \leq i \leq k-3\}$

  $C_j^{'} = \{\{z_1, z_2, y_j^1\}\} \cup \{\{\overline{y}_j^i, z_{i+2}, y_j^{i+1}\} : 1 \leq i \leq k-4\} \cup \{\{\ \overline{y}_j^{k-3}, z_{k-1}, z_k\}\}$

Para demostrar que es una transformación, debemos demostrar que el conjunto C' de cláusulas se satisface si y sólo si C lo hace. Supongamos que $t : U \to \{T, F\}$ es una asignación de valores de verdad que satisface C; veremos que puede ser extendida a una asignación de valores de verdad $t' : U' \to \{T, F\}$ que satisface C'. Como las variables en U'-U están particionadas en conjuntos U'_j y como las variables en cada U'_j ocurren sólo en cláusulas relacionadas con C'_j, debemos probar cómo t puede ser extendida a los conjuntos U'_j uno a la vez, y en cada caso verificar que todas las cláusulas en el correspondiente C'_j son satisfechas. Podemos hacerlo como sigue: si U'_j fue construido según el caso 1 o el caso 2, las cláusulas C'_j se satisfacen por t, por eso podemos extender t arbitrariamente a U'_j tal que $t'(y) = T$ para todo $y \in U'_j$. Si U'_j fue construido bajo el caso 3, entonces U'_j es vacío y la única cláusula en C'_j se satisface por t. El único caso destacado es el 4, que corresponde a una cláusula $\{z_1, z_2, ..., z_k\}$ de C con $k > 3$. Como t es una asignación de valores de verdad que satisface C, tiene que existir un entero menor tal que el literal z_l es verdadero bajo t. Si l es 1 o 2, entonces $t'(y^i_j) = F$ para $1 \leq i \leq k-3$. Si l es $k-1$ o k, entonces $t'(y^i_j) = T$ para $1 \leq i \leq k-3$. De otro modo $t'(y^i_j) = T$ para $1 \leq i \leq l-2$ y $t'(y^i_j) = F$ para $l-1 \leq i \leq k-3$. Es fácil verificar que estas opciones aseguran que todas las cláusulas en C'_j serán satisfechas, y por lo tanto todas las cláusulas en C' serán satisfechas por t'. Inversamente, si t' es una asignación de valores de verdad para C', es fácil verificar que la restricción de t' a las variables en U debe ser una asignación de valores de verdad por C. Entonces C' se satisface si y sólo si C lo hace.

Para ver que esta transformación se puede hacer en tiempo polinomial, es suficiente observar que el número de cláusulas con tres literales en C' es acotado por un polinomio en nm. El tamaño de una instancia del 3-SAT es acotado por una función polinomial en el tamaño de una instancia del SAT. □

Teorema: Cubrimiento por vértices (VC) es $\mathcal{NP}$-Completo.

Demostración:

$*$ VC $\in \mathcal{NP}$ pues dado un subconjunto de vértices se puede chequear en tiempo polinomial si contiene al menos uno de los vértices para cada una de las ramas del grafo y si cumple con el tamaño establecido.

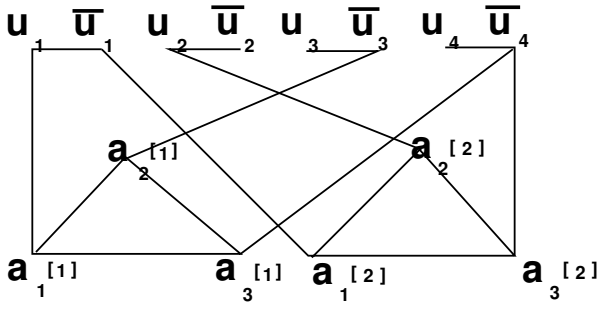
$*$ Vamos a transformar 3-SAT en VC.

Sea $U = \{u_1, u_2, ..., u_n\}$ y $C = \{c_1, c_2, ..., c_m\}$ una instancia del 3-SAT. Vamos a construir un grafo $G = (V, E)$ y un entero positivo $K \leq \mid V \mid$ de forma que G tiene un cubrimiento por vértices de tamaño menor o igual a K si y sólo si C se satisface. Para cada variable $u_i \in U$, se definirá un T_i=(V_i, E_i), con $V_i = \{u_i, \overline{u}_i\}$ y $E_i = \{\{u_i, \overline{u}_i\}\}$, o sea, dos vértices unidos por una rama. Notemos que cualquier VC debe contener a u_i o a $\overline{u}_i$ para cubrir la única rama de E_i.

Para cada cláusula $c_j \in C$, se define $S_j = (V'_j, E'_j)$ que consiste de tres vértices y tres ramas que los unen y forman un triángulo: $V'_j = \{a_1[j], a_2[j], a_3[j]\}$ y $E'_j = \{\{a_1[j], a_2[j]\}, \{a_1[j], a_3[j]\}, \{a_2[j], a_3[j]\}\}$. Notemos que cualquier VC debe contener al menos dos vértices de V'_j para cubrir las ramas de E'_j.

Para cada cláusula $c_j \in$ C, denotemos a los tres literales de c_j como x_j, y_j y z_j, las

ramas que emanan de S_j son: E_j''= {{$a_1[j], x_j$}, {$a_2[j], y_j$}, {$a_3[j], z_j$}}. La construcción de nuestra instancia de VC se completa definiendo a $K = n + 2m$ y $G = (V, E)$ donde V=($\cup_{1\leq i\leq n} V_i$) $\cup$ ($\cup_{1\leq j\leq m} V_j'$) y E=($\cup_{1\leq i\leq n} E_i$) $\cup$ ($\cup_{1\leq j\leq m} E_j'$) $\cup$ ($\cup_{1\leq j\leq m} E_j''$)



La figura representa la instancia del VC que resulta de la instancia del 3-SAT en donde $U = \{u_1, u_2, u_3, u_4\}$, $C = \{\{u_1, \overline{u}_3, \overline{u}_4\}, \{\overline{u}_1, u_2, \overline{u}_4\}\}$ y $K = n + 2m = 8$.

Es fácil ver que la construcción puede ser realizada en tiempo polinomial. Faltaría demostrar que C se satisface si y sólo si G tiene un VC de tamaño menor o igual a K.

Supongamos primero que $V' \subseteq V$ es un VC para G con $\mid V' \mid \leq K$, entonces V' debe contener al menos un vértice de cada T_i y al menos dos vértices de cada S_j. Como esto da un total de al menos $n + 2m = K$ vértices, V' debe contener exactamente un vértice de cada T_i y exactamente dos vértices de cada S_j. Podemos usar la forma en que V' interseca cada componente para obtener una asignación de valores de verdad $t : U \to \{T, F\}$. Definimos $t(u_i) = T$ si $u_i \in V'$ y $t(u_i) = F$ si $\overline{u}_i \in V'$. Para ver que esta asignación satisface cada una de las cláusulas $c_j \in C$, consideremos las tres ramas de E_j''. Sólo dos de estas ramas pueden ser cubiertas por vértices de $V_j' \cap V'$, por lo tanto una de ellas debe ser cubierta por un vértice de algún V_i que pertenecen a V'. Esto implica que el literal correspondiente, u_i o $\overline{u}_i$, de la cláusula c_j es verdadero bajo la asignación t, luego c_j se satisface por t. Como sucede para cada $c_j \in C$, se sigue que t es una asignación de valores de verdad que se satisface para C.

Inversamente, supongamos que $t : U \to \{T, F\}$ es una asignación de valores de verdad que se satisface para C. El correspondiente VC, V', incluye un vértice de cada T_i y dos vértices de cada S_j. El vértice de $T_i \in V'$ es u_i si $t(u_i) = T$ y es $\overline{u}_i$ si $t(u_i) = F$. Esto asegura que al menos una de las tres ramas de cada conjunto E_j'' es cubierta, porque t satisface cada cláusula c_j. Necesitamos incluir en V' los vértices de S_j de las otras dos ramas en E_j'' y terminamos obteniendo el VC deseado. □

Teorema: Circuito hamiltoniano (HC) es $\mathcal{NP}$-Completo.

Demostración:

$*$ HC $\in \mathcal{NP}$ pues dado un orden de los vértices se puede chequear en tiempo polinomial si efectivamente es un ciclo que pasa sólo una vez por cada uno de los vértices del grafo.

$*$ Vamos a transformar Cubrimiento por vértices en Circuito Hamiltoniano.

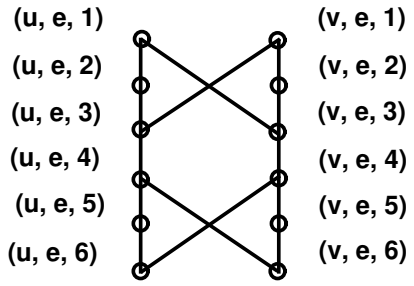
Supongamos que tenemos una instancia arbitraria de VC dada por un grafo $G = (V, E)$ y un entero positivo $K \leq \mid V \mid$. Vamos a construir un grafo G'=(V', E') de forma

que G' tiene un circuito hamiltoniano si y sólo si G tiene un VC de tamaño menor o igual que K.

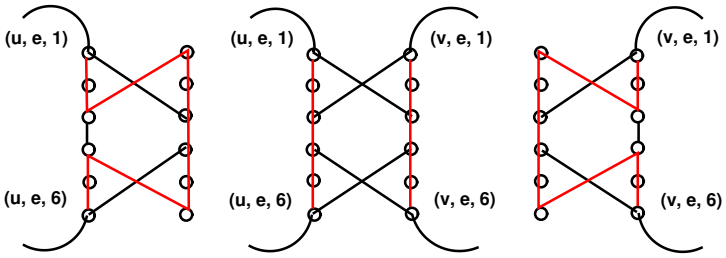
El grafo G' tendrá K vértices "selectores" a_1, a_2, ..., a_K que serán utilizados para seleccionar K vértices del conjunto de vértices V de G.

Por cada rama en E, contiene una componente que asegurará que al menos un extremo de esa rama esté entre los K vértices seleccionados.

La componente para $e = \{u, v\}$ tiene 12 vértices $V'_e = \{(u, e, i), (v, e, i) : 1 \leq i \leq 6\}$ y 14 ramas $E'_e = \{\{(u, e, i), (u, e, i + 1)\}, \{(v, e, i), (v, e, i + 1)\} : 1 \leq i \leq 5\} \cup \{\{(u, e, 3), (v, e, 1)\}, \{(v, e, 3), (u, e, 1)\}\} \cup \{\{(u, e, 6), (v, e, 4)\}, \{(v, e, 6), (u, e, 4)\}\}$

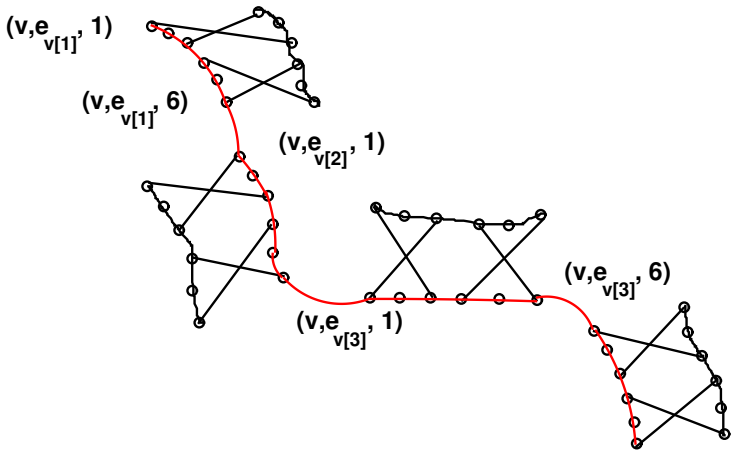


Los únicos vértices que estarán involucrados con ramas adicionales serán $(u, e, 1)$, $(v, e, 1)$, $(u, e, 6)$ y $(v, e, 6)$. Cualquier circuito hamiltoniano de G' tiene que recorrer las ramas de E'_e en exactamente una de las tres configuraciones expuestas a continuación.



Si por ejemplo el circuito "entra"por $(u, e, 1)$, debiera "salir"por $(u, e, 6)$ y visitar todos los vértices o sólo los 6 vértices (u, e, i) $1 \leq i \leq 6$.

Agregaremos algunas ramas adicionales que servirán para unir pares de componentes o para conectar una componente con un vértice selector. Para cada vértice $v \in V$, ordenemos arbitrariamente las ramas que inciden en v como $e_{v[1]}$, $e_{v[2]}$,..., $e_{v[deg(v)]}$, donde $deg(v)$ denota el grado de v en G, esto es, el número de ramas que inciden en v. Todas las componentes que tienen a v como un extremo son unidas a través de las siguientes ramas $E'_v = \{\{(v, e_{v[i]}, 6), (v, e_{v[i+1]}, 1)\} : 1 \leq i < deg(v)\}$. Como se muestra en el siguiente gráfico se crea un camino simple en G' que incluye exactamente los vértices (x, y, z) con $x = v$.



Las últimas ramas que definimos en $G^{'}$ conectan el primer y el último vértice de cada uno de estos caminos con cada uno de los vértices a_1, a_2,..., a_K, las especificamos de la siguiente manera: $E^{''} = \{\{a_i, (v, e_{v[1]}, 1)\}, \{a_i, (v, e_{v[deg(v)]}, 6)\} : 1 \leq i \leq K, v \in V\}$.

El grafo completo $G^{'} = (V^{'}, E^{'})$ tiene $V^{'} = \{a_i : 1 \leq i \leq K\} \cup (\cup_{e \in E} V_e^{'})$ y $E^{'} = (\cup_{e \in E} E_e^{'}) \cup (\cup_{v \in V} E_v^{'}) \cup E^{''}$. $G^{'}$ se puede construir a partir de G y K en tiempo polinomial.

Demostraremos que $G^{'}$ tiene un circuito hamiltoniano si y sólo si G tiene un VC de tamaño menor o igual que K. Supongamos que $< v_1, v_2, ..., v_n >$ donde $n = \mid V^{'} \mid$ es un circuito hamiltoniano para $G^{'}$. Consideremos una porción del circuito que empieza y termina con un vértice en el conjunto $\{a_1, a_2, ..., a_K\}$ y que no contenga otro de estos vértices en su interior. Debido a la restricción mencionada anteriormente sobre la manera en que el circuito hamiltoniano puede pasar por cada componente, esta porción del circuito debe pasar por un conjunto de componentes correspondientes a aquellas ramas de E que son incidentes en un vértice particular $v \in V$. Cada una de las componentes es atravesada en alguna de las tres formas descriptas anteriormente y no se encuentran vértices de otra componente. Los K vértices de $\{a_1, a_2, ..., a_K\}$ dividen al circuito hamiltoniano en K caminos, cada camino se corresponde con un vértice distinto $v \in V$. Como el HC debe incluir todos los vértices de cada una de las componentes, y como los vértices de cada componente para la rama $e \in E$ puede ser atravesada por un único camino que corresponde a un extremo de e, cada rama de E debe tener al menos un extremo entre los K vértices "selectores". Es más, este conjunto de K vértices forma el cubrimiento por vértices de G.

Inversamente, supongamos $V^* \subseteq V$ es un cubrimiento por vértices de G con $\mid V^* \mid \leq K$. Podemos asumir que $\mid V^* \mid = K$, ya que, vértices adicionales de V pueden ser agregados y seguiremos teniendo un cubrimiento por vértices. Supongamos que etiquetamos a los elementos de V^* como v_1, v_2, ..., v_K. Las siguientes ramas son elegidas para estar en el circuito hamiltoniano de $G^{'}$. De la componente que representa cada rama $e = \{u, v\} \in E$, elegir las ramas representadas en la segunda gráfica dependiendo de si $\{u, v\} \cap V^*$ es igual a $\{u\}$, $\{u, v\}$ o $\{v\}$. Una de estas posibilidades debe valer debido a que V^* es un VC de G. Luego tomemos todas las ramas de $E_{v_i}^{'}$ para $1 \leq i \leq K$. Finalmente, agregamos las ramas $\{a_i, (v_i, e_{v_j[1]}, 1)\}$ con $1 \leq i \leq K$,

$\{a_{i+1}, (v_i, e_{v_i[deg(v_i)]}, 6)\}$ con $1 \leq i < K$ y $\{a_1, (v_K, e_{v_K[deg(v_K)]}, 6)\}$. El conjunto de estas ramas se corresponden con un circuito hamiltoniano de G'. □

1.3.4. Clase NP- Hard

Definición: Un problema Π es $\mathcal{NP}$- Hard si todo problema de $\mathcal{NP}$ se puede reducir polinomialmente a Π.

Observemos que si un problema Π_1 es $\mathcal{NP}$- Completo y Π_1 se reduce polinomialmente a Π_2 entonces Π_2 es $\mathcal{NP}$- Hard.

Citaremos algunos problemas que se encuentran dentro de esta clase:

1. En la sección 1.3.3 probamos que HC se reduce polinomialmente a TSPD y además que HC es $\mathcal{NP}$- Completo con lo que podemos concluir que TSPD es $\mathcal{NP}$- Hard.

2. Job Scheduling es $\mathcal{NP}$- Hard pues Partición, que es $\mathcal{NP}$- Completo, se reduce a él [2].

   JS: Se tienen m máquinas iguales, n trabajos y p_i representa el tiempo que tarda el trabajo i (en cualquiera de las máquinas tarda lo mismo).

   P: Dados p_1, p_2,...,p_n ¿existe $I \subset \{1, 2, ..., n\}$ tal que $\sum_{i \in I} p_i = \sum_{i \in I'} p_i$?

3. Como los problemas de decisión Partición, Knapsack y Bin Packing son $\mathcal{NP}$- Completos puede verse que sus correspondientes problemas de optimización son $\mathcal{NP}$-Hard.

Capítulo 2

Clases de algoritmos, el método branch and bound

2.1. Categorías de algoritmos

Como comentamos anteriormente para los problemas $\mathcal{NP}$-Hard no se conocen algoritmos eficientes (de tiempo polinomial) que hallen la solución exacta. Es por eso que intentaremos el uso de métodos eficientes que sólo pretenden una solución aproximada. Existen dos categorías de tales algoritmos:

$*$ la primer categoría consiste de algoritmos polinomiales que encuentran soluciones que si bien no son óptimas, a lo sumo difieren del óptimo en un cierto porcentaje calculable. Indiquemos con A a un algoritmo de tiempo polinomial aplicado a un problema de minimización $\mathcal{NP}$-Hard que obtiene un valor A(I) aproximado del óptimo para cada instancia I del problema y sea Opt(I) el valor óptimo. Estamos interesados en acotar el error relativo de A(I).

Decimos que el problema es aproximable si existe $k \geq 1$ tal que para cualquier instancia I del problema: $\mathrm{A(I)} \leq k\ \mathrm{Opt(I)}$, o en el caso que se tratara de un problema de maximización $\mathrm{A(I)} \geq k\ \mathrm{Opt(I)}$ con $0 < k \leq 1$.

Veremos tres casos:

1. se satisface la desigualdad para algún $k \geq 1$. En este caso decimos que el problema es k-aproximable.

2. se satisface la desigualdad para cualquier $k \geq 1$ prefijado.

3. no se puede acotar el error en la forma antes descripta. En este caso, decimos que el problema no es aproximable.

En la primera parte de este capítulo veremos ejemplos de este tipo de algoritmos llamados de *aproximación*.

∗ la segunda categoría consiste de algoritmos que encuentran en tiempo polinomial una solución que uno espera sea "buena"pero cuya distancia al óptimo se desconoce. A estos algoritmos se los conoce con el nombre de *heurísticos*. Los métodos utilizados para diseñar tales algoritmos tienden a estar relacionados con cada uno de los problemas en forma específica, sin embargo, se pueden identificar una serie de principios generales. Una de las técnicas que más se aplica es la de búsqueda local. Partiendo de una solución inicial S_0 (generalmente elegida al azar) se busca una solución mejor S dentro de un entorno o vecindad de S_0. Una vez hallada S se reemplaza S_0 por S y se repite el procedimiento hasta que no se obtiene ninguna solución mejor en el entorno, obteniéndose de esta manera una solución óptima local. En la práctica, estos algoritmos son utilizados con éxito aunque comúnmente se les practican refinamientos para lograr una performance satisfactoria. Por ejemplo, para controlar el tiempo de ejecución y que el algoritmo resulte de tiempo polinomial, se agregan restricciones sobre la cantidad de iteraciones. Es difícil predecir qué tan buenos serán analizándolos de antemano. Usualmente son evaluados y comparados con una combinación de estudios empíricos.

2.1.1. Algoritmos de aproximación

Empezaremos dando una descripción formal de lo que entendemos por problema de optimización.

Notemos que un problema de optimización combinatoria Π es un problema de minimización o de maximización que contiene las siguientes tres partes:
∗ un conjunto D_Π de instancias,
∗ para cada instancia $I \in D_\Pi$, un conjunto finito $S_\Pi(I)$ de posibles candidatos a solución para I y
∗ una función m_Π que asigna a cada instancia $I \in D_\Pi$ y cada candidato a solución $\sigma \in S_\Pi$ (I) un número racional positivo $m_\Pi(I, \sigma)$, llamado el valor solución para σ.

Si Π es un problema de minimización (o de maximización), entonces una solución óptima para una instancia $I \in D_\Pi$ es un candidato a solución $\sigma^* \in S_\Pi(I)$ tal que, para todo $\sigma \in S_\Pi(I)$, $m_\Pi(I, \sigma^*) \leq m_\Pi(I, \sigma)$ ($m_\Pi(I, \sigma^*) \geq m_\Pi(I, \sigma)$). Usaremos $\mathrm{Opt}_\Pi(I)$ para denotar el valor $m_\Pi(I, \sigma^*)$ de una solución óptima para I.

Dado un algoritmo A que para cualquier instancia $I \in D_\Pi$ encuentra un candidato a solución $\sigma_I \in S_\Pi(I)$, denotemos por A(I) al valor $m_\Pi(I, \sigma_I)$. Si además existe $k \geq 1$ tal que $A(I) \leq k\ \mathrm{Opt}(I)\ \forall\, I \in D_\Pi(I)$ (o $0 < k \leq 1$ y $A(I) \geq k\ \mathrm{Opt}(I)\ \forall\, I \in D_\Pi(I)$ si se tratara de un problema de maximización) diremos que A es un algoritmo de aproximación para Π. Notemos que en el caso que $k = 1$ el algoritmo A resuelve el problema en forma exacta (encuentra el óptimo).

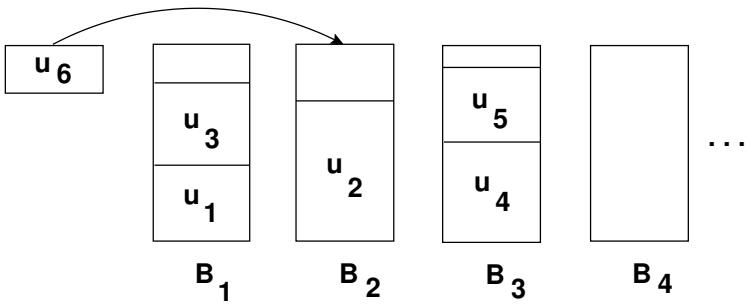
Ejemplificaremos todas estas definiciones en el TSP. En principio, se trata de un problema de minimización, una instancia I es un conjunto finito de ciudades junto con la distancia existente entre ellas. Los candidatos a solución para una instancia particular son todas las permutaciones de las ciudades. El valor solución para cada permutación es la distancia total del correspondiente tour. Un algoritmo de aproximación A para este

problema necesita encontrar solamente alguna permutación de las ciudades cuyo valor A(I) esté "cerca"de Opt(I) (es decir, A(I) $\leq k$ Opt (I) para un valor de k prefijado de antemano), mientras que un algoritmo de optimización debe siempre hallar una permutación que corresponda a un tour de mínima distancia.

En los siguientes ejemplos veremos algoritmos de aproximación A que corren en tiempo polinomial.

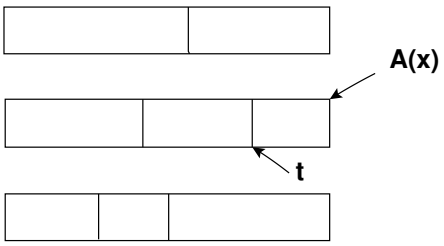
Bin packing: Dado un conjunto finito $U = \{u_1, u_2, ..., u_n\}$ de items y un número racional $a_i = a(u_i) \in [0,1]$ para cada item $u_i \in U$, encontrar una partición de U en subconjuntos disjuntos $U_1, U_2, ..., U_k$ tal que la suma de los tamaños de los items en cada U_i es menor o igual que 1 y el k es lo más pequeño posible. Podemos pensar cada subconjunto U_i como un conjunto de items a ser colocados en camiones de capacidad 1. Nuestro objetivo es ubicar los items de U utilizando la menor cantidad posible de camiones.

Un algoritmo de 2-aproximación que lo resuelve en tiempo polinomial es el **FirstFit**. Imaginemos que empezamos con una secuencia infinita $B_1, B_2, ...$ de camiones de capacidad 1, todos vacíos. El algoritmo ubica los items en los camiones, uno por vez, siguiendo la siguiente regla: siempre poner el próximo item u_i en el camión de menor índice para el cual la suma de los pesos de los items no excede $1 - a(u_i)$. En otras palabras, u_i es siempre colocado en el primer camión en donde cabe sin exceder su capacidad. Veamos como ejemplo el gráfico, donde cada item es representado por un rectángulo cuyo peso es proporcional a su tamaño.



Intuitivamente parece ser un algoritmo razonable y natural. No se comienza un nuevo camión hasta que todos los otros están casi completos. Llamemos w_i al peso que contiene el camión i usando A y $\overline{w}_i$ al peso que contiene el camión en el óptimo. Notemos que $a_1 + ... + a_n \leq \text{Opt}(x)$, en efecto, si $r = \text{Opt}(x)$ entonces $a_1 + ... + a_n = \overline{w}_1 + ... + \overline{w}_r \leq r = \text{Opt}(x)$ pues $\overline{w}_i \leq 1$ para $1 \leq i \leq r$. Además $w_i \leq 1/2$ para a lo sumo un i (si $w_i \leq 1/2$ y $w_j \leq 1/2$ entonces habríamos puesto todo en un solo camión) por lo tanto $w_i \geq 1/2$ para todo $i \neq i_0$. Sea $k = \text{A}(x)$ entonces $a_1 + ... + a_n = w_1 + ... + w_k \geq 1 - w_{i_1} + \sum_{i \neq i_0} w_i = 1 + \sum_{i \neq i_1, i_0} w_i \geq 1 + (k-2)/2 = k/2$ (pues $w_{i_0} \geq 1 - w_{i_1}$ si no, hubiéramos puesto la carga del camión i_0 en el camión i_1, o viceversa). Luego $\text{A}(x) = k \leq 2(a_1 + ... + a_n) \leq 2$ Opt(x).

Job scheduling: Consideremos m máquinas iguales y n trabajos $J_1, J_2, ..., J_n$ que requieren $p_i (i = 1, 2, ..., n)$ tiempo en cualquiera de las m máquinas. Se trata de asignar los trabajos a las máquinas de forma que el tiempo total para hacer todos los trabajos sea mínimo. Este problema es $\mathcal{NP}$-Hard. Un algoritmo $(2 - 1/m)$-aproximado es el siguiente. Tenemos los trabajos en una lista ordenados de cualquier manera, ponemos el J_1 en la máquina 1, el J_2 en la máquina 2 y seguimos así hasta poner el J_m en la máquina m. Si J_i es el primero de la lista que falta procesar, ponerlo en la primera máquina que se desocupa. Sea $\mathrm{Opt}(x)$ la solución óptima y $\mathrm{A}(x)$ la solución dada usando este algoritmo. Es claro que el caso más favorable sería que ninguna máquina quedara parada como indica la figura.



Notemos que $p_i \leq \mathrm{Opt}(x)\ \forall i$. Sean $I_1, ..., I_m$ tal que $I_j = \{i / J_i$ se procesa en la máquina $j\}$ entonces $\sum_{i \in I_j} p_i =$ momento en que termina el proceso en la máquina j, $\sum_{i \in I_j} p_i \leq \mathrm{Opt}(x)\ \forall j$, y $\sum_{i=1}^{n} p_i = \sum_{j=1}^{m} \sum_{i \in I_j} p_i \leq m\ \mathrm{Opt}(x)$; entonces $\sum_{i=1}^{n} p_i / m \leq \mathrm{Opt}(x)$. Sea t el instante en que se comienza a procesar el trabajo J_k que termina último en una cierta máquina S y $\mathrm{A}(x)$ el instante en que termina. En el instante t todas las máquinas distintas de S deben estar funcionando. Vale que $t \leq \sum_{i \in I_j} p_i$ $\forall j \neq S$ (si $j \neq S$ entonces el momento en que se termina el proceso en la máquina j es posterior al momento en que se comienza el trabajo k en la máquina S). Entonces $\sum_{i=1}^{n} p_i = \sum_{i \in I_S} p_i + \sum_{j \neq S} \sum_{i \in I_j} p_i \geq t + p_k + (m-1)t = mt + p_k$. Así, $\sum_{i \neq k} p_i \geq mt$ y por lo tanto $t \leq \sum_{i \neq k} p_i / m$.

Luego $\mathrm{A}(x) = t + p_k \leq \sum_{i \neq k} p_i / m + p_k = \sum_{i=1}^{n} p_i / m + (1 - 1/m) p_k \leq \mathrm{Opt}(x) + (1 - 1/m)\ \mathrm{Opt}(x) \leq (2 - 1/m)\ \mathrm{Opt}(x)$.

Problema del cubrimiento por vértices: Se trata de hallar un mínimo conjunto C^* de vértices de un grafo G tal que toda rama de G sea incidente en un vértice de C^*. Daremos dos ejemplos. El primero es 2-aproximable. En cambio, para el segundo no existe k tal que sea k-aproximable.

Algoritmo:
0) $C = \emptyset$ $E = E(G)$
1) elegir $(u, v) \in E$, $G = G - \{u, v\}$ (al grafo G se le extraen los vértices u, v y todas las ramas que inciden en ellos), $E = E(G)$, $C = C \bigcup \{u, v\}$
2) si $E \neq \emptyset$ ir a 1

Sea C^* un cubrimiento mínimo y C el hallado por el algoritmo. Observemos que $\mid C \mid /2$ es el número de ramas elegidas por el algoritmo que no tienen vértices comunes.

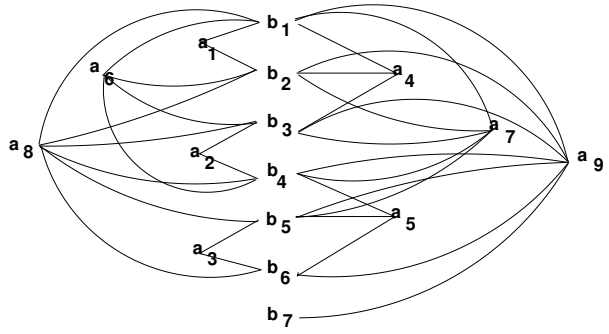
Como C^* es un cubrimiento entonces por lo menos un extremo de cada una de esas ramas pertenece a C^*, así que $\mid C^* \mid \geq \mid C \mid /2$ entonces $\mid C \mid \leq 2 \mid C^* \mid$.

Sobre este ejemplo veremos un algoritmo cuyo error porcentual es tan grande como se quiera.

Algoritmo:
$C = \emptyset$
1) elegir en G un nodo u de máximo grado, $C = C \bigcup \{u\}$, $G = G - \{u\}$
2) si $V(G) \neq \emptyset$ ir a 1

Este algoritmo parece mejor que el anterior, sin embargo, podemos construir un $G = (V, E)$ tal que el error k del algoritmo precedente sea tan grande como se quiera. Construimos G de la siguiente manera:
Vértices: $V = \{b_1, ..., b_n, a_1, ..., a_s\}$ con $s = [n/2] + [n/3] + [n/4] + ... + [n/n-1] + 1$
Ramas: particionamos los b_i en conjuntos de 2 vértices y unimos cada par con un nuevo vértice a_t, $t = 1, 2, ..., [n/2]$. Luego particionamos los b_i en conjuntos de 3 vértices y unimos cada terna con un nuevo vértice a_w, $w = [n/2] + 1, [n/2] + 2, ..., [n/2] + [n/3]$. Continuamos con este procedimiento hasta que finalmente unimos los vértices b_1, ..., b_n con un nuevo vértice a_s.

Graficaremos la situación en el caso en que $n = 7$.



Observemos que:
$*$ $deg\ a_1 = 2$, ..., $deg\ a_{[n/2]} = 2$
$*$ $deg\ a_{1+[n/2]} = 3$, ..., $deg\ a_{[n/2]+[n/3]} = 3$
.
.
.
$*$ $deg\ a_s = n$
$*$ $deg\ b_i \leq n - 1$

El algoritmo pone a_s en C. Sacando a_s el grado de los b_i's baja en 1, ahora el nodo de mayor grado es a_{s-1}. Siguiendo con este razonamiento, cuando quedan a_1, ..., $a_{[n/2]}$ y los b_i's el $deg\ a_t = 2$ y $deg\ b_i \leq 1$ entonces elige los a_t. Por lo tanto, el cubrimiento encontrado por el algoritmo es $C = \{a_1, ..., a_s\}$ donde $s \simeq n \log n$.

Si existe $k \geq 1$ tal que $\mathrm{A}(x) \leq k\ \mathrm{Opt}(x)\ \forall x$ entonces $n \log n \simeq s = \mathrm{A}(x) \leq k\ \mathrm{Opt}(x) \leq kn$, pues $\{b_1, ..., b_n\}$ es un cubrimiento, pero entonces vale que $\log n \leq k\ \forall n$ lo cual es un absurdo.

Knapsack optimización: Supongamos que tenemos n items y conocemos el peso p_i y el valor v_i de cada uno de ellos. Queremos llenar una mochila bajo ciertas condiciones para lo cual debemos elegir un subconjunto $S \subset \{1, 2, ..., n\}$ tal que la suma de los valores v_i sea máxima y la suma de los pesos p_i no supere a K (peso máximo que puede soportar la mochila). Como dijimos al comienzo p_i, v_i y K son dato y además debe valer que $p_i \leq K\ \forall i$. Llamemos (1) al problema planteado anteriormente. Mostraremos a continuación que dado $k \geq 1$ existe un algoritmo polinomial que soluciona toda instancia de Knapsack optimización con un error menor que k. Primero resolveremos el problema en forma exacta con un algoritmo pseudopolinomial mediante programación dinámica. Para $m = 1, 2, ..., \sum v_i$, sean
$f_j(m)$ = peso mínimo de un subconjunto de $\{1, 2, ..., j\}$ cuya suma de valores es igual a m, o bien
$f_j(m) = \infty$ si no existe tal subconjunto.

Usaremos las condiciones de borde $f_j(0) = 0$ $(j = 1, 2, ..., n)$. Para cada $m = 1, 2, ..., C$ $(C = \sum_i v_i)$ calculamos $f_j(m) = \mathrm{mín}(f_{j-1}(m), p_j + f_{j-1}(m - p_j))$ para $j = 2, ..., n$. El tiempo de ejecución del algoritmo es proporcional al número de veces que calculamos f, es decir, $O(nC)$. Si llamamos $v_{\mathrm{máx}} = \mathrm{máx}_i\, v_i$, como $C \leq nv_{\mathrm{máx}}$, el tiempo de ejecución es $O(n^2 v_{\mathrm{máx}})$.

Dado $k \geq 1$ veamos como podemos conseguir una solución aproximada reduciendo el tiempo de ejecución. En (1) sustituimos v_i por $\lfloor v_i/w \rfloor$ donde w es un factor de escala, por ejemplo, $w = 10^t$. Determinamos w de forma que el máx de (1) quede determinado con un error relativo k, es decir, tal que $k = (\sum_{j \in S^*} v_j - \sum_{j \in S_w} v_j)/(\sum_{j \in S^*} v_j)$ siendo S^* el subconjunto de $\{1, 2, ..., n\}$ óptimo de (1) y S_w el subconjunto de $\{1, 2, ..., n\}$ óptimo de (1) con la sustitución anterior. Vale que:

$$\sum_{j \in S_w} v_j \geq \sum_{j \in S_w} w \lfloor v_j/w \rfloor$$

porque $v_j \geq w \lfloor v_j/w \rfloor$,

$$\sum_{j \in S_w} w \lfloor v_j/w \rfloor \geq \sum_{j \in S^*} w \lfloor v_j/w \rfloor$$

pues $\sum_{j \in S_w} \lfloor v_j/w \rfloor \geq \sum_{j \in S^*} \lfloor v_j/w \rfloor$,

$$\sum_{j \in S^*} w \lfloor v_j/w \rfloor \geq \sum_{j \in S^*} (v_j - w)$$

ya que $w \lfloor v_j/w \rfloor \geq v_j - w$, y como

$$\sum_{j \in S^*} (v_j - w) = \sum_{j \in S^*} v_j - w \mid S^* \mid \geq \sum_{j \in S^*} v_j - wn,$$

se deduce que $\sum_{j\in S^*} v_j - \sum_{j\in S_w} v_j \leq wn$. Luego,

$$k = (\sum_{j\in S^*} v_j - \sum_{j\in S_w} v_j)/(\sum_{j\in S^*} v_j) \leq wn/v_{\text{máx}},$$

lo que muestra que fijando w podemos obtener un error relativo prefijado. A su vez queda disminuido el tiempo de ejecución en $\mathrm{O}(n^2 v_{\text{máx}}/w)$.

TSP:
1) Si $\mathcal{P} \neq \mathcal{NP}$ entonces el TSP no es aproximable en tiempo polinomial.

Veremos que si existe un algoritmo A de tiempo polinomial tal que $\mathrm{A}(x) \leq k\ \mathrm{Opt}(x)$ para el TSP entonces A resuelve HC (por lo tanto, HC sería polinomial).

Sea G un grafo, supongamos que existe A que cumple lo antes mencionado. Si definimos $c_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 2 + n(k-1) & \text{si no} \end{cases}$ entonces tenemos un grafo completo con costos. Aplicamos A, pueden ocurrir sólo dos cosas:

1. Todas las ramas del ciclo tienen costo = 1. Entonces $\mathrm{A}(x) = n$ en cuyo caso G tiene un HC.

2. Alguna rama tiene costo $2 + n(k-1)$ y las restantes costo ≥ 1. Por lo tanto $\mathrm{A}(x) \geq n - 1 + 2 + n(k-1) = kn + 1 > kn$ por lo tanto G no puede tener un HC (si lo tuviera entonces $\mathrm{Opt}(x) = n \therefore kn = k\ \mathrm{Opt}(x)$).

Se deduce que no existe algoritmo que aproxime a TSP.

2.1.2. Algoritmos heurísticos

Por lo comentado en la sección anterior un problema de optimización lo indicaremos mediante el par (F, c) donde F es el conjunto de soluciones factibles de una instancia de un problema de optimización y $c : F \to \mathbb{R}$ es la función costo.

El objetivo es encontrar un $f_0 \in F$ tal que $c(f_0) \leq c(f)$ para todo $f \in F$. La base del método consiste en asignar a cada $t \in F$ un entorno de t, es decir, un subconjunto $N(t) \subset F$ que contiene a t y a cuyos elementos los consideramos "vecinos" de t.

El algoritmo de *búsqueda local* consiste en partir de una solución inicial t_0, intentar encontrar un $s \in N(t)$ con costo menor para de esta forma disminuir $c(t)$, y repetir este procedimiento hasta llegar a un t en cuyo entorno resulte imposible disminuir el costo. Si esto sucediera estaríamos en presencia de un mínimo local. Observemos que hay por lo menos dos maneras de definir s en cada iteración: hallar la primera s que mejora el costo o hallar una s que minimiza el costo sobre todo $N(t)$.

Este algoritmo presenta los siguientes problemas:
$*$ cómo debe elegirse una solución t_0 inicial. A veces conviene usar varias t iniciales en cuyo caso el problema es cuántas y cómo distribuirlas en F.

$*$ definir el entorno $N(t)$ para cada t es un problema cuya resolución requiere de intuición. Además existe una disyuntiva entre usar N grande o chico. Grande consume mucho tiempo de búsqueda pero el mínimo local es de buena calidad, en cambio, chico es rápido pero el mínimo puede ser pobre.

Estas decisiones se toman en forma empírica.

Cuando el mínimo en cualquier entorno es también un mínimo global decimos que la función $N(t)$ es exacta.

Daremos ejemplos de entornos en algunos de los problema con los cuales estuvimos trabajando.

TSP:
$F = \{f$: f es un circuito hamiltoniano en el grafo completo $G\}$
$N_2(f) = \{g$: g es un circuito hamiltoniano que resulta de suprimir dos ramas en f y reemplazarlas por otras dos que formen de nuevo un circuito en $G\}$.

Bubble sort:

Tenemos una sucesión de números $x_1, x_2, ..., x_n$ y queremos ordenarlos de menor a mayor. El bubble sort es un algoritmo que se puede utilizar para resolver este problema. Compara x_i con x_{i+1} $(i = 1, 2, ..., n-1)$ y los intercambia si $x_i > x_{i+1}$, este procedimiento debe repetirse $n-1$ veces.
$F = \{f$: f es una permutación de la sucesión $\}$
$N_k(f) = \{g$: permutación que resulta de intercambiar k pares distintos en $f\}$.

Con $k = 1$ se obtiene el bubble sort. Observemos que el entorno definido es exacto.

2.2. El método de branch and bound

Un árbol dirigido con raíz es un grafo dirigido, conexo y acíclico que tiene un vértice distinguido s al que llamamos raíz tal que para cualquier otro vértice v hay un camino dirigido de s a v. Si (u, v) es una rama de un árbol dirigido con raíz diremos que u es el padre de v y que v es el hijo de u. Dado un vértice u, el conjunto de vértices v tales que existe un camino dirigido de u a v se llama la descendencia de u. Decimos que el árbol es binario si cada vértice que no sea una hoja tiene exactamente dos hijos.

Consideremos el siguiente problema: dado un árbol dirigido con raíz s, donde cada vértice x tiene asignado un costo $c(x)$ que satisface $c(x) \leq c(y)$ para toda rama (x, y), queremos hallar una hoja de mínimo costo. Describiremos un algoritmo, conocido como *branch and bound*, que resuelve este problema. El procedimiento utiliza una mezcla de backtracking (volver al vértice anterior para examinar alguno de sus hijos que todavía no ha sido examinado) y un criterio particular de poda que consiste en eliminar toda la descendencia de un vértice x cuando se satisfaga que $c(x) \geq c(h)$ para alguna hoja h encontrada previamente (es decir, cuando ninguna hoja descendiente de x puede ser la solución óptima del problema).

Descripción del algoritmo:
0) $L = \{s\}$, $c = \infty$
1) si x es, de los elementos de L, el último que ingresó, $L = L - \{x\}$. Calcular $c(x)$. Si $c(x) \geq c$, ir a 4
2) Si x no es una hoja, $L = L \cup \{$ hijos de $x\}$, ir a 1 (conviene usar una lista para L pues interesa el orden)
3) $h = x$, $c = c(x)$
4) si $L \neq \emptyset$ ir a 1
5) stop

El valor de c en cada iteración del algoritmo es el costo de la hoja más barata encontrada hasta el momento y h guarda la información de cuál es la hoja cuyo costo es c (cuando el algoritmo encuentra una hoja de costo menor que el valor de c presente en ese momento, actualiza c y h). Si al examinar un nodo i resulta que $c(i) \geq c$ entonces toda su descendencia es podada ya que ninguna hoja que sea descendiente de i puede tener costo menor que c, es decir, costo menor que el de la hoja más barata hallada hasta ese momento. Como una hoja no es examinada por el algoritmo sólo cuando es seguro que no puede ser una solución óptima entonces en alguna iteración del algoritmo una hoja de mínimo costo es examinada. Cuando el algoritmo encuentra la primer hoja de mínimo costo los valores de h y c son actualizados ya que su costo es menor que el valor de c que estaba presente en ese momento, y a partir de allí c y h permanecen constantes hasta terminar el algoritmo ya que ninguna otra hoja encontrada más tarde puede tener costo menor que el de esta hoja de mínimo costo. Luego, al terminar el algoritmo la hoja que se encuentra almacenada en h es una hoja de mínimo costo y $c = c(h)$.

2.2.1. Aplicación del método branch and bound a programación lineal entera

Consideremos el problema de programación lineal

$$\begin{gathered} \min cx \\ \text{sujeto a } Ax \leq b \\ 0 \leq x_j \leq u_j \ (1 \leq j \leq n) \end{gathered}$$

donde $u_j < \infty$ para todo j y consideremos el problema de programación lineal entera que resulta de agregar la restricción adicional: x_i entero $(1 \leq i \leq m)$ donde $m \leq n$ es un número natural dado. Resolveremos este problema de P.L.E. generando un árbol binario dirigido cuya raíz será el problema original y los restantes vértices serán subproblemas que resulten de agregar ciertas restricciones del tipo $x_j \leq k$ o $x_j \geq k+1$ para algunos j. Las hojas serán los subproblemas cuya solución óptima verifica la restricción adicional y los subproblemas que no sean factibles (se dice que x es una solución *factible* si verifica todas las restricciones del problema). Para cada vértice u definiremos $c(u)$ como el valor

del funcional en la solución óptima del subproblema u si éste es factible y definiremos $c(u) = \infty$ si el subproblema u no es factible. La función $c(u)$ verificará $c(u) \leq c(v)$ para toda rama (u, v) del árbol. Usaremos el método de branch and bound para encontrar una hoja de mínimo costo y eso nos dará la solución del problema de P.L.E.

Observaciones:

* La condición $0 \leq x_j \leq u_j$ asegura que el poliedro definido por las restricciones es acotado. Como también es cerrado (pues es intersección de semiespacios de la forma $\{x/ax \leq b\}$ o $\{x/ax \geq b\}$ que son cerrados) entonces es compacto. Luego, si el problema original es factible entonces siempre existe una solución óptima ya que el funcional es una función continua y toda función continua sobre un compacto alcanza su máximo y su mínimo. Lo mismo vale para cualquier subproblema que resulte de agregar al problema original restricciones del tipo $x_j \leq k$ o $x_j \geq k + 1$ para algunos j.
* Si z^* es el valor del funcional en una solución óptima de un problema u de programación lineal y z es el valor del funcional en el óptimo de cualquier subproblema v que se obtenga agregándole una restricción a u entonces $z^* \leq z$ ya que toda solución óptima del subproblema v es una solución factible del problema u.

Veamos con más detalle cómo construir el árbol. La raíz s del árbol es el problema original. Notemos que toda solución factible del problema de P.L.E. es una solución factible de s. Supongamos que hemos construido una parte del árbol donde cada vértice es un subproblema que resulta de agregar al problema original restricciones del tipo $x_j \leq k$ o $x_j \geq k + 1$ para algunos j, donde k es un entero no negativo y tal que toda solución factible del problema de P.L.E. es una solución factible de alguna hoja de este subárbol. Para cada subproblema u que sea una hoja del subárbol que tenemos construido hasta ahora (es decir, cada vértice u que no tenga hijos) hacemos lo siguiente: si u tiene una solución óptima que no satisface la condición adicional, sea j $(1 \leq j \leq m)$ el primer índice para el cual x_j no es entero y sea r un entero tal que $r < x_j < r + 1$. Entonces creamos dos hijos de u, uno agregando al problema u la restricción $x_j \leq r$ y el otro agregando a u la restricción $x_j \geq r + 1$. Notemos que si una solución factible del problema de P.L.E. es una solución factible de u entonces es una solución factible de alguno de los hijos de u ya que, como r es entero, si la j-ésima coordenada de esta solución es entera entonces o bien es menor o igual que r o bien es mayor o igual que $r + 1$. Si en cambio u tiene una solución óptima que satisface la restricción adicional o si no tiene soluciones factibles entonces no creamos ningún hijo de u. De esta manera obtenemos un árbol binario dirigido con raíz s cuyas hojas son los subproblemas cuyas solución óptima verifica la restricción adicional y los subproblemas que no sean factibles. Además, toda solución factible del problema de P.L.E. es una solución factible de alguna hoja del árbol. Para cada vértice u de este árbol definimos $c(u)$ como el valor del funcional en la solución óptima del subproblema u si éste es factible y $c(u) = \infty$ si no. Debido a la forma en que hemos construido el árbol esta función satisface $c(u) \leq c(v)$ para toda rama (u, v) del árbol ya que si (u, v) es una rama del árbol entonces v es un subproblema que se obtuvo agregando una restricción a

u. De esta manera, el costo de una hoja que corresponda a un problema factible será el valor del funcional en una solución factible del problema de P.L.E.. Como las soluciones factibles del problema de P.L.E. son las soluciones factibles de cada una de las hojas del árbol entonces una hoja de mínimo costo será una solución óptima del problema de P.L.E..

Podemos plantear el problema del viajante como un problema de programación lineal entera. Tenemos un grafo dirigido completo $G = (V, E)$ donde $V = \{0, 1, 2, ..., n\}$ y asignamos costo c_{ij} a cada rama (i, j). Dado un circuito hamiltoniano C, para cada (i, j) sea

$$\delta_{ij} = \begin{cases} 1 & \text{si } (i,j) \in C \\ 0 & \text{si } no. \end{cases}$$

Entonces el costo de C es $\sum c_{ij}\delta_{ij}$. Observemos que para cada vértice i hay una sola rama de C cuya cola es i y una sola rama de C cuya punta es i. Luego se satisface

$$\sum_j \delta_{ij} = 1 \ (0 \leq i \leq n)$$

$$\sum_i \delta_{ij} = 1 \ (0 \leq j \leq n)$$

Supongamos ahora que para cada (i, j) tenemos definido un δ_{ij} tal que $\delta_{ij} = 0$ o $\delta_{ij} = 1$ y de manera tal que valga lo anterior. Si el conjunto de ramas (i, j) tales que $\delta_{ij} = 1$ fuese un circuito entonces podríamos pensar a cada circuito hamiltoniano como una solución factible de

$$\sum_j \delta_{ij} = 1 \ (0 \leq i \leq n)$$

$$\sum_i \delta_{ij} = 1 \ (0 \leq j \leq n)$$

$$0 \leq \delta_{ij} \leq 1, \ \delta_{ij} \text{ entero}$$

cuyo costo es $\sum c_{ij}\delta_{ij}$. Lamentablemente esto no es así, ya que podría ser una solución factible pero el conjunto de ramas (i, j) tales que $\delta_{ij} = 1$ pueden no formar un circuito sino más. Por ejemplo, si consideramos $V = \{0, 1, 2, 3, 4, 5\}$, tomando $\delta_{01} = \delta_{20} = \delta_{12} = \delta_{34} = \delta_{45} = \delta_{53} = 1$ y los restantes δ_{ij} iguales a cero, el conjunto (i, j) tales que $\delta_{ij} = 1$ es $\{(0, 1), (2, 0), (1, 2), (3, 4), (4, 5), (5, 3)\}$. En este caso δ_{ij} $(0 \leq i, j \leq 5)$ es una solución factible pero el conjunto de ramas no forman un circuito sino dos subcircuitos.

Esto puede evitarse definiendo unas variables reales nuevas, u_i $1 \leq i \leq n$, a las que les pediremos que cumplan la condición $u_i - u_j + n\delta_{ij} \leq n - 1$ $(1 \leq i, j \leq n, i \neq j)$ lo que nos permitirá plantear el problema del viajante por programación lineal entera. Esto se debe a que, como veremos, si δ_{ij} es una solución factible entonces existen u_i satisfaciendo esta nueva condición si y sólo si el conjunto de ramas (i, j) tales que $\delta_{ij} = 1$ es un circuito (que empieza y termina en la ciudad 0). En tal caso el costo de ese circuito es $\sum c_{ij}\delta_{ij}$.

El problema de programación lineal entera

$$min \sum c_{ij}\delta_{ij}$$

$$\sum_j \delta_{ij} = 1 \ (0 \leq i \leq n)$$

$$\sum_i \delta_{ij} = 1 \ (0 \leq j \leq n)$$

$$u_i - u_j + n\delta_{ij} \leq n-1 \ (1 \leq i,j \leq n, i \neq j)$$

$$0 \leq \delta_{ij} \leq 1, \ \delta_{ij} \text{ entero}$$

resuelve el problema del viajante.

Mostraremos que cada circuito hamiltoniano puede representarse como una solución factible. Supongamos que C es un circuito hamiltoniano. Para cada (i,j) sea

$$\delta_{ij} = \begin{cases} 1 & \text{si } (i,j) \in C \\ 0 & \text{si } no \end{cases}$$

y sea $u_i = 1$ si i es la primera ciudad visitada, $u_i = 2$ si i es la segunda ciudad visitada, etc. Entonces

$$u_i - u_j + n\delta_{ij} = \begin{cases} u_i - u_j & \text{si } (i,j) \notin C \\ -1+n & \text{si } (i,j) \in C \end{cases}$$

ya que si $(i,j) \notin C$ entonces $\delta_{ij} = 0$ y si $(i,j) \in C$ entonces el viajante visita j justo después de visitar i, de donde $u_j = u_i + 1$ y como $(i,j) \in C$ entonces $\delta_{ij} = 1$. Luego, como $1 \leq u_i \leq n$ entonces $u_i - u_j + n\delta_{ij} \leq n-1$ para todo $i \neq j$ $(1 \leq i,j \leq n)$. Esto muestra que cada circuito hamiltoniano determina una solución factible.

Recíprocamente, supongamos ahora que tenemos una solución factible. Probaremos que el conjunto de ramas $C = \{(i,j)/\delta_{ij} = 1\}$ es un circuito hamiltoniano. Como cada δ_{ij} es cero o uno y valen

$$\sum_j \delta_{ij} = 1 \ (0 \leq i \leq n)$$

$$\sum_i \delta_{ij} = 1 \ (0 \leq j \leq n)$$

entonces para cada i hay un único j tal que $\delta_{ij} = 1$ y un único j tal que $\delta_{ji} = 1$ (notemos que la segunda igualdad puede escribirse $\sum_j \delta_{ji} = 1$, $0 \leq i \leq n$). Esto significa que para cada vértice i hay una sola rama de C cuya cola es i y una sola rama de C cuya punta es i. Luego, sólo debemos ver que las ramas de C forman un circuito y no dos o más subcircuitos. Supongamos que formaran dos o más subcircuitos. Entonces podemos elegir uno de ellos que no pase por el vértice 0. Sea E el conjunto de ramas que lo forman

y sea $k =\mid E \mid$. Como $u_i - u_j + n\delta_{ij} \leq n-1$ para todo $1 \leq i,j \leq n$ tal que $i \neq j$ entonces

$$\sum_{(i,j)\in E} u_i - u_j + n\delta_{ij} \leq k(n-1).$$

Pero como para cada i hay una única rama en E cuya cola es i y una única rama cuya punta es i entonces

$$\sum_{(i,j)\in E} u_i - u_j = 0$$

ya que cada u_i aparece una vez sumando y una vez restando. Además, como las ramas de E forman un subcircuito de C entonces $\delta_{ij} = 1$ para todo $(i,j) \in E$. Por lo tanto,

$$\sum_{(i,j)\in E} n\delta_{ij} = kn$$

de donde

$$kn = 0 + kn = \sum_{(i,j)\in E} u_i - u_j + \sum_{(i,j)\in E} n\delta_{ij} = \sum_{(i,j)\in E} u_i - u_j + n\delta_{ij} \leq k(n-1)$$

con lo cual se tiene que $kn \leq k(n-1)$, lo que es un absurdo pues $k > 0$. Luego, cada solución factible determina un ciclo hamiltoniano.

2.2.2. Aplicación de branch and bound al problema del viajante

Si bien podemos resolver el TSP planteándolo como un problema de P.L.E. y aplicando branch and bound, no arroja buenos resultados. Hay otra forma de generar un árbol dirigido con raíz para resolver el problema del viajante que es más eficiente. En realidad, el método de branch and bound surgió justamente para resolver el TSP cuando Little, Murty, Sweeney y Karel idearon esta manera que vamos a contar a continuación.

Sea $G = (V, E)$ un grafo completo dirigido, donde $V = \{1, 2, ..., n\}$ y donde cada rama (i,j) de G tiene asignado un costo c_{ij} tal que $0 \leq c_{ij} \leq \infty$. Recordemos que un circuito hamiltoniano C es un ciclo dirigido en G que pasa por cada vértice una y sólo una vez. El costo $c(C)$ del circuito se define como la suma de los costos de las ramas que lo forman, es decir, $c(C) = \sum_{(i,j)\in C} c_{ij}$. El problema consiste en hallar un circuito hamiltoniano C de mínimo costo.

Como antes, para resolver este problema generaremos un árbol binario dirigido con raíz y definiremos una función $c(u)$ conveniente que satisfaga $c(u) \leq c(v)$ para toda rama

(u, v) de manera que nuestro problema se traduzca en hallar una hoja de mínimo costo. En lo que queda de esta sección, la palabra circuito significará circuito hamiltoniano.

La raíz del árbol será el problema del viajante, es decir, hallar un circuito C tal que $c(C)$ sea mínimo. Supongamos que hemos construido una parte del árbol donde cada vértice es un subproblema que resulta de agregar a su vértice padre la restricción $(i, j) \in C$ o la restricción $(i, j) \notin C$, para cierta rama $(i, j) \in E$. Para cada subproblema u que sea una hoja del subárbol que tenemos construido hasta ahora hacemos lo siguiente: sea $A_u = \{e \in E/$ "$e \in$ C" es una restricción de $u\}$, si $\mid A_u \mid < n-1$ entonces elegimos convenientemente una rama $(i, j) \in E$ y generamos dos hijos de u, uno agregando a u la restricción $(i, j) \in C$ y el otro agregando a u la restricción $(i, j) \notin C$. Si, en cambio, $\mid A_u \mid = n-1$ entonces no generamos ningún hijo de u.

Veamos en un ejemplo cuál es la manera en que se eligen las ramas (i, j) que definen las restricciones de los subproblemas, cómo calcular el costo de cada vértice del árbol que se genera y la razón por la cual el problema se traduce en encontrar una hoja de mínimo costo. En lo que sigue, por número entenderemos un número real o infinito. Sea G el grafo completo dirigido con los $n = 7$ vértices 1, 2, ..., 7. Supongamos que el costo de cada rama (i, j) de G $(1 \leq i, j \leq 7)$ está dado por la matriz

$$\|c_{ij}\| = \begin{pmatrix} \infty & 3 & 93 & 13 & 33 & 9 & 57 \\ 4 & \infty & 77 & 42 & 21 & 16 & 34 \\ 45 & 17 & \infty & 36 & 16 & 28 & 25 \\ 39 & 90 & 80 & \infty & 56 & 7 & 91 \\ 28 & 46 & 88 & 33 & \infty & 25 & 57 \\ 3 & 88 & 18 & 46 & 92 & \infty & 7 \\ 44 & 26 & 33 & 27 & 84 & 39 & \infty \end{pmatrix}$$

Paso 1. Definimos la raíz s del árbol como el problema de hallar un circuito C tal que $c(C)$ sea mínimo.

Paso 2. Calculamos el costo y generamos los hijos de cada nodo u que sea una hoja del subárbol que tenemos construido hasta ahora y que satisfaga que $\mid A_u \mid < n-1$. El subárbol que tenemos construido hasta ahora tiene un solo nodo (la raíz s), que es una hoja. Como $\mid A_s \mid = 0 < 6 = n-1$ pues $A_s = \emptyset$, para calcular $c(s)$ y generar los hijos de s procedemos de la siguiente manera: asociamos a s la matriz $\|c_{ij}(s)\| = \|c_{ij}\|$ y consideremos la matriz D_1 que se obtiene restando un número no negativo k_1 a cada coeficiente de la fila 1 de $\|c_{ij}(s)\|$ de manera que cada coeficiente de la nueva matriz sea no negativo. Como cada circuito C pasa una sola vez por el vértice 1 entonces existe un único j, $1 \leq j \leq 7$, $j \neq 1$ tal que la rama $(1, j) \in C$. Luego el costo de cualquier circuito C calculado utilizando la nueva matriz de costos D_1 es igual a $c(C) - k_1$ ya que la única rama cuyo costo sufrió una modificación es la rama $(1, j)$ cuyo costo fue disminuido en k_1. Hacemos esto con el máximo valor posible de k_1, en este caso $k_1 = 3$, para que la nueva matriz D_1 tenga al menos un coeficiente nulo en la primera fila. Ahora consideremos la matriz D_2 que se obtiene restando un número no negativo k_2 a cada coeficiente de la fila 2 de D_1 de manera que cada coeficiente de la nueva matriz sea

no negativo. Como cada circuito C pasa una sola vez por el vértice 2 entonces existe un único j, $1 \leq j \leq 7$, $j \neq 2$ tal que la rama $(2, j) \in C$. Luego el costo de cualquier circuito C calculado utilizando la nueva matriz de costos D_2 es igual al costo de C calculado usando la matriz de costos D_1 menos k_2, lo que es igual a $c(C) - k_1 - k_2$. Hacemos esto con el máximo valor posible de k_2, en este caso $k_2 = 4$. De esta manera D_2 tiene al menos un coeficiente nulo en cada una de las dos primeras filas. Repitiendo este procedimiento para las restantes filas obtenemos la matriz

$$D_n = D_7 = \begin{pmatrix} \infty & 0 & 90 & 10 & 30 & 6 & 54 \\ 0 & \infty & 73 & 38 & 17 & 12 & 30 \\ 29 & 1 & \infty & 20 & 0 & 12 & 9 \\ 32 & 83 & 73 & \infty & 49 & 0 & 84 \\ 3 & 21 & 63 & 8 & \infty & 0 & 32 \\ 0 & 85 & 15 & 43 & 89 & \infty & 4 \\ 18 & 0 & 7 & 1 & 58 & 13 & \infty \end{pmatrix}$$

que tiene al menos un coeficiente nulo en cada fila. El costo de cualquier circuito C calculado utilizando la nueva matriz de costos D_n es $c(C) - k_1 - k_2 - ... - k_7$, donde $k_1 = 3$, $k_2 = 4$, $k_3 = 16$, $k_4 = 7$, $k_5 = 25$, $k_6 = 3$ y $k_7 = 26$. Ahora para cada j, $(1 \leq j \leq 7)$ restamos un número no negativo r_j a cada coeficiente de la columna j de D_n de manera que cada coeficiente de la nueva matriz sea no negativo. Eligiendo r_j el máximo valor posible, en este caso $r_1 = 0$, $r_2 = 0$, $r_3 = 7$, $r_4 = 1$, $r_5 = 0$, $r_6 = 0$ y $r_7 = 4$, obtenemos la matriz

$$\|c'_{ij}(s)\| = \begin{pmatrix} \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ 0 & \infty & 66 & 37 & 17 & 12 & 26 \\ 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ 32 & 83 & 66 & \infty & 49 & 0 & 80 \\ 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ 0 & 85 & 8 & 42 & 89 & \infty & 0 \\ 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}$$

que tiene al menos un coeficiente nulo en cada fila y en cada columna. El costo de cualquier circuito C calculado utilizando la nueva matriz de costos $\|c'_{ij}(s)\|$ definido por $c'(C, s) = \sum_{(i,j)\in C} c'_{ij}(s)$ satisface $c'(C, s) = c(C) - k_1 - ... - k_7 - r_1 - ... - r_7$. Definimos $c(s) = k_1 + ... + k_n + r_1 + ... + r_n$. En este caso, $c(s) = 96$. De esta manera,

$$c(C) = c'(C, s) + c(s) = c'(C, s) + 96.$$

Observación: Para cualquier circuito C se verifica que $c(C) \geq c(s)$. En efecto, como los coeficientes de $\|c'_{ij}(s)\|$ son no negativos entonces $c'(C, s) \geq 0$. Luego, $c(C) = c'(C, s) + c(s) \geq c(s)$.

Ahora generamos dos hijos u_1 y v_1 de s, eligiendo una rama $(i_1, j_1) \in E$ tal que el coeficiente $c'_{i_1,j_1}(s)$ correspondiente a la fila i_1 y a la columna j_1 de $\|c'_{ij}(s)\|$ sea nulo, y tomando como u_1 el subproblema que resulta de agregar al problema s la

condición de que (i_1, j_1) pertenezca al circuito y como v_1 el que resulta de agregar a s la condición de que (i_1, j_1) no pertenezca al circuito. Supongamos que en este caso elegimos $(i_1, j_1) = (4, 6)$. Entonces el subproblema u_1 es el de hallar un circuito de mínimo costo que contenga a la rama (4, 6) y v_1 el de hallar un circuito de mínimo costo que no la contenga.

Paso 3. Calculamos el costo y generamos los hijos de cada nodo u que sea una hoja del subárbol que tenemos construido hasta ahora y que satisfaga que $\mid A_u \mid < n-1$. El subárbol que tenemos construido hasta ahora consiste de la raíz s y sus dos hijos, u_1 y v_1. Las hojas son, por lo tanto, u_1 y v_1. Como $\mid A_{u_1} \mid = 1 < 6 = n-1$ pues $A_{u_1} = \{(i_1, j_1)\}$, para calcular $c(u_1)$ y luego generar sus hijos le asociamos al problema u_1 la matriz de costos $\|c_{ij}(u_1)\|$ que se obtiene reemplazando en $\|c'_{ij}(s)\|$ el coeficiente $c'_{j_1 i_1}(s)$ por ∞ y eliminando la fila i_1 y la columna j_1. Si C es un circuito que contiene a la rama (i_1, j_1) entonces para todo $j \neq j_1$ la rama (i_1, j) no puede pertenecer a C, para todo $i \neq i_1$ la rama (i, j_1) no puede pertenecer a C. Como además $c'_{i_1 j_1} = 0$ eso significa que no necesitamos guardar la información sobre la fila i_1 y la columna j_1 de $\|c'_{ij}(s)\|$ y por eso son eliminadas. Como además la rama (j_1, i_1) no puede pertenecer al circuito, para evitar que esto pudiera ocurrir reemplazamos el coeficiente $c'_{j_1 i_1}(s)$ por ∞. Como eliminaremos una fila y una columna, utilizaremos una tabla de doble entrada en lugar de la notación matricial con el objeto de no perder la información de a cuáles ramas corresponden los distintos costos (el costo de cada rama (i, j) según $\|c_{ij}(u_1)\|$ es el valor correspondiente a la fila i y la columna j).

En nuestro ejemplo,

$$\|c_{ij}(u_1)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 17 & 26 \\ fila\ 3 & 29 & 1 & \infty & 19 & 0 & 5 \\ fila\ 5 & 3 & 21 & 56 & 7 & \infty & 28 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & \infty \end{pmatrix}.$$

En este caso, el costo de la rama (5, 7) según esta matriz es 28, es decir, $c_{57}(u_1) = 28$. Ahora generamos la matriz $\|c'_{ij}(u_1)\|$ que se obtiene restando a cada coeficiente de la fila i de $\|c_{ij}(u_1)\|$ el máximo número no negativo k_i posible y luego restando a cada coeficiente de la columna j el máximo número no negativo r_j posible, de manera que cada coeficiente de la nueva matriz resulte ser no negativo y definimos $c(u_1) = c(s) + \sum k_i + \sum r_j$. Luego $c(u_1) \geq c(s)$. Notemos que $\|c'_{ij}(u_1)\|$ tiene un coeficiente nulo en cada fila y cada columna.

En nuestro caso resulta que $k_5 = 3$, $k_i = 0$ para $i \neq 5$ y $r_j = 0$ para todo j, de donde

$$\|c'_{ij}(u_1)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 17 & 26 \\ fila\ 3 & 29 & 1 & \infty & 19 & 0 & 5 \\ fila\ 5 & 0 & 18 & 53 & 4 & \infty & 25 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & \infty \end{pmatrix}$$

y $c(u_1) = c(s) + 3 = 96 + 3 = 99$.

Si C es un circuito que contiene a la rama (i_1, j_1) (es decir, una solución factible de u_1) entonces

$$c(C) = c'(C, s) + c(s) = \sum_{(i,j)\in C} c'_{ij}(s) + c(s) = \sum_{\substack{(i,j)\neq(i_1,j_1)\\(i.j)\in C}} c'_{ij}(s) + c(s)$$

ya que (i_1, j_1) fue elegido verificando $c'_{i_1 j_1}(s) = 0$. Sean

$$c(C, u_1) = \sum_{\substack{(i,j)\neq(i_1,j_1)\\(i,j)\in C}} c_{ij}(u_1) \text{ y } c'(C, u_1) = \sum_{\substack{(i,j)\neq(i_1,j_1)\\(i,j)\in C}} c'_{ij}(u_1).$$

Notemos que $c(C, u_1)$ y $c'(C, u_1)$ están bien definidos pues si C es un circuito que contiene a la rama (i_1, j_1) entonces para todo $(i, j) \in C$, $(i, j) \neq (i_1, j_1)$ vale que $i \neq i_1$, que $j \neq j_1$ y que $(i, j) \neq (j_1, i_1)$. Como $\|c_{ij}(u_1)\|$ fue construida eliminando la fila i_1 y la columna j_1 de $\|c'_{ij}(s)\|$ y reemplazando $c'_{j_1 i_1}(s)$ por ∞ entonces para todo $(i, j) \in C$, $(i, j) \neq (i_1, j_1)$ existen los coeficientes $c_{ij}(u_1)$ y $c'_{ij}(u_1)$ y vale

$$\sum_{\substack{(i,j)\neq(i_1,j_1)\\(i,j)\in C}} c'_{ij}(s) = \sum_{\substack{(i,j)\neq(i_1,j_1)\\(i,j)\in C}} c_{ij}(u_1) = c(C, u_1).$$

Luego, $c(C) = c(C, u_1) + c(s)$.

Con el mismo razonamiento utilizado para ver que $c'(C, s) = c(C) - 96$ puede verse que $c'(C, u_1) = c(C, u_1) - 3$ (esto se debe a que $\|c'_{ij}(u_1)\|$ fue construida a partir de $\|c_{ij}(u_1)\|$ con un procedimiento análogo al utilizado para construir $\|c'_{ij}(s)\|$ a partir de $\|c_{ij}(s)\| = \|c_{ij}\|$).

Luego $c(C) = c(C, u_1) + c(s) = c'(C, u_1) + 3 + c(s) = c'(C, u_1) + 3 + 96 = c'(C, u_1) + 99 = c'(C, u_1) + c(u_1)$.

Por lo tanto, $c(C) = c'(C, u_1) + c(u_1)$. Además, como $A_{u_1} = \{(i_1, j_1)\}$ entonces $c'(C, u_1) = \sum_{\substack{(i,j)\in C\\(i,j)\notin A_{u_1}}} c'_{ij}(u_1)$.

Ahora consideremos el problema v_1. Como $\mid A_{v_1} \mid = 0 < 6 = n - 1$ pues $A_{v_1} = \emptyset$, para calcular $c(v_1)$ y luego generar sus hijos le asociamos al problema v_1 la matriz de

costos $\|c_{ij}(v_1)\|$ que se obtiene reemplazando en $\|c'_{ij}(s)\|$ el coeficiente $c'_{i_1j_1}(s)$ por ∞. Esto garantiza que la rama (i_1, j_1) no pertenecerá al circuito (recordemos que v_1 es el problema de hallar un circuito de mínimo costo que no contenga esa rama).

En nuestro ejemplo,

$$\|c_{ij}(v_1)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 6 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 17 & 12 & 26 \\ fila\ 3 & 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ fila\ 4 & 32 & 83 & 66 & \infty & 49 & \infty & 80 \\ fila\ 5 & 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & \infty & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}.$$

Ahora generamos la matriz $\|c'_{ij}(v_1)\|$ que se obtiene restando a cada coeficiente de la fila i de $\|c_{ij}(v_1)\|$ el máximo número no negativo k_i y luego restando a cada coeficiente de la columna j el máximo número no negativo r_j de manera que cada coeficiente de la nueva matriz resulte ser no negativo, y definimos $c(v_1) = c(s) + \sum k_i + \sum r_j$. Luego, $c(v_1) \geq c(s)$. Notemos que $\|c'_{ij}(v_1)\|$ tiene un coeficiente nulo en cada fila y cada columna.

En nuestro caso resulta que $k_4 = 32$, $k_i = 0$ para $i \neq 4$ y $r_j = 0$ para todo j, de donde

$$\|c'_{ij}(v_1)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 6 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 17 & 12 & 26 \\ fila\ 3 & 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ fila\ 4 & 0 & 51 & 34 & \infty & 17 & \infty & 48 \\ fila\ 5 & 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & \infty & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}$$

y $c(v_1) = c(s) + 32 = 96 + 32 = 128$.

Si C es un circuito que no contiene a la rama (i_1, j_1) (es decir, una solución factible de v_1) entonces para todo $(i, j) \in C$ vale que $(i, j) \neq (i_1, j_1)$ de donde $c'_{ij}(s) = c_{ij}(v_1)$ para todo $(i, j) \in C$ (ya que $\|c_{ij}(v_1)\|$ se obtuvo reemplazando en $\|c'_{ij}(s)\|$ el coeficiente $c'_{i_1j_1}(s)$ por ∞). Sean

$$c(C, v_1) = \sum_{(i,j)\in C} c_{ij}(v_1) \text{ y } c'(C, v_1) = \sum_{(i,j)\in C} c'_{ij}(v_1).$$

Ahora se tiene que $c'(C, v_1) = c(C, v_1) - 32$ y como $c'_{ij}(s) = c_{ij}(v_1)$ para todo $(i, j) \in C$ entonces $c(C) = c'(C, s) + c(s) = \sum_{(i,j)\in C} c'_{ij}(s) + c(s) = \sum_{(i,j)\in C} c_{ij}(v_1) + c(s) = c(C, v_1) +$

$c(s) = c^{'}(C, v_1) + 32 + c(s) = c^{'}(C, v_1) + 32 + 96 = c^{'}(C, v_1) + 128 = c^{'}(C, v_1) + c(v_1)$. Luego, $c(C) = c^{'}(C, v_1) + c(v_1)$ y como $A_{v_1} = \emptyset$, entonces $c^{'}(C, v_1) = \sum_{\substack{(i,j)\in C \\ (i,j)\notin A_{v_1}}} c^{'}_{ij}(v_1)$.

Ahora generamos los hijos de u_1 a partir de la matriz $\|c^{'}_{ij}(u_1)\|$. Elegimos una rama (i_2, j_2) de G tal que el coeficiente correspondiente a la fila i_2 y a la columna j_2 de $\|c^{'}_{ij}(u_1)\|$ sea nulo, por ejemplo $(i_2, j_2) = (3, 5)$, y generamos dos hijos u_2 y v_2 de u_1: el primero será el subproblema que resulta de agregar al problema u_1 la condición de que (i_2, j_2) pertenezca al circuito y el segundo el que resulta de agregar a u_1 la condición de que (i_2, j_2) no pertenezca al circuito.

Finalmente, generamos los hijos de v_1 a partir de la matriz $\|c^{'}_{ij}(v_1)\|$. Elegimos una rama (i_3, j_3) de G tal que el coeficiente correspondiente a la fila i_3 y a la columna j_3 de $\|c^{'}_{ij}(v_1)\|$ sea nulo, por ejemplo, $(i_3, j_3) = (2, 1)$ y generamos dos hijos u_3 y v_3 de v_1: el primero será el subproblema que resulta de agregar al problema v_1 la condición de que (i_3, j_3) pertenezca al circuito y el segundo el que resulta de agregar a v_1 la condición de que (i_3, j_3) no pertenezca al circuito.

Paso 4. Calculamos el costo y generamos los hijos de cada nodo u que sea una hoja del subárbol que tenemos construido hasta ahora y que satisfaga que $\mid A_u \mid < n-1$. El subárbol que tenemos construido hasta ahora consiste de la raíz s, sus dos hijos u_1 y v_1, los hijos u_2 y v_2 de u_1 y los hijos u_3 y v_3 de v_1. Las hojas son, por lo tanto, u_2, v_2, u_3 y v_3. Comencemos por los hijos de u_1, que son u_2 y v_2. Como $\mid A_{u_2} \mid = 2 < 6 = n-1$ pues $A_{u_2} = \{(i_1, j_1), (i_2, j_2)\}$, para calcular $c(u_2)$ y luego generar sus hijos le asociamos a u_2 la matriz de costos $\|c_{ij}(u_2)\|$ que se obtiene reemplazando en $\|c^{'}_{ij}(u_1)\|$ el coeficiente $c^{'}_{j_2 i_2}(u_1)$ por ∞ y eliminando la fila i_2 y la columna j_2; análogamente, como $\mid A_{v_2} \mid = 1 < 6 = n-1$ pues $A_{v_2} = \{(i_1, j_1)\}$, para calcular $c(v_2)$ y luego generar sus hijos le asociamos a v_2 la matriz de costos $\|c_{ij}(v_2)\|$ que se obtiene reemplazando en $\|c^{'}_{ij}(u_1)\|$ el coeficiente $c^{'}_{i_2 j_2}(u_1)$ por ∞.

En nuestro caso

$$\|c_{ij}(u_2)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 26 \\ fila\ 5 & 0 & 18 & \infty & 4 & 25 \\ fila\ 6 & 0 & 85 & 8 & \infty & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & \infty \end{pmatrix}$$

y

$$\|c_{ij}(v_2)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 17 & 26 \\ fila\ 3 & 29 & 1 & \infty & 19 & \infty & 5 \\ fila\ 5 & 0 & 18 & 53 & 4 & \infty & 25 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & \infty \end{pmatrix}$$

Restando primero a cada coeficiente de la fila i de $\|c_{ij}(u_2)\|$ el máximo número no negativo k_i y kuego a cada coeficiente de la columna j el máximo número no negativo r_j, de manera que cada coeficiente de la nueva matriz sea no negativo, obtenemos la matriz $\|c'_{ij}(u_2)\|$ que tiene un coeficiente nulo en cada fila y cada columna y como antes definimos $c(u_2) = c(u_1) + \sum k_i + \sum r_j$. Luego $c(u_2) \geq c(u_1)$. En nuestro ejemplo $k_i = 0 = r_j$ para todo i, j de donde $\|c'_{ij}(u_2)\| = \|c_{ij}(u_2)\|$ y por lo tanto se tiene que $c(u_2) = c(u_1) + 0 = c(u_1) = 99$.

Es sencillo ver que si C es un circuito que contiene a las ramas (i_1, j_1) e (i_2, j_2) (es decir, una solución factible de u_2), definiendo

$$c'(C, u_2) = \sum_{\substack{(i,j)\neq(i_1,j_1)\neq(i_2,j_2)\\(i,j)\in C}} c'_{ij}(u_2)$$

resulta que $c(C) = c'(C, u_2) + c(u_2)$ y $c'(C, u_2) = \sum_{\substack{(i,j)\in C\\(i,j)\notin A_{u_2}}} c'_{ij}(u_2)$.

De manera análoga, restando primero a cada coeficiente de la fila i de $\|c_{ij}(v_2)\|$ el máximo número no negativo k_i y luego a cada coeficiente de la columna j el máximo número no negativo r_j, de manera que cada coeficiente de la nueva matriz sea no negativo, obtenemos la matriz $\|c'_{ij}(v_2)\|$ que tiene un coeficiente nulo en cada fila y cada columna y definimos $c(v_2) = c(u_1) + \sum k_i + \sum r_j$. Luego $c(v_2) \geq c(u_1)$.

En nuestro caso $k_3 = 1$, $k_i = 0$ para $i \neq 3$, $r_5 = 17$ y $r_j = 0$ para $j \neq 5$, de donde

$$\|c'_{ij}(v_2)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 13 & 50 \\ fila\ 2 & 0 & \infty & 66 & 37 & 0 & 26 \\ fila\ 3 & 28 & 0 & \infty & 18 & \infty & 4 \\ fila\ 5 & 0 & 18 & 53 & 4 & \infty & 25 \\ fila\ 6 & 0 & 85 & 8 & \infty & 72 & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 41 & \infty \end{pmatrix}$$

y $c(v_2) = c(u_1) + 18 = 99 + 18 = 117$.

Si C es un circuito que contiene a la rama (i_1, j_1) y no contiene a (i_2, j_2) (es decir, una solución factible de v_2), definiendo

$$c'(C, v_2) = \sum_{\substack{(i,j)\neq(i_1,j_1)\\(i,j)\in C}} c'_{ij}(v_2)$$

resulta que $c(C) = c'(C, v_2) + c(v_2)$ y $c'(C, v_2) = \sum_{\substack{(i,j)\notin A_{v_2}\\(i,j)\in C}} c'_{ij}(v_2)$.

Ahora consideremos los hijos de v_1, que son u_3 y v_3. Como $\mid A_{u_3} \mid = 1 < 6 = n - 1$ pues $A_{u_3} = \{(i_3, j_3)\}$, para calcular $c(u_3)$ y luego generar sus hijos le asociamos a u_3 la matriz de costos $\|c_{ij}(u_3)\|$ que se obtiene reemplazando en $\|c'_{ij}(v_1)\|$ el coeficiente $c'_{j_3 i_3}(v_1)$ por ∞ y eliminando la fila i_3 y la columna j_3, y como $\mid A_{v_3} \mid = 0 < 6 = n - 1$

pues $A_{v_3} = \emptyset$, para calcular $c(v_3)$ y luego generar sus hijos le asociamos a v_3 la matriz de costos $\|c_{ij}(v_3)\|$ que se obtiene reemplazando en $\|c'_{ij}(v_1)\|$ el coeficiente $c'_{i_3 j_3}(v_1)$ por ∞.

En nuestro ejemplo,

$$\|c_{ij}(u_3)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 6 & col\ 7 \\ fila\ 1 & \infty & 83 & 9 & 30 & 6 & 50 \\ fila\ 3 & 1 & \infty & 19 & 0 & 12 & 5 \\ fila\ 4 & 51 & 34 & \infty & 17 & \infty & 48 \\ fila\ 5 & 21 & 56 & 7 & \infty & 0 & 28 \\ fila\ 6 & 85 & 8 & \infty & 89 & \infty & 0 \\ fila\ 7 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}$$

y

$$\|c_{ij}(v_3)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 6 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ fila\ 2 & \infty & \infty & 66 & 37 & 17 & 12 & 26 \\ fila\ 3 & 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ fila\ 4 & 0 & 51 & 34 & \infty & 17 & \infty & 48 \\ fila\ 5 & 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & \infty & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}$$

Restando primero a cada coeficiente de la fila i de $\|c_{ij}(u_3)\|$ el máximo número no negativo k_i y luego a cada coeficiente de la columna j el máximo número no negativo r_j, de manera que cada coeficiente de la nueva matriz sea no negativo, obtenemos la matriz $\|c'_{ij}(u_3)\|$ que tiene un coeficiente nulo en cada fila y cada columna y definimos $c(u_3) = c(v_1) + \sum k_i + \sum r_j$. Luego $c(u_3) \geq c(v_1)$.

En nuestro caso, $k_1 = 6$, $k_4 = 17$, $k_i = 0$ para $i \neq 1, 4$, y $r_j = 0$ para todo j de donde

$$\|c'_{ij}(u_3)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 6 & col\ 7 \\ fila\ 1 & \infty & 77 & 3 & 24 & 0 & 44 \\ fila\ 3 & 1 & \infty & 19 & 0 & 12 & 5 \\ fila\ 4 & 34 & 17 & \infty & 0 & \infty & 31 \\ fila\ 5 & 21 & 56 & 7 & \infty & 0 & 28 \\ fila\ 6 & 85 & 8 & \infty & 89 & \infty & 0 \\ fila\ 7 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}$$

y $c(u_3) = c(v_1) + 23 = 128 + 23 = 151$.

Si C es un circuito que no contiene a la rama (i_1, j_1) y sí contiene a (i_3, j_3) (es decir, una solución factible de u_3), definiendo

$$c'(C, u_3) = \sum_{\substack{(i,j) \in C \\ (i,j) \neq (i_3, j_3)}} c'_{ij}(u_3)$$

se tiene que $c(C) = c'(C, u_3) + c(u_3)$ y $c'(C, u_3) = \sum_{\substack{(i,j) \in C \\ (i,j) \notin A_{u_3}}} c'_{ij}(u_3)$.

Análogamente, restando primero a cada coeficiente de la fila i de $\|c_{ij}(v_3)\|$ el máximo número no negativo k_i y luego a cada coeficiente de la columna j el máximo número no negativo r_j, de manera que cada coeficiente de la nueva matriz sea no negativo, obtenemos la matriz $\|c'_{ij}(v_3)\|$ que tiene un coeficiente nulo en cada fila y cada columna y definimos $c(v_3) = c(v_1) + \sum k_i + \sum r_j$. Luego $c(v_3) \geq c(v_1)$.

En nuestro caso $k_2 = 12$, $k_i = 0$ para $i \neq 2$, y $r_j = 0$ para todo j, de donde

$$\|c'_{ij}(v_3)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 & col\ 5 & col\ 6 & col\ 7 \\ fila\ 1 & \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ fila\ 2 & \infty & \infty & 54 & 25 & 5 & 0 & 14 \\ fila\ 3 & 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ fila\ 4 & 0 & 51 & 34 & \infty & 17 & \infty & 48 \\ fila\ 5 & 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ fila\ 6 & 0 & 85 & 8 & \infty & 89 & \infty & 0 \\ fila\ 7 & 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{pmatrix}$$

y $c(v_3) = c(v_1) + 12 = 128 + 12 = 140$.

Además, si C es un circuito que no contiene a las ramas (i_1, j_1) y (i_3, j_3) definiendo

$$c'(C, v_3) = \sum_{(i,j)\in C} c'_{ij}(v_3)$$

resulta que $c(C) = c'(C, v_3) + c(v_3)$ y $c'(C, v_3) = \sum_{\substack{(i,j)\in C \\ (i,j)\notin A_{v_3}}} c'_{ij}(v_3)$.

Ahora deberíamos generar los hijos de u_2, v_2, u_3 y v_3 e ir al paso 5, cosa que no haremos porque damos por entendido cómo continuar el proceso.

Resumiendo: hasta ahora, en cada paso calculamos el costo y generamos los hijos de cada vértice u que sea una hoja del subárbol que se tiene hasta ese momento y que satisfaga que $\mid A_u \mid < n-1$, asociando a u una matriz $\|c'_{ij}(u)\|$ que tiene un cero en cada fila y en cada columna y satisface: Si $A_u = \{e \in E / \text{“}e \in C''\text{es una restricción de } u\}$, para cualquier solución factible C de u se tiene que $c(C) = c'(C, u) + c(u)$ donde $c'(C, u) = \sum_{\substack{(i,j)\in C \\ (i,j)\notin A_u}} c'_{ij}(u)$. Además, el costo $c(u)$ de u que calculamos satisface que $c(u) \leq c(v)$ para cada hijo v de u.

Finalmente, veamos ahora cómo procedemos cuando tenemos una hoja u del subárbol generado hasta el momento que satisface $\mid A_u \mid = n - 1$. En este caso, no generamos ningún hijo de u y para calcular $c(u)$ primero determinamos si u es o no es factible. Si es factible entonces calculamos $c(u)$ con el mismo procedimiento de antes. En cambio, si no es factible, ponemos $c(u) = \infty$.

De esta manera obtenemos un árbol binario con raíz donde cada nodo u tiene asignado un costo $c(u)$ en forma tal que vale $c(u) \leq c(v)$ si (u, v) es una rama del árbol. Las hojas de este árbol son los vértices h tales que $\mid A_h \mid = n - 1$. Además,

para cada hoja h del árbol que sea un problema factible tenemos definida una matriz $\|c'_{ij}(h)\|$ que tiene un cero en cada fila y cada columna y satisface: Si $A_h = \{e \in E/$"$e \in C''$es una restricción de $h\}$, para cualquier solución factible C de h se tiene que $c(C) = c'(C,h) + c(h)$ donde $c'(C,h) = \sum_{\substack{(i,j)\in C \\ (i,j)\notin A_h}} c'_{ij}(h)$.

Notemos que determinar si una hoja h del árbol es factible es fácil porque $\mid A_h \mid = n-1$ y entonces hay $n-1$ ramas que necesariamente deben pertenecer a cualquier solución factible de h. Por la forma en que fuimos procediendo (cada vez que agregamos una restricción del tipo "$(i,j) \in C$" eliminamos la fila i y la columna j), no puede haber en A_h dos ramas con la misma punta ni dos ramas con la misma cola. Luego, los conjuntos $\{i/(i,j) \in A_h\}$ y $\{j/(i,j) \in A_h\}$ tienen $n-1$ elementos cada uno. Por lo tanto, existe un único i_0 y un único j_0 tales que $i_0 \notin \{i/(i,j) \in A_h\}$ y $j_0 \notin \{j/(i,j) \in A_h\}$. Notemos que si C es una solución factible de h entonces tiene n ramas, verifica que para cada i entre 1 y n hay una y sólo una rama de C cuya cola sea i y una y sólo una rama de C cuya punta sea i y las $n-1$ ramas de A_h deben ser ramas de C. Esto muestra que el único camino P que podría ser una solución factible de h es aquél cuyas ramas son las $n-1$ ramas pertenecientes a A_h y la rama (i_0, j_0) (ya que si agregamos una rama (i,j) con $i \neq i_0$ o $j \neq j_0$ a las ramas pertenecientes a A_h ese camino no tendría ninguna rama cuya punta es i_0 o ninguna rama cuya cola es j_0). Por otra parte, como P no contiene dos ramas con la misma punta ni dos ramas con la misma cola, entonces es un circuito (en cuyo caso es la única solución factible de h) o bien consiste de dos o más subcircuitos (en cuyo caso h no es factible), cosa que puede chequearse fácilmente.

Veamos ahora que la solución al problema del viajante es la hoja de mínimo costo. Si h es una hoja factible entonces $\mid A_h \mid = n-1$. Como cada vez que agregamos una restricción del tipo "(i, j) ∈ C" eliminamos una fila y una columna y para obtener h hemos agregado $n-1$ de estas restricciones, entonces $\|c'_{ij}(h)\|$ es una matriz de 1×1 que tiene un cero en cada fila y cada columna. Luego debe ser $\|c'_{ij}(h)\| = (0)$ de donde $c'(C,h) = 0$. Por lo tanto, si C es una solución factible de h resulta que $c(C) = c'(C,h) + c(h) = c(h)$. Como las soluciones factibles del problema del viajante son las soluciones factibles de cada una de las hojas del árbol y como el costo de dicha solución factible coincide con el costo de su correspondiente hoja entonces nuestro problema se traduce en encontrar una hoja de mínimo costo. Además, como vimos antes, si h es la hoja de mínimo costo entonces tiene una única solución factible que puede hallarse fácilmente y que, por lo dicho, resulta ser la solución al problema del viajante. Para hallar una hoja de mínimo costo, utilizamos el método de branch and bound y, como en el caso de P.L.E., en lugar de generar el árbol y luego aplicar el algoritmo, en cada paso calculamos sólo los vértices y los costos que necesitamos.

Resolvamos el problema del viajante para el grafo completo de $n = 4$ vértices que tiene como matriz de costos a la matriz

$$\|c_{ij}\| = \begin{pmatrix} \infty & 4 & 99 & 23 \\ 8 & \infty & 86 & 55 \\ 6 & 24 & \infty & 20 \\ 12 & 87 & 22 & \infty \end{pmatrix}$$

Iniciamos el algoritmo poniendo $c = \infty$. Primero calculamos $\|c'_{ij}(s)\|$ y $c(s)$ y obtenemos

$$\|c'_{ij}(s)\| = \begin{pmatrix} \infty & 0 & 85 & 5 \\ 0 & \infty & 68 & 33 \\ 0 & 18 & \infty & 0 \\ 0 & 75 & 0 & \infty \end{pmatrix}$$

y $c(s) = 54 < \infty = c$.

Ahora generamos uno de los hijos de s, que denotaremos por p_1, agregando la restricción $(4, 1) \in C$. Se tiene entonces que

$$\|c_{ij}(p_1)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & 0 & 85 & \infty \\ fila\ 2 & \infty & 68 & 33 \\ fila\ 3 & 18 & \infty & 0 \end{pmatrix}$$

de donde

$$\|c'_{ij}(p_1)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & 0 & 50 & \infty \\ fila\ 2 & \infty & 0 & 0 \\ fila\ 3 & 18 & \infty & 0 \end{pmatrix}$$

y $c(p_1) = 122 < \infty = c$.

Ahora generamos uno de los hijos de p_1, al que denotaremos por p_2, obtenido agregándole a p_1 la restricción $(2, 3) \in C$. Se tiene entonces que

$$\|c_{ij}(p_2)\| = \begin{pmatrix} & col\ 2 & col\ 4 \\ fila\ 1 & 0 & \infty \\ fila\ 3 & \infty & 0 \end{pmatrix}$$

de donde $\|c'_{ij}(p_2)\| = \|c_{ij}(p_2)\|$ y $c(p_2) = 122 < \infty = c$.

Ahora generamos uno de los hijos de p_2, al que denotaremos por p_3, obtenido agregándole a p_2 la restricción $(1, 2) \in C$. Como $\mid A_{p_3} \mid = 3 = n - 1$ entonces para calcular su costo primero debemos ver si es factible. Como $A_{p_3} = \{(4, 1), (2, 3), (1, 2)\}$ entonces $\{i/(i, j) \in A_{p_3}\} = \{4, 2, 1\}$ y $\{j/(i, j) \in A_{p_3}\} = \{1, 3, 2\}$. Por lo tanto, el único i_0 entre 1 y 4 tal que $i_0 \notin \{i/(i, j) \in A_{p_3}\}$ es $i_0 = 3$ y el único j_0 entre 1 y 4 tal que $j_0 \notin \{j/(i, j) \in A_{p_3}\}$ es $j_0 = 4$. Luego, el único camino que podría ser una solución factible de p_3 es áquel cuyas ramas son (4, 1), (2, 3), (1, 2), y (3, 4). Como este camino es un circuito entonces p_3 es factible y su única solución es el circuito $4 \to 1 \to 2 \to 3 \to 4$. Calculemos $c(p_3)$. Como

$$\|c_{ij}(p_3)\| = \begin{pmatrix} & col\ 4 \\ fila\ 3 & 0 \end{pmatrix}$$

entonces $\|c'_{ij}(p_3)\| = \|c_{ij}(p_3)\|$ y $c(p_3) = 122 < \infty = c$. Como además p_3 es una hoja entonces actualizamos h y c poniendo $h = p_3$ y $c = 122$.

Ahora hacemos un backtracking y volvemos a p_2 para generar su otro hijo, que denotaremos por p_4, obtenido agregando a p_2 la restricción $(1,2) \notin C$. Como

$$\|c_{ij}(p_4)\| = \begin{pmatrix} & col\ 2 & col\ 4 \\ fila\ 1 & \infty & \infty \\ fila\ 3 & \infty & 0 \end{pmatrix}$$

entonces

$$\|c'_{ij}(p_4)\| = \begin{pmatrix} & col\ 2 & col\ 4 \\ fila\ 1 & 0 & 0 \\ fila\ 3 & \infty & 0 \end{pmatrix}$$

y $c(p_4) = \infty \geq 122 = c$, de modo que hacemos otro backtracking y volvemos a p_1 para generar su otro hijo, que denotaremos por p_5, obtenido agregando a p_1 la restricción $(2,3) \notin C$. Se tiene que

$$\|c_{ij}(p_5)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & 0 & 50 & \infty \\ fila\ 2 & \infty & \infty & 0 \\ fila\ 3 & 18 & \infty & 0 \end{pmatrix}$$

de donde

$$\|c'_{ij}(p_5)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & 0 & 0 & \infty \\ fila\ 2 & \infty & \infty & 0 \\ fila\ 3 & 18 & \infty & 0 \end{pmatrix}$$

y $c(p_5) = 172$. Como $c(p_5) = 172 \geq 122 = c$ entonces podamos toda la descendencia de p_5 y hacemos otro backtracking para volver a s y generar su otro hijo, que denotaremos por p_6, obtenido agregando a s la restricción $(4,1) \notin C$. Se tiene que

$$\|c_{ij}(p_6)\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 0 & 85 & 5 \\ fila\ 2 & 0 & \infty & 68 & 33 \\ fila\ 3 & 0 & 18 & \infty & 0 \\ fila\ 4 & \infty & 75 & 0 & \infty \end{pmatrix}$$

de donde $\|c'_{ij}(p_6)\| = \|c_{ij}(p_6)\|$ y $c(p_6) = 54 < 122 = c$.

Ahora generamos uno de los hijos de p_6, al que denotaremos por p_7, obtenido agregándole a p_6 la restricción $(2,1) \in C$. Se tiene entonces que

$$\|c_{ij}(p_7)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 85 & 5 \\ fila\ 3 & 18 & \infty & 0 \\ fila\ 4 & 75 & 0 & \infty \end{pmatrix}$$

luego,

$$\|c'_{ij}(p_7)\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 80 & 0 \\ fila\ 3 & 0 & \infty & 0 \\ fila\ 4 & 57 & 0 & \infty \end{pmatrix}$$

de donde $c(p_7) = 77 < 122 = c$.

Ahora generamos uno de los hijos de p_7, al que denotaremos por p_8, obtenido agregándole a p_7 la restricción $(3,2) \in C$. Se tiene entonces que

$$\|c_{ij}(p_8)\| = \begin{pmatrix} & col\ 3 & col\ 4 \\ fila\ 1 & 80 & 0 \\ fila\ 4 & 0 & \infty \end{pmatrix}$$

de donde $\|c'_{ij}(p_8)\| = \|c_{ij}(p_8)\|$ y $c(p_8) = 77 < 122 = c$.

Ahora generamos uno de los hijos de p_8, al que denotaremos por p_9, obtenido agregándole a p_8 la restricción $(4,3) \in C$. Como $\mid A_{p_9} \mid = 3 = n-1$, para calcular su costo debemos ver si es factible. Es fácil verificar que p_9 es factible y que su única solución factible es el circuito $4 \to 3 \to 2 \to 1 \to 4$. Calculemos su costo. Como

$$\|c_{ij}(p_9)\| = \begin{pmatrix} & col\ 4 \\ fila\ 1 & 0 \end{pmatrix}$$

entonces $\|c'_{ij}(p_9)\| = \|c_{ij}(p_9)\|$ y $c(p_9) = 77 < 122 = c$. Como además p_9 es una hoja actualizamos h y c poniendo $h = p_9$ y $c = 77$.

Ahora hacemos un backtracking y volvemos a p_8 para generar su otro hijo, que denotaremos por p_{10}, obtenido agregando a p_8 la restricción $(4,3) \notin C$. Como

$$\|c_{ij}(p_{10})\| = \begin{pmatrix} & col\ 3 & col\ 4 \\ fila\ 1 & 80 & 0 \\ fila\ 4 & \infty & \infty \end{pmatrix}$$

entonces

$$\|c'_{ij}(p_{10})\| = \begin{pmatrix} & col\ 3 & col\ 4 \\ fila\ 1 & 80 & 0 \\ fila\ 4 & 0 & 0 \end{pmatrix}$$

y $c(p_{10}) = \infty \geq 77 = c$, de modo que podamos toda su descendencia y hacemos otro backtracking volviendo a p_7 para generar su otro hijo, que denotaremos por p_{11}, obtenido agregando a p_7 la restricción $(3,2) \notin C$. Se tiene que

$$\|c_{ij}(p_{11})\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 80 & 0 \\ fila\ 3 & \infty & \infty & 0 \\ fila\ 4 & 57 & 0 & \infty \end{pmatrix}$$

luego,

$$\|c'_{ij}(p_{11})\| = \begin{pmatrix} & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 80 & 0 \\ fila\ 3 & \infty & \infty & 0 \\ fila\ 4 & 0 & 0 & \infty \end{pmatrix}$$

y $c(p_{11}) = 134$. Como $c(p_{11}) = 134 \geq 77 = c$ entonces podamos toda la descendencia de p_{11} y hacemos otro backtracking para volver a p_6 y generar su otro hijo, que denotaremos por p_{12}, obtenido agregando a p_6 la restricción $(2,1) \notin C$. Se tiene que

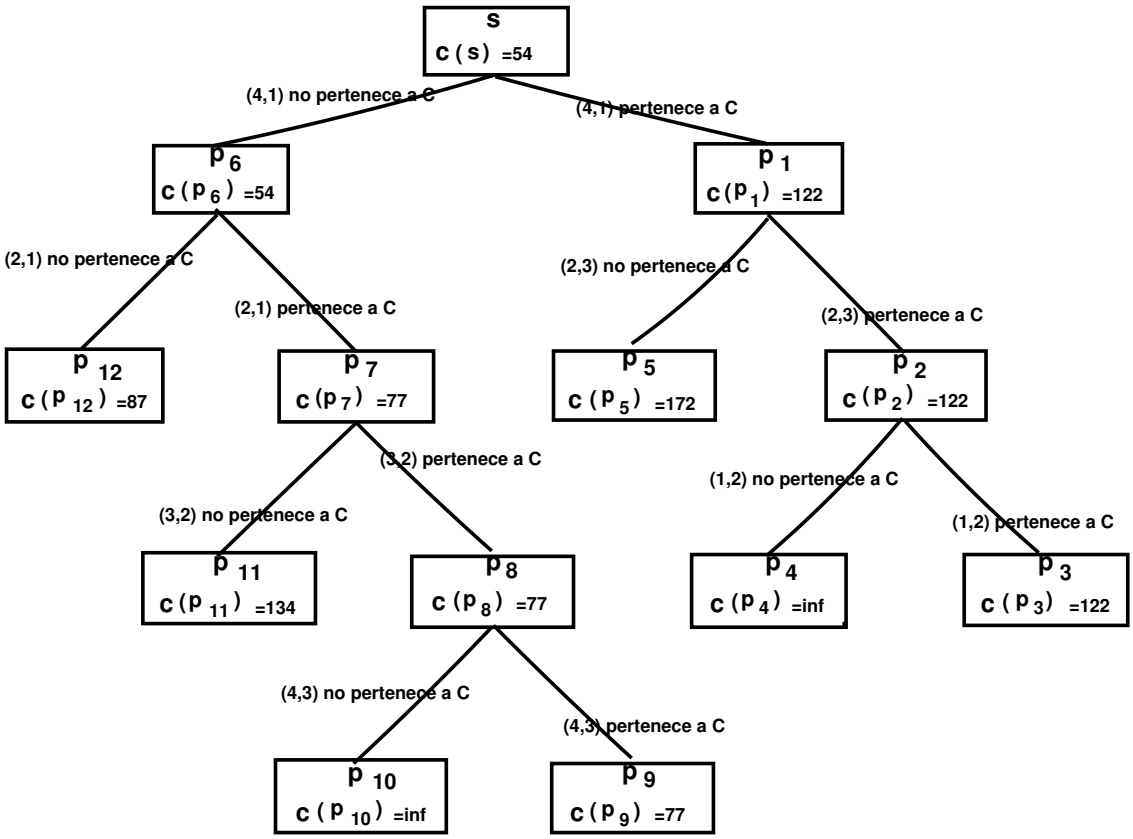
$$\|c_{ij}(p_{12})\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 0 & 85 & 5 \\ fila\ 2 & \infty & \infty & 68 & 33 \\ fila\ 3 & 0 & 18 & \infty & 0 \\ fila\ 4 & \infty & 75 & 0 & \infty \end{pmatrix}$$

de donde

$$\|c'_{ij}(p_{12})\| = \begin{pmatrix} & col\ 1 & col\ 2 & col\ 3 & col\ 4 \\ fila\ 1 & \infty & 0 & 85 & 5 \\ fila\ 2 & \infty & \infty & 35 & 0 \\ fila\ 3 & 0 & 18 & \infty & 0 \\ fila\ 4 & \infty & 75 & 0 & \infty \end{pmatrix}$$

y $c(p_{12}) = 87$. Como $c(p_{12}) = 87 \geq 77 = c$ entonces podamos toda la descendencia de p_{12} y como ya hemos examinado todos los hijos de todos los vértices a los que podemos llegar con backtracking, el algoritmo se detiene.

Luego, la hoja de mínimo costo es $h = p_9$ con costo $c = 77$, lo que significa que el circuito C de mínimo costo es $4 \to 3 \to 2 \to 1 \to 4$ con costo $c(C) = 77$. El subárbol generado por el algoritmo es



Si bien este método es mejor que convertir TSP en P.L.E. y usar branch and bound, aún así, es exponencial y no es recomendable cuando la cantidad de ciudades es grande.

Capítulo 3

Algoritmos para el TSP

Que el problema del viajante sea $\mathcal{NP}$-Hard hace que resulte poco probable encontrar un algoritmo eficiente que garantice hallar un tour óptimo cuando el número de ciudades es grande. Se puede tener un algoritmo cuyo tiempo de ejecución sea rápido o uno que encuentre un tour óptimo, pero no uno que cumpla las dos condiciones simultáneamente. En la práctica se diseñan algoritmos heurísticos eficientes que aunque no garanticen el hallazgo de tours óptimos obtienen tours que se espera que sean "cercanos" al óptimo. Como, en general, es imposible determinar el óptimo verdadero, la eficiencia de estos algoritmos se estudia utilizando distintos criterios que intentan determinar qué tan buena es la performance del algoritmo sobre las instancias de un problema. Cada uno de estos análisis tiene sus ventajas y sus desventajas.

Comenzaremos el capítulo comentando el algoritmo codicioso. Observaremos que se pueden construir instancias en donde el tour hallado a partir del mismo difiere notablemente del óptimo. Luego nos concentraremos en el algoritmo del mínimo spanning tree que tiene una performance mejor. Veremos que el tamaño del tour que resulta de su aplicación es a lo sumo el doble del tamaño del óptimo. Como una extensión de este último algoritmo expondremos el algoritmo de Christofides en el que el tamaño del tour encontrado es menor o igual a tres medios el tamaño del óptimo. Dedicaremos el resto del capítulo al estudio de uno de los mejores algoritmos heurísticos, el algoritmo de Lin y Kernighan, que intenta encontrar una solución "buena" en una cantidad de tiempo aceptable. El mismo ha sido evaluado y comparado empíricamente con distintos problemas tests. Para poder comprobar nosotros mismos su eficacia, decidimos implementarlo en Visual Basic y ponerlo a prueba con tres problemas clásicos de la literatura.

3.1. Algoritmo codicioso

Describiremos una familia de instancias del TSP para la cual la aplicación del algoritmo codicioso arrojará consecuencias indeseables [3]. De aquí en adelante, asumire-

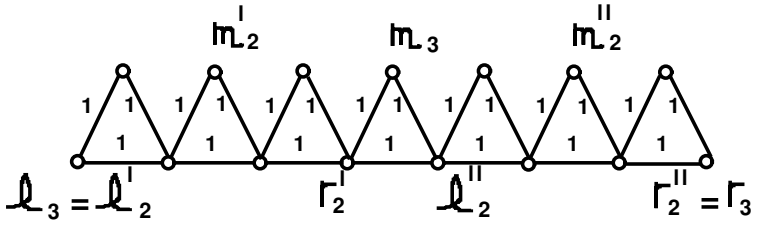
mos que las distancias satisfacen la desigualdad triangular ($d(i,k) + d(k,j) \geq d(i,j)$ $\forall\ 1 \leq i,j,k \leq n$). Es decir, consideramos el $\triangle$TSP. Llamaremos *tour parcial* a un camino que visita cada una de las ciudades a lo sumo una vez. En cambio, por *tour* entenderemos a un camino cerrado que pasa por cada una de las n ciudades exactamente una vez. Recordemos que el objetivo en el TSP es encontrar un tour de mínimo tamaño.

El algoritmo codicioso realiza los siguientes pasos para construir un tour:

a) Se comienza con un tour parcial consistente de una única ciudad a_1 elegida arbitrariamente.
b) Si el tour parcial es $a_1, ..., a_k$, y a_{k+1} es la ciudad más cercana a a_k entre las que no se encuentran en el tour, se agrega a_{k+1}.
c) Se finaliza cuando el tour contiene todas las ciudades.

Construcción de la familia de instancias del TSP

Para $k \geq 1$, consideremos el grafo $G_k = (V_k, E_k)$ que consiste de una cadena de $2^k - 1$ triángulos. Como ejemplo, podemos observar en la siguiente figura el grafo G_3.



El grafo G_k tiene 2^k vértices en su nivel inferior y $2^k - 1$ vértices en su nivel superior. Al vértice que se encuentra en el extremo izquierdo del nivel inferior se lo llama l_k, al que se halla en el extremo derecho del mismo nivel, r_k y al vértice central del superior se lo nota m_k. Una definición equivalente en forma recursiva de G_k ($k \geq 1$) es la siguiente: el grafo G_1 es un triángulo con tres vértices destacados (l_1, m_1 y r_1). Para $k \geq 2$ el grafo G_k se puede formar a partir de dos copias del grafo G_{k-1} (G'_{k-1} y G''_{k-1}) junto con un nuevo vértice m_k. Se agrega una rama entre los vértices r'_{k-1} y l''_{k-1}, otra entre m_k y r'_{k-1} y una última uniendo los vértices m_k y l''_{k-1}. Para finalizar, a los vértices l'_{k-1} y r''_{k-1} se los renombra como l_k y r_k respectivamente. Dados dos vértices i y j cualesquiera; si no se especifica c_{ij} (pues como podemos apreciar se observan sólo ciertas distancias $c_{ij} = 1$ que corresponden a los lados de los triángulos en la figura) entonces el tamaño de dicha rama se lo considera como el del menor camino entre las ciudades i y j a través de ramas cuyos tamaños son conocidos. Esta forma de elegir el tamaño garantiza que la desigualdad triangular se satisfaga.

Lema: Sea $k \geq 1$ y G un grafo no dirigido que contiene a G_k como grafo inducido tal que todas las ramas entre G_k y $G - G_k$ son incidentes en los vértices l_k o r_k de

G_k. Si I_G denota la instancia del TSP que corresponde a G y cuya distancia cumple las condiciones expuestas anteriormente, entonces existe un tour parcial τ_k construido a partir del algoritmo codicioso para la instancia I_G que cumple lo siguiente:

(a) Visita todas las ciudades de G_k.

(b) Empieza en la ciudad l_k y termina en la ciudad m_k.

(c) Tiene tamaño $(k+3)2^{k-1}-2$.

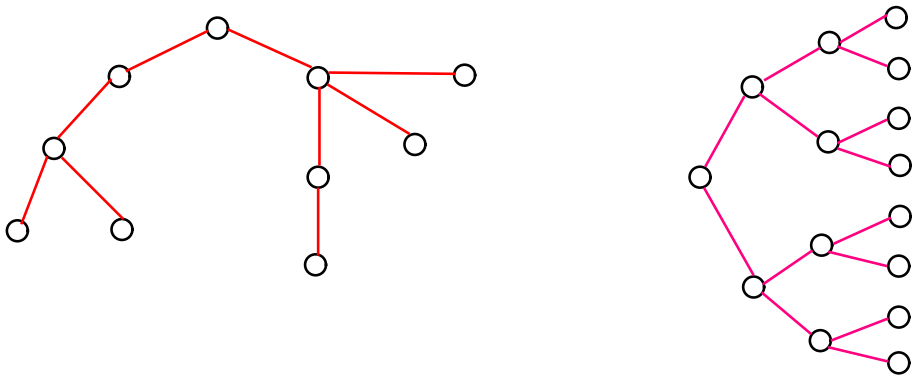
Demostración: Haremos inducción sobre k. Cuando $k=1$, elegimos el camino $l_1 \to r_1 \to m_1$ de tamaño 2 para τ_1. Para $k \geq 2$, usamos la definición recursiva de G_k que lo define en términos de dos copias G'_{k-1} y G''_{k-1} de G_{k-1} junto con el vértice nuevo m_k. Por hipótesis inductiva existe un tour parcial τ'_{k-1} de tamaño $(k+2)2^{k-2}-2$ en G'_{k-1} construido a partir del algoritmo codicioso que empieza en la ciudad l'_{k-1} $(=l_k)$ y termina en la ciudad central m'_{k-1} visitando todas las ciudades de G'_{k-1}. Notemos que la ciudad l''_{k-1} está a distancia $2^{k-2}+1$ de la ciudad m'_{k-1} (pues el camino más corto en G_k sería el que recorre la rama de la derecha a m'_{k-1}, luego pasa por todas las ramas del nivel inferior en forma recta hasta llegar a r'_{k-1}, habiendo pasado por la mitad de los nodos de G'_{k-1} con un costo de 2^{k-2} y por último se agrega la rama que une r'_{k-1} con l''_{k-1} de costo 1). Como m'_{k-1} está a distancia 2^{k-2} de l'_{k-1} y r'_{k-1}, todas las otras ciudades que aún no han sido visitadas se encuentran a una distancia de al menos $2^{k-2}+1$ de m'_{k-1}. Por lo tanto, es factible según el algoritmo codicioso dirigirse a l''_{k-1} después de m'_{k-1}. Aplicando nuevamente la hipótesis inductiva, se atraviesa G''_{k-1} desde l''_{k-1} hasta m''_{k-1} de acuerdo al tour parcial τ''_{k-1} con un tamaño de $(k+2)2^{k-2}-2$. Para finalizar, se visita la ciudad m_k pues es la más cercana entre las que no fueron recorridas todavía. Logramos construir el tour parcial τ_k que va de l_k a m_k en G_k de tamaño $(k+2)2^{k-2}-2+(2^{k-2}+1)+(k+2)2^{k-2}-2+(2^{k-2}+1)=(k+3)2^{k-1}-2$.

Teorema: Para cada $k \geq 1$, existe una instancia del TSP con $n=2^{k+1}$ vértices y un tour óptimo de tamaño 2^{k+1} a la que al aplicarle el algoritmo codicioso se construye un tour de tamaño $(k+4)2^{k-1}$ (los costos cumplen con la condición expuesta al comienzo). En otras palabras, la razón entre los tamaños del tour hallado por el algoritmo y el óptimo resulta $(3+\log n)/4$.

Demostración: Agregamos una ciudad nueva v a G_k, conectamos v a l_k y a r_k. Como el grafo que resulta es hamiltoniano, la instancia del TSP tiene un tour de tamaño $n=2^{k+1}$. El tour parcial τ_k del lema previo junto con las distancias de las ramas que unen m_k con v y v con l_k forman un tour de tamaño $(k+3)2^{k-1}-2+(2^{k-1}+1)+1=(k+4)2^{k-1}$. Podemos concluir que la razón entre los tamaños es $(k+4)2^{k-1}/2^{k+1}=(\log_2 n+3)n/4n=(\log_2 n+3)/4$.

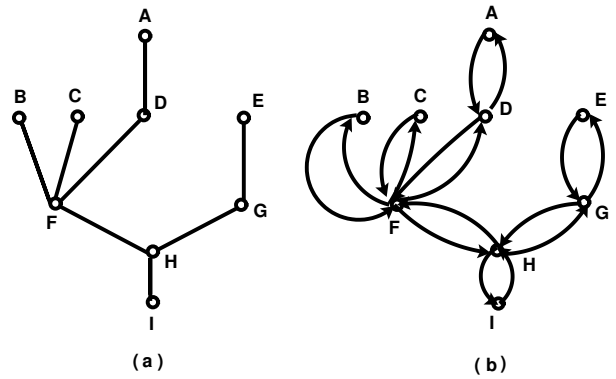
3.2. Algoritmo del mínimo spanning tree

Se puede apreciar una relación entre los tours del problema del viajante y los spanning tree mínimos. Un *spanning tree* para un conjunto de n ciudades es una colección de $n-1$ ramas que unen todas las ciudades sin formar ciclos. Veamos algunos ejemplos:



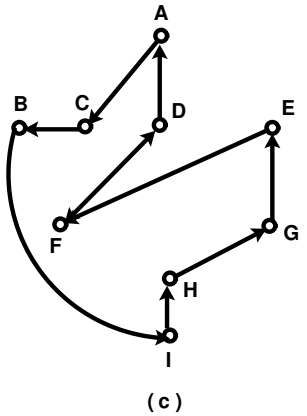
Hay una amplia variedad de algoritmos de tiempo polinomial que construyen un spanning tree de mínima distancia, donde la distancia de un spanning tree es la suma de las distancias de las ramas que lo forman [4]. Cuando el input es dado en la forma de una matriz C que contiene las distancias, el mínimo spanning tree puede ser encontrado en tiempo $O(n^2)$. A diferencia del TSP, el problema del mínimo spanning tree puede ser resuelto en forma eficiente. Esta solución nos provee una cota inferior para el tamaño del tour óptimo, pues observemos que borrando cualquier rama de un tour se obtiene un camino que contiene todas las ciudades y por lo tanto un spanning tree. Veremos como la desigualdad triangular nos permite usar el spanning tree mínimo para obtener una cota superior para el tamaño del tour óptimo para el $\triangle TSP$ [5].

Supongamos que queremos visitar todas las ciudades pero sólo se nos permite usar ramas del mínimo spanning tree. Podríamos comenzar en una hoja (vértices de *deg* 1) del árbol y aplicar la siguiente estrategia: si nos encontramos en un vértice y hay alguna rama de las que inciden en él que todavía no fue recorrida, atravesarla para llegar a un nuevo vértice. Si todas las ramas fueron atravesadas, se tiene que retroceder. Se finaliza cuando se regresa al vértice inicial. Este procedimiento se conoce con el nombre de depth first search, no es difícil verificar que se visitan todos los vértices (ciudades) y que se atraviesan exactamente dos veces cada una de las ramas del mínimo spanning tree. Grafiquemos este método en un ejemplo:



En la figura (a) tenemos un mínimo spanning tree T y en la figura (b) aplicamos el depth first search en T: $I \to H \to G \to E \to G \to H \to F \to D \to A \to D \to F \to C \to F \to B \to F \to H \to I$. Este método nos provee una forma de visitar todas las ciudades de tamaño igual que dos veces la del mínimo spanning tree. El único inconveniente es que algunas ciudades se visitan más de una vez, y es aquí cuando si ponemos la condición de que la distancia cumpla con la desigualdad triangular obtenemos un beneficio. Podemos evitar repetir ciudades introduciendo "shortcuts" que no incrementan la distancia total. Se comienza al igual que antes, en una hoja de un mínimo spanning tree. Sin embargo, cada vez que el depth first search retrocede a una ciudad que ya fue visitada, la saltea y va directamente a la próxima que no haya sido visitada aún.

En el recorrido anterior eliminamos la repetición de ciudades de la siguiente manera: reemplazamos $E \to G \to H \to F$ por $E \to F$. Análogamente, cambiamos $A \to D \to F \to C$ por $A \to C$; $C \to F \to B$ por $C \to B$ y $B \to F \to H \to I$ por $B \to I$.



En la figura (c) se observa el tour hallado luego de aplicar el depth first search con "shortcuts": $I \to H \to G \to E \to F \to D \to A \to C \to B \to I$.

Construimos un nuevo tour que visita cada ciudad exactamente una vez. Su distancia no es más grande que la del camino obtenido con el depth first search; así, la distancia del tour es a lo sumo dos veces la del tour original. Tenemos un algoritmo

de tiempo polinomial con una performance razonable. Este algoritmo al que llamaremos del mínimo spanning tree corre en tiempo $O(n^2)$ como se puede ver del siguiente resumen:

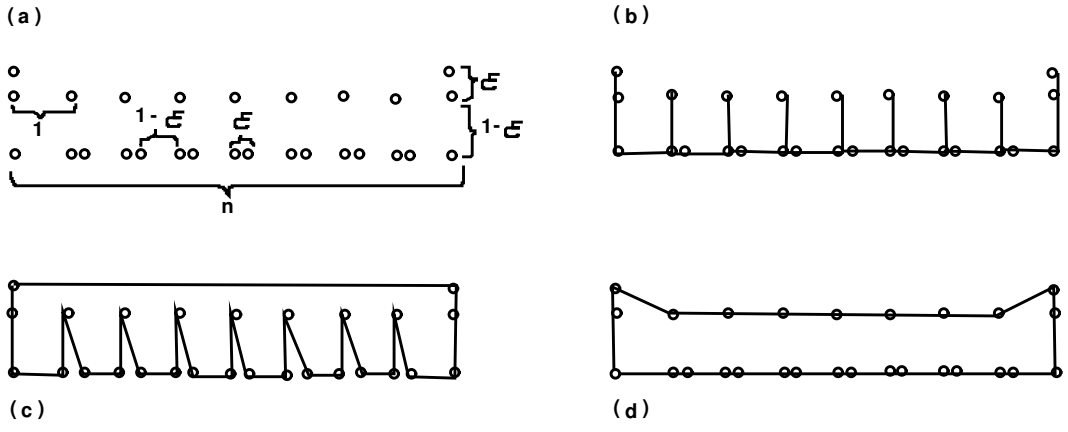
1. Encontrar un mínimo spanning tree para el conjunto dado de ciudades ($O(n^2)$).
2. Aplicar depht first search ($O(n)$).
3. Introducir "shortcuts" en el depth first search para obtener un tour ($O(n)$).

Los pasos 2 y 3 se pueden hacer juntos (aunque sigue siendo $O(n)$ y no cambia la complejidad).

Teorema: Para toda instancia I del TSP que cumple la desigualdad triangular, si MST(I) es el tamaño del tour que resulta de aplicar a I el algoritmo del mínimo spanning tree, entonces MST(I) $\leq$ 2 Opt(I).

Demostración: Como comentamos anteriormente, el tour encontrado al aplicar el depth first search ($tour_{DFS}$) atraviesa exactamente dos veces cada una de las ramas del mínimo spanning tree (T), por lo tanto, $c(tour_{DFS}) = 2c(T)$. El inconveniente de quedarnos con este tour es que algunas ciudades se visitan más de una vez, por eso se construye un nuevo tour introduciendo "shortcuts"que visita cada ciudad exactamente una vez. El costo del nuevo tour ($tour_{shortcuts}$) no es más grande que el del tour construido a partir del depth first search pues la distancia cumple la desigualdad triangular, es decir, $c(tour_{shortcuts}) \leq c(tour_{DFS})$. Por último, basta observar que borrando cualquier rama del tour óptimo se obtiene un camino que contiene todas las ciudades y por lo tanto un spanning tree, con lo que $c(T) \leq c(tour_{optimo})$. Finalmente, concluimos que $c(tour_{shortcuts}) \leq c(tour_{DFS}) = 2c(T) \leq 2c(tour_{optimo})$.

Es la mejor cota que se puede garantizar como se observa en el siguiente ejemplo. Tenemos que MST(I) $\rightarrow_{\epsilon \rightarrow 0} 4n$ y Opt(I) $\rightarrow_{\epsilon \rightarrow 0} 2n + 2$, por lo que si λ es tal que $\forall n$ MST(I) $\leq \lambda$ Opt(I) debe ser $\lambda \geq 2$. Las distancias no sólo cumplen la desigualdad triangular sino que entre los puntos del plano se verifica la distancia euclideana.



(a) Instancia del TSP euclideano.

(b) El mínimo spanning tree, tamaño $= n + (n+1)(1-\epsilon) + 2\epsilon = 2n - \epsilon(n-1) + 1$.

(c) Tour doble del spanning tree con shortcuts, tamaño $\sim 4n$.

(d) Tour óptimo, tamaño $\sim 2n + 2$.

3.3. Algoritmo de Christofides

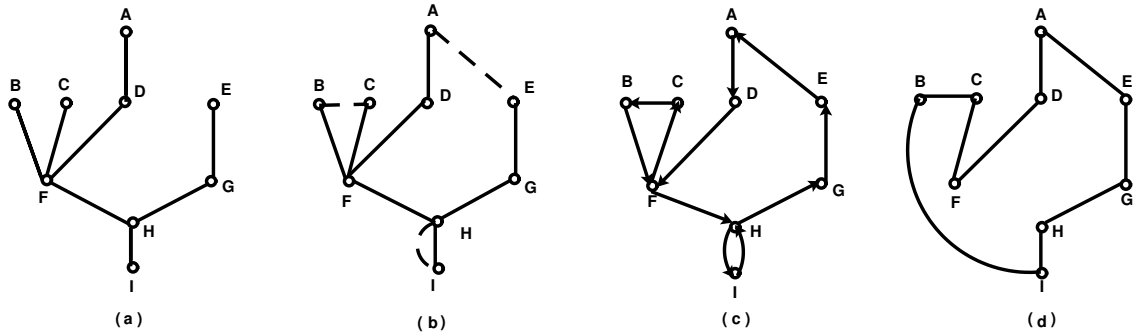
Las ideas del algoritmo del mínimo spanning tree fueron extendidas por Christofides para lograr una mejor ejecución para el problema del viajante. En ella se combinan las nociones de grafo euleriano, tour euleriano y matching.

Un grafo *euleriano* es un grafo conexo en donde cada vértice tiene grado par. Es fácil ver que un grafo euleriano es aquel que contiene un tour euleriano, es decir, un ciclo que pasa por cada rama exactamente una vez. Dado un grafo euleriano, se puede encontrar un tour euleriano en tiempo $O(n)$. Supongamos que tenemos un grafo euleriano con las ciudades de una instancia del TSP como sus vértices. Podemos usar el tour euleriano de este grafo para obtener un tour del problema del viajante utilizando la técnica de "shortcut". Empezamos con un mínimo spanning tree, duplicamos sus ramas para obtener un grafo euleriano, encontramos un tour euleriano para este grafo y luego lo convertimos en un tour del TSP usando "shortcuts". Por la desigualdad triangular el tour del TSP no es más grande que el tour euleriano.

Esto sugiere que si queremos encontrar mejores tours del TSP, todo lo que necesitamos es una mejor forma de generar grafos eulerianos que conecten las ciudades. El mejor método para producir estos grafos fue realizado por Christofides y utiliza el concepto de *matching de mínimo peso* [5].

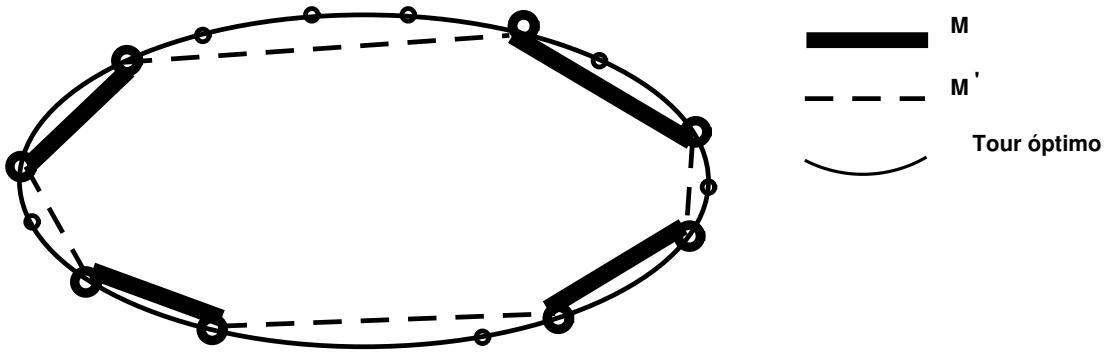
Dado un conjunto que contiene un número par de ciudades, un matching es una colección de ramas M tal que cada ciudad es el extremo de exactamente una rama en M. Un matching de mínimo peso es aquel cuyo peso total de las ramas sea mínimo. Esta clase de matchings pueden ser encontrados en tiempo $O(n^3)$.

Veamos un ejemplo: observamos en la figura (a) un mínimo spanning tree T para una instancia del TSP. Algunos de los vértices en T ya tienen grado par y por lo tanto no deben recibir más ramas si queremos convertir el árbol en un grafo euleriano. Los únicos vértices de los que debemos ocuparnos son de aquellos con grado impar. Notemos que hay una cantidad par de vértices con grado impar (pues la suma de los grados de todos los vértices debe ser par).



Como podemos apreciar en la figura (b) una forma simple de construir un grafo euleriano que incluya T es agregando un matching para los vértices de grado impar. Esto incrementará el grado de cada vértice de grado impar en uno y conservará el grado par de los vértices que lo tuvieran. Es sencillo ver que si a T le agregamos un matching de mínimo peso para los vértices de grado impar, obtenemos un grafo euleriano que tiene mínimo peso entre todos aquellos que contienen a T. En la figura (c) tenemos un tour euleriano de (b): $I \to H \to G \to E \to A \to D \to F \to C \to B \to F \to H \to I$. En la figura (d) observamos un tour obtenido por el algoritmo de Christofides, usando "shortcuts": $I \to H \to G \to E \to A \to D \to F \to C \to B \to I$.

Consideremos el siguiente dibujo que nos muestra un tour óptimo del TSP, con las ciudades que corresponden a los vértices de grado impar en T destacadas. Podemos convertir al tour mínimo en un tour que pasa únicamente por los vértices distinguidos. Por la desigualdad triangular, el tamaño de este nuevo tour es menor o igual al tamaño del tour original. Este tour construido determina dos matching M y M', el menor de estos matchings no puede exceder al tamaño de la mitad del tour y por lo tanto, debe tener peso menor o igual que Opt(I)/2. Entonces el tamaño de un matching de mínimo peso para los vértices de T de grado impar será a lo sumo Opt(I)/2. Concluimos que el tamaño del grafo euleriano construido es a lo sumo (3/2)Opt(I).



Tenemos el siguiente algoritmo llamado algoritmo de Christofides:

1. Construimos un mínimo spanning tree T sobre el conjunto de todas las ciudades en I ($O(n^2)$).

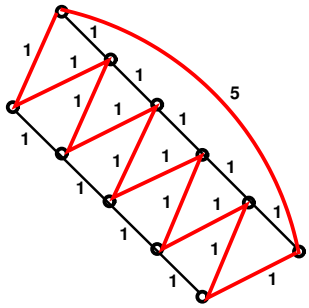
2. Construimos un matching mínimo M^* para el conjunto de todos los vértices de grado impar de T ($O(n^3)$).

3. Encontramos un tour euleriano en el grafo euleriano que es la unión de T y M^*, y luego lo convertimos en un tour usando shortcuts ($O(n)$).

El tiempo que tarda en correr el algoritmo es dominado por el tiempo que se tarda en encontrar el matching en el paso 2 y es $O(n^3)$.

Teorema: Para cualquier instancia I del TSP que cumple la desigualdad triangular, si C(I) es el tamaño de un tour construido cuando el algoritmo de Christofides se aplica a I, C(I) $\leq$ (3/2)Opt(I).

Ésta es la mejor cota garantizada por el algoritmo de Christofides aún en el caso euclideano, como se sigue del siguiente ejemplo donde podemos ver una instancia I del TSP euclideano en donde C(I) = (3/2)[Opt(I)-1], en este caso Opt(I) = 11 y C(I) = 15.



3.4. Algoritmo de Lin y Kernighan

El algoritmo de Lin y Kernighan [6] pertenece a una clase de algoritmos heurísticos muy usados: los de búsqueda local. Se comienza con un tour inicial; si no hay tours vecinos de menor costo, nos quedamos con el que tenemos hasta el momento porque se trata de un óptimo local, de lo contrario reemplazamos el tour que tenemos en ese momento por un tour vecino de menor costo. Este proceso finaliza en algún momento pues se tiene un número finito de posibles tours. El tiempo que requiere la búsqueda de un vecino mejor depende del número de iteraciones que se necesitan (o sea, el número de veces que el tour puede ser mejorado antes de encontrar un óptimo local). Lin y Kernighan definen los vecinos de un tour como aquellos tours que pueden ser obtenidos a partir de él intercambiando un número finito de ramas del tour por ramas que no pertenecen al tour. El tiempo de ejecución de este algoritmo es aproximadamente n^2 (n representa la cantidad total de ciudades).

En lo que resta del capítulo nos dedicaremos por completo a este procedimiento heurístico que genera soluciones óptimas en ciertos casos y en otros, soluciones cercanas al óptimo, para el problema del viajante simétrico.

3.4.1. Estructura general del algoritmo

En esta sección comentaremos la estructura general en la que se basa el algoritmo de Lin y Kernighan; más adelante lo describiremos en detalle.

Muchos problemas de optimización combinatoria pueden ser formulados en forma abstracta como "encontrar en un conjunto S un subconjunto T que satisface cierto criterio C y minimiza una función objetivo f". Por ejemplo, en el TSP, tenemos que encontrar en el conjunto de todas las ramas de un grafo completo, un subconjunto que forme un tour y tenga distancia mínima. Un enfoque heurístico para los problemas de optimización combinatoria es la mejora iterativa de un conjunto de soluciones factibles seleccionada aleatoriamente:

1. Generar al azar una solución factible, es decir, un conjunto T que satisface C.
2. Tratar de encontrar una solución factible mejor $T^{'}$ por alguna transformación de T.
3. Si se encuentra una solución mejor, es decir, tal que $f(T^{'}) < f(T)$, entonces se reemplaza T por $T^{'}$ y se repite el paso 2.
4. Si no se puede encontrar una solución mejor, T es una solución óptima local.

Repetir desde el paso 1 hasta que se termine el tiempo fijado de antemano o la respuesta sea satisfactoria.

El corazón de este procedimiento iterativo es el paso 2, en donde se intenta mejorar la solución dada. Una transformación que ha sido aplicada a una gran variedad de problemas es el cambio de un número fijo k de elementos de T con k elementos de $S - T$, de forma que la solución resultante $T^{'}$ sea factible y mejor. Este mecanismo se repite hasta que no sea posible mejorar T a través de estos cambios, en cuyo caso estaremos en presencia de una solución óptima local. El principal problema es encontrar los elementos correctos a reemplazar.

Esta estrategia de intercambios fue aplicada al TSP por Croes ($k = 2$) [7] y por Lin ($k = 3$) [8] con un éxito considerable. Tener que especificar el valor de k de antemano es un serio problema. El tiempo computacional se incrementa rápidamente cuando el k aumenta y además es difícil saber qué k es más adecuado usar para que exista un balance entre el tiempo utilizado y la calidad de la solución. El método heurístico que estamos abordando se basa en una generalización de la ya mencionada transformación de intercambios.

Supongamos que T es una solución factible pero no óptima, entonces podemos decir que T no es óptima porque hay k elementos $x_1, ..., x_k$ en T que son "incorrectos"; para convertir a T en un óptimo deberían ser reemplazados por k elementos $y_1, ..., y_k$ de $S - T$. El problema es simplemente identificar los x's y los y's. Trataremos de encontrar $k, x_1, ..., x_k, y_1, ..., y_k$ de la mejor forma que podamos, buscando elemento por elemento.

Empezaremos intentando identificar x_1 e y_1, luego x_2 e y_2 y así continuaremos hasta haber encontrado todos los pares de elementos.

Más formalmente, el esquema del algoritmo es el siguiente:

1. Generar una solución inicial T.

2. *a)* $i = 1$.

   b) Seleccionar x_i en $T - \{x_1, ..., x_{i-1}\}$ e y_i en $S - T - \{y_1, ..., y_{i-1}\}$ de forma tal que se maximice la ganancia cuando $x_1, ..., x_i$ es cambiado por $y_1, ..., y_i$.

   c) Si no se puede obtener más ganancia, de acuerdo a una apropiada regla de finalización, ir al paso 3; de otra forma, $i = i + 1$ y volver a 2 b).

3. Si la mejora se encontró para $i = k$, cambiar $x_1, ..., x_k$ con $y_1, ..., y_k$ para obtener un nuevo T e ir al paso 2; si no se encuentra mejora alguna, ir al paso 4.

4. Repetir desde el paso 1 si se desea.

Se necesitan varias cosas para hacer este trabajo más claro:
* Una regla de selección que nos diga con cuáles pares nos quedamos en el paso 2 b).
* Una función que represente la ganancia total para un conjunto de intercambios propuesto. Supongamos que g_i es la ganancia asociada con el cambio de x_i con y_i dado que $x_1, y_1, ..., x_{i-1}, y_{i-1}$ ya han sido elegidos. Sería más beneficioso si la ganancia de los cambios de $x_1, ..., x_i$ con $y_1, ..., y_i$ es $g_1 + ... + g_i$, o sea, si la ganancia es aditiva. Cumpliéndose esta aditividad no es necesario que el proceso de selección se detenga inmediatamente cuando alguno de los g_i es negativo, en realidad necesitamos que se detenga cuando $\sum_{i=1}^{k} g_i \leq 0$ para todo k de interés. Esto ayuda a evadir mínimos locales.
* Debemos asegurar que la solución que se obtiene una vez que se hicieron los cambios resulta factible, o sea, que satisface el criterio C. Se requiere que cada cambio propuesto nos deje en un estado factible.
* Una regla que haga detener el proceso y nos asegure que no se puede obtener mejora alguna intentando otro cambio.
* Los conjuntos $x_1, ..., x_k$ e $y_1, ..., y_k$ deben ser disjuntos. Una vez que se realizó el cambio de un elemento resulta imposible que vuelva a reincorporarse en esa iteración.

Después de examinar cierta secuencia $x_1, ..., x_m$ e $y_1, ..., y_m$ de intercambios con sus correspondientes ganancias $g_1, ..., g_m$, el valor de k que define el conjunto a intercambiar es aquel para el cual $g_1 + ... + g_k$ es máximo y factible. Si la suma de las ganancias es positiva, los conjuntos son intercambiados obteniéndose una nueva y mejor solución T, y el proceso se itera desde este nuevo punto de partida. Si, en cambio, $g_1 + ... + g_k$ es cero o negativo, esto indica que no se puede hacer mejora alguna y que la solución que tenemos es nuestro óptimo local.

Para aplicar este método al problema del viajante, sea S el conjunto de todas las ramas entre las n ciudades ($\mid S \mid = n(n-1)/2$), y T un subconjunto de S que forma un tour ($\mid T \mid = n$) (criterio de factibilidad C). Queremos encontrar un tour de mínima distancia (función objetivo f).

Consideremos un tour arbitrario T de tamaño $f(T)$ y un tour $T^{'}$ con tamaño $f(T^{'}) < f(T)$. Supongamos que T y $T^{'}$ difieren (como conjuntos de n ramas) en k ramas. El algoritmo intenta transformar T en $T^{'}$ identificando sucesivamente los k pares de ramas a ser cambiadas entre T y $S - T$. Tratamos de encontrar dos conjuntos de ramas $X = \{x_1, ..., x_k\}$ e $Y = \{y_1, ..., y_k\}$ de forma que si las ramas en X son extraídas y reemplazadas por las ramas en Y, el resultado es un tour de menor costo. Notemos que numeramos las ramas: los extremos de x_1 son t_1 y t_2, en general tenemos que $x_i = (t_{2i-1}, t_{2i})$, $y_i = (t_{2i}, t_{2i+1})$ y $x_{i+1} = (t_{2i+1}, t_{2i+2})$ para $i \geq 1$, como observamos x_i e y_i comparten un extremo y lo mismo sucede con y_i y x_{i+1}. Generalmente es posible realizar esta numeración y por lo tanto convertir T en $T^{'}$. Si asumimos que la numeración puede realizarse, la secuencia de intercambios que queremos encontrar es $x_1, y_1; x_2, y_2; ...;$ etc. Se desea lograr una secuencia que reduzca T a $T^{'}$ con $f(T^{'}) < f(T)$ e iterar el proceso en $T^{'}$ hasta que no se puedan hacer más reducciones.

Supongamos que los tamaños de x_i y y_i son $\mid x_i \mid$ y $\mid y_i \mid$ respectivamente, y definamos $g_i = \mid x_i \mid - \mid y_i \mid$. Esta es la ganancia de intercambiar x_i con y_i. Aunque alguno de los g_i puede ser negativo, si $f(T^{'}) < f(T)$, claramente tenemos que $\sum_{i=1}^{k} g_i = f(T) - f(T^{'}) > 0$. Parte de este procedimiento se basa en lo siguiente: si una secuencia de números tiene una suma positiva, hay una permutación cíclica de estos números de forma que cada suma parcial es positiva. En efecto, si consideramos a k como el mayor índice para el cual $g_1 + g_2 + ... + g_{k-1}$ es mínimo:
* si $k \leq j \leq n$, $g_k + ... + g_j = (g_1 + ... + g_j) - (g_1 + ... + g_{k-1}) > 0$
* si $1 \leq j \leq k$, $g_k + ... + g_n + g_1 + ... + g_j \geq g_k + ... + g_n + g_1 + ... + g_{k-1} > 0$

Entonces, como se buscan secuencias de g_i's que tengan suma positiva, sólo necesitamos considerar secuencias de ganancias cuyas sumas parciales son siempre positivas. Esto reduce enormemente el número de secuencias que necesitamos examinar, es el corazón de la regla de finalización.

3.4.2. Algoritmo

1. Generar al azar un tour inicial T.

2. $G^* = 0$ (mejoras realizadas). Elegir cualquier nodo t_1 y sea x_1 una de las ramas de T adyacentes a t_1.

   $i = 1$.

3. Del otro extremo t_2 de x_1, elegir y_1 con extremo t_3 con $g_1 > 0$. Si no existe y_1, ir al paso 6(d).

4. $i = i + 1$. Elegir x_i (que une t_{2i-1} con t_{2i}) e y_i como sigue:

a) x_i es elegido de forma que, si t_{2i} se une con t_1, la configuración que resulta es un tour (x_i no debe ser ninguno de los x_j con $j < i$).

Antes de construir y_i debemos chequear si uniendo t_{2i} con t_1 tenemos un valor mejor que el anterior. Sea y_i^* la rama que conecta t_{2i} con t_1, $g_i^* =\mid x_i \mid - \mid y_i^* \mid$ y $G_i = \sum_{j=1}^{i} g_j$. Si $G_{i-1} + g_i^* > G^*$, cambiamos $G^* = G_{i-1} + g_i^*$ y $k = i$.

b) y_i es alguna rama con extremo en t_{2i} sujeta a c), d) y e):

- c) Los x's y los y's deben ser disjuntos.
- d) $G_i = \sum_{j=1}^{i} g_j > 0$.
- e) El y_i elegido debe permitir extraer un x_{i+1}.

Además, $\mid y_i \mid$ debe ser lo más pequeño posible y no debe ser ninguno de los y_j agregados anteriormente. Si no existe y_i, ir al paso 5.

5. La construcción de x_i e y_i en los pasos 2 a 4 se termina cuando no hay ramas x_i e y_i que satisfacen 4 c)-e) o cuando $G_i \leq G^*$. Si $G^* > 0$ tomar el tour $T^{'}$ con $f(T^{'}) = f(T) - G^*$ y repetir todo el proceso desde el paso 2, usando $T^{'}$ como la solución inicial.

6. Si $G^* = 0$ se aplica un backtracking limitado como sigue:

   a) Repetir pasos 4 y 5 eligiendo y_2 tratando de incrementar su tamaño y que satisfaga que $g_1 + g_2 > 0$.

   b) Si toda opción de y_2 es examinada sin beneficio alguno, volver a 4 (a) y elegir la otra posibilidad para x_2.

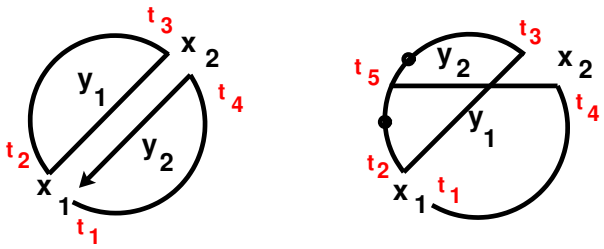
   c) Si esto falla se trata de encontrar y_1 que aumente el tamaño (se vuelve al paso 3).

   d) Si c) no da una mejora, se considera la otra rama x_1 (paso 2).

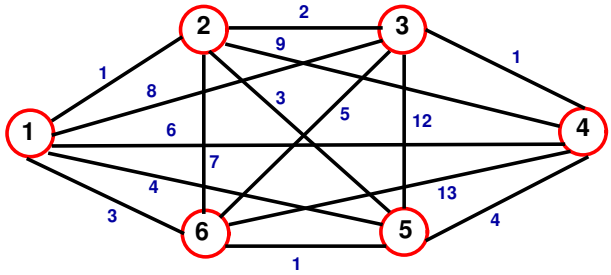
   e) Si d) falla, un nuevo t_1 es seleccionado (paso 2).

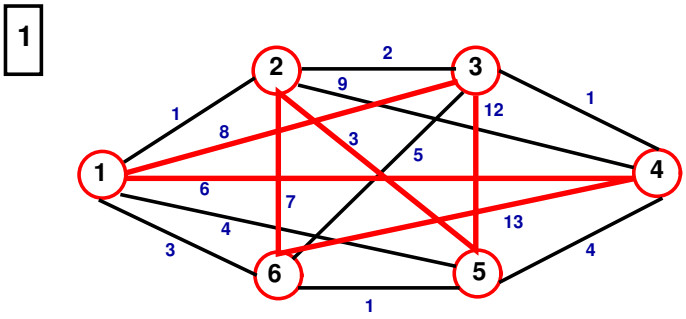
7. El procedimiento termina cuando los n valores de t_1 han sido examinados sin encontrar mejora alguna.

Notemos que en el backtracking al permitir cambiar x_2 estamos violando en forma temporal el criterio de factibilidad. Se permite porque aunque agrega cierta complejidad aumenta la efectividad. Como podemos observar en la siguiente figura al unir t_4 con t_1 no se forma un tour pero si elegimos y_2 de forma que t_5 se encuentre entre t_2 y t_3 entonces el tour se cerrará en el próximo paso.

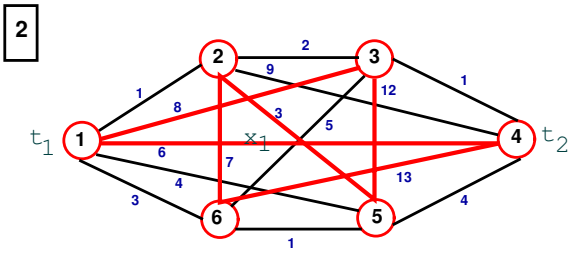


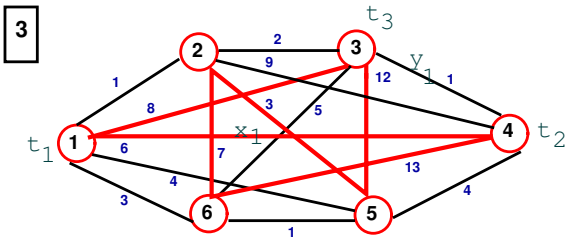
Veamos un ejemplo:



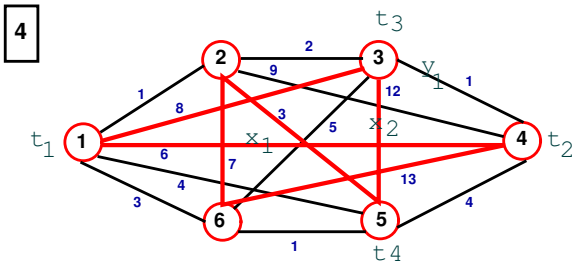


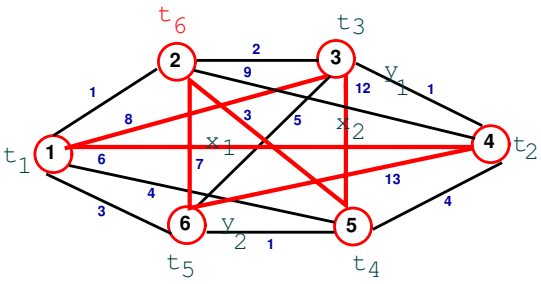
Tour inicial T, f(T)=49



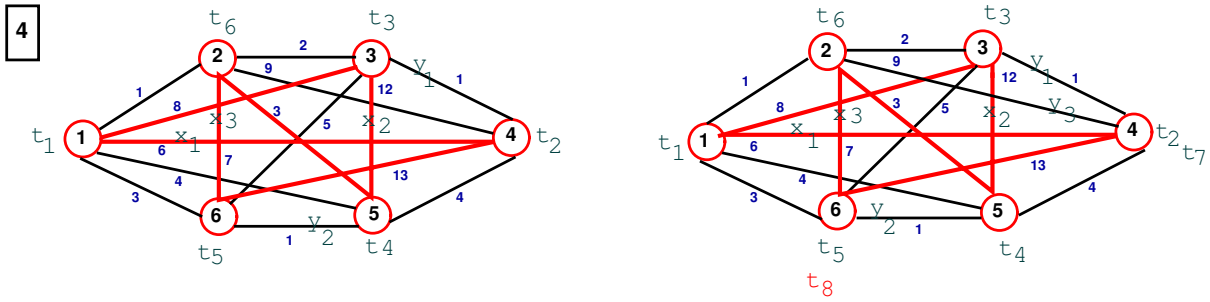


2) $G^* = 0$, $i = 1$
3) $g_1 = 6 - 1 = 5 > 0$



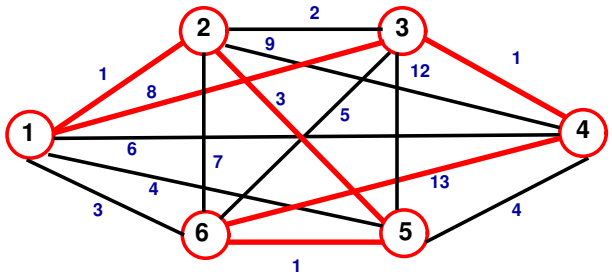


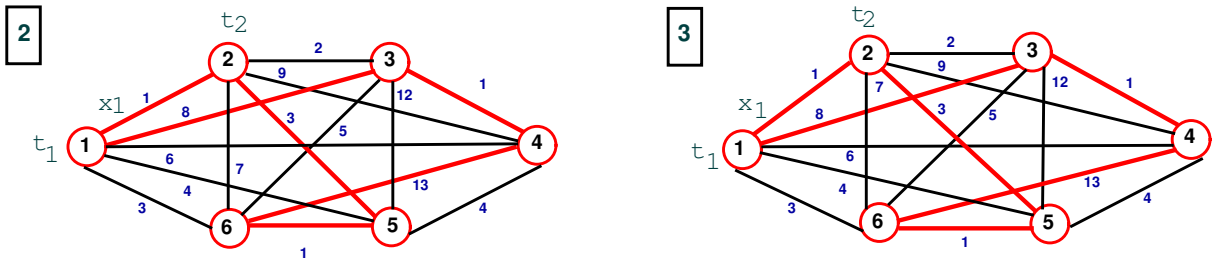
4) $i = 2$
a) $g_2^* = 12 - 4 = 8$ como $G_1 + g_2^* = 5 + 8 = 13 > G^* = 0$ entonces $G^* = 13$ y $k = 2$
b) $G_2 = 5 + 11 = 16$, al ser $G_2 > G^*$ seguimos



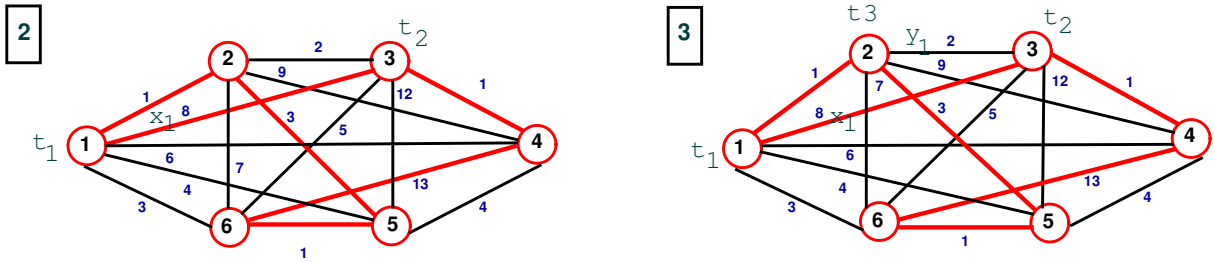
4) $i = 3$
a) $g_3^* = 7 - 1 = 6$ como $G_2 + g_3^* = 16 + 6 = 22 > G^* = 13$ entonces $G^* = 22$ y $k = 3$
b) $G_3 = 16 + (-2) = 14$

5) Al ser $G_3 = 14 \leq G^* = 22$ y $G^* > 0$ cambiamos el tour T por el T' siendo $f(T') = f(T) - G^* = 49 - 22 = 27$ y volvemos al paso 2)

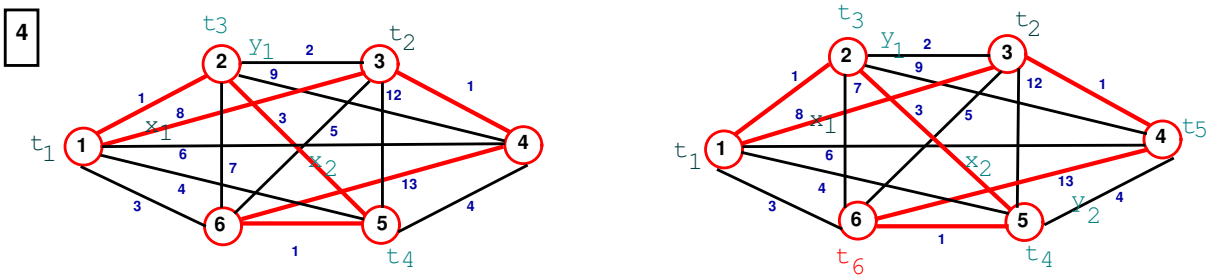




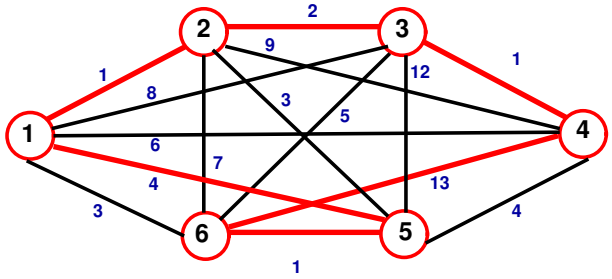
2) $G^* = 0$, $i = 1$
3) Como no puedo encontrar y_1 / $g_1 > 0$ vamos al paso 6) d)

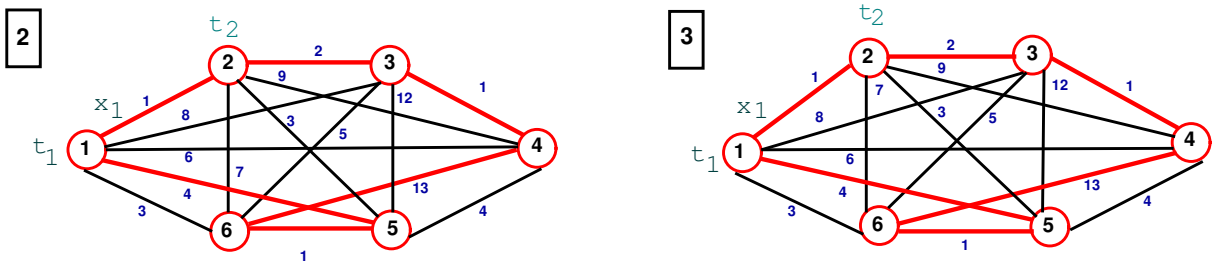


3) $g_1 = 8 - 2 = 6 > 0$

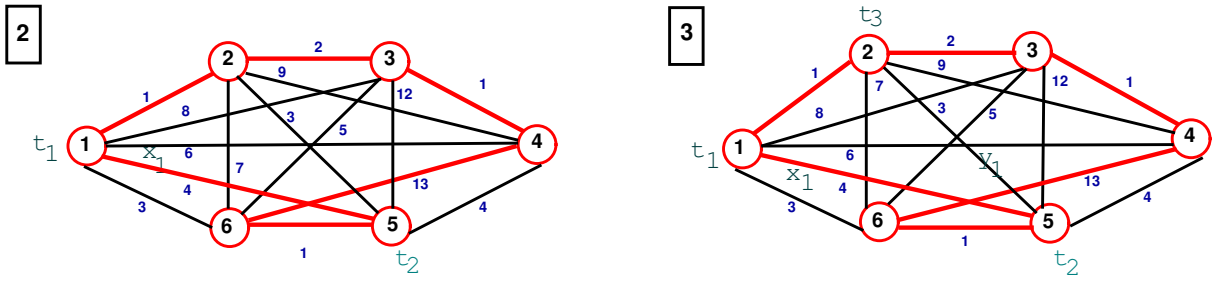


4) $i = 2$
a) $g_2^* = 3 - 4 = -1$ como $G_1 + g_2^* = 6 + (-1) = 5 > G^* = 0$ entonces $G^* = 5$ y $k = 2$
b) $G_2 = 6 + (-1) = 5$, al ser $G_2 \leq G^*$ y $G^* > 0$ cambiamos el tour T por el T' siendo $f(T') = f(T) - G^* = 27 - 5 = 22$ y volvemos al paso 2)

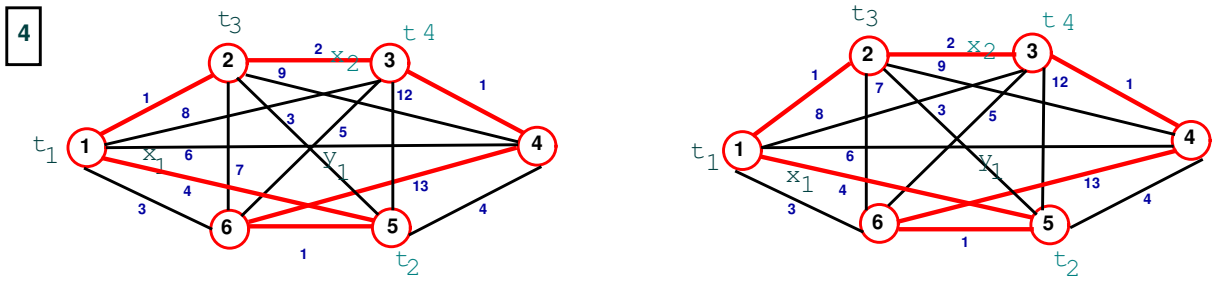




2) $G^* = 0$, $i = 1$
3) Como no puedo encontrar y_1 / $g_1 > 0$ vamos al paso 6) d)

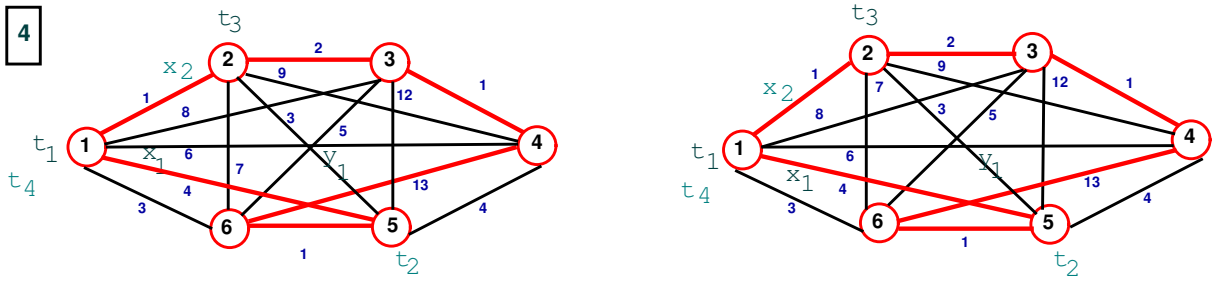


3) $g_1 = 4 - 3 = 1 > 0$



4) $i = 2$
a) $g_2^* = 2 - 8 = -6$ y como $G_1^* + g_2^* = 1 + (-6) = -5 < G^* = 0$ no cambio G^*
b) no existe y_2

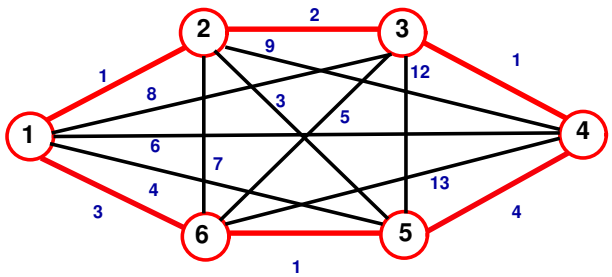
6) Al ser $G^* = 0$ aplicamos backtracking, alternamos x_2



4)
a) Notemos que no se cumple la condición de factibilidad
b) no existe y_2

6) Resulta imposible alternar y_1, como ya consideramos todas las opciones de x_1 sólo resta elegir otro t_1

Y continuando de esta forma con la aplicación del algoritmo se llega a que el tour óptimo es



3.4.3. Resultados computacionales

Este algoritmo fue implementado en Visual Basic y probado con tres problemas clásicos:
1) 42 ciudades de Danzig, Fulkerson y Johnson
2) 105 ciudades de Lin y Kernighan
3) 318 ciudades de Lin y Kernighan

En la codificación podemos observar una estructura principal y quince subrutinas.

Subrutinas

1) Function RamasQueQuedan(n) Entradas: n, tour y vector Salida: ind

(n representa el número de ramas $x's$ que serán eliminadas hasta el momento)

(creamos un vector llamado ind donde se guardará uno de los extremos de las ramas que no se eliminarán del tour)

```
For i = 1 To CantidadDeCiudades
   ind(i) = 0
Next i

w = 1
For t = 1 To CantidadDeCiudades
   For i = 1 To n
```

(se compara cada rama del tour con las que han sido elegidas como $x's$)

```
        If (tour(t) = vector(2 * i) And t = vector(2 * i - 1)) Or
            (tour(t) = vector(2 * i - 1) And t = vector(2 * i)) Then
```

(si coincide con alguna de las $x's$ no se guarda en ind y se busca otra rama del tour)

```
           Exit For
        Else
```

(si no es igual a ninguna de las $x's$ guardamos uno de los extremos de dicha rama en el vector ind)

```
           If i = n Then
              ind(w) = t
              w = w + 1
           End If
```

```
         End If
      Next i
   Next t
```

End Function

2) Function CargandoRamas(n) Entradas: n, vector, ind y tour Salida: CargandoAristas

(n representa el número de ramas $y's$ que ingresan al tour)

(generamos una matriz llamada CargandoAristas en donde guardaremos las ramas que quedan del tour original, las ramas $y's$ que agregamos y la y^*)

```
For i = 1 To CantidadDeCiudades
   CargandoAristas(1, i) = 0
Next i

For i = 1 To CantidadDeCiudades
   CargandoAristas(2, i) = 0
Next i
```

(leemos los pares de vértices comenzando por la primera coordenada)
(guardamos las ramas $y's$)

```
For i = 1 To n - 1
   If CargandoAristas(1, vector(2 * i)) = 0 Then
      CargandoAristas(1, vector(2 * i)) = vector(2 * i + 1)
   Else
      CargandoAristas(2, vector(2 * i)) = vector(2 * i + 1)
   End If
Next i
```

(guardamos y^*)

```
If CargandoAristas(1, vector(2 * n)) = 0 Then
   CargandoAristas(1, vector(2 * n)) = vector(1)
Else
   CargandoAristas(2, vector(2 * n)) = vector(1)
End If
```

(guardamos las ramas que quedan del tour original)

```
For i = 1 To CantidadDeCiudades - n
   If CargandoAristas(1, ind(i)) = 0 Then
      CargandoAristas(1, ind(i)) = tour(ind(i))
   Else
      CargandoAristas(2, ind(i)) = tour(ind(i))
   End If
Next i
```

(leemos los pares de vértices comenzando por la segunda coordenada)
(archivamos las ramas $y's$)

```
For i = 1 To n - 1
   If CargandoAristas(1, vector(2 * i + 1)) = 0 Then
      CargandoAristas(1, vector(2 * i + 1)) = vector(2 * i)
   Else
      CargandoAristas(2, vector(2 * i + 1)) = vector(2 * i)
   End If
Next i
```

(archivamos y^*)

```
If CargandoAristas(1, vector(1)) = 0 Then
   CargandoAristas(1, vector(1)) = vector(2 * n)
Else
   CargandoAristas(2, vector(1)) = vector(2 * n)
End If
```

(archivamos las ramas que quedan del tour original)

```
For i = 1 To CantidadDeCiudades - n
   If CargandoAristas(1, tour(ind(i))) = 0 Then
      CargandoAristas(1, tour(ind(i))) = ind(i)
   Else
      CargandoAristas(2, tour(ind(i))) = ind(i)
   End If
Next i
```

End Function

3) Function Check() Entradas: CargandoAristas Salida: TourTemporal y Check

```
For i = 1 To CantidadDeCiudades+1
   TourTemporal(i) = 0
Next i

TourTemporal(1) = 1
TourTemporal(2) = CargandoAristas(1, 1)

For t = 3 To CantidadDeCiudades+1
   TourTemporal(t) = 0
```

(nos fijamos si CargandoAristas(1, TourTemporal($t-1$)) es distinto de todos los TourTemporal anteriores para ver si podemos definir al TourTemporal(t) como CargandoAristas(1, TourTemporal($t-1$)))

```
      For i = 1 To t - 1
         If CargandoAristas(1, TourTemporal(t - 1)) <> TourTemporal(i) Then
            If i = t - 1 Then
               TourTemporal(t) = CargandoAristas(1, TourTemporal(t - 1))
               Exit For
            End If
         Else
            Exit For
         End If
      Next i
```

(si CargandoAristas(1, TourTemporal($t-1$)) es igual a algún TourTemporal previo, TourTemporal(t) = 0)

```
      If TourTemporal(t) = 0 Then
```

(analizamos si CargandoAristas(2, TourTemporal($t-1$)) es distinto de todos los TourTemporal previos para definir TourTemporal(t) como CargandoAristas(2, TourTemporal($t-1$)))

```
         For i = 1 To t - 1
            If CargandoAristas(2, TourTemporal(t - 1)) <> TourTemporal(i) Then
               If i = t - 1 Then
                  TourTemporal(t) = CargandoAristas(2, TourTemporal(t - 1))
                  Exit For
               End If
            Else
               GoTo fin
            End If
         Next i
      End If

   Next t

fin:
```

(si no pudimos definir TourTemporal(t) con t menor o igual a CantidadDeCiudades es porque no era un tour, de lo contrario, existe un tour)

```
   If t <= CantidadDeCiudades Then
      Check = 0
   Else
      Check = 1
   End If
```

End Function

4) Function VerificacionDeLasX(v_1, v_2, n) Entradas: v_1, v_2, n, vector Salida: VerificacionDeLasX

(v_1 y v_2 son los extremos de una potencial rama x a quebrar y n es el número de ramas a extraer elegidas hasta el momento)

```
VerificacionDeLasX = 0
```

(se comparan v_1 y v_2 con los extremos de todas las ramas $x's$ para verificar si ya fue elegida previamente)
(recordemos que la forma de las x_j es (vector$(2j-1)$, vector$(2j)$))

```
For i = 1 To n
   If (v1 = vector(2 * i) And v2 = vector(2 * i - 1)) Or
       (v1 = vector(2 * i - 1) And v2 = vector(2 * i)) Then
```

(si la rama estudiada es igual a alguna de las ramas $x's$ anteriores se define VerificacionDeLasX $= 1$ y salimos de la subrutina)

```
      VerificacionDeLasX = 1
      Exit For
   End If
Next i
```

End Function

5) Function VerificacionDeLasY(v_1, v_2, n) Entradas: v_1, v_2, n, vector Salida: VerificacionDeLasY

(v_1 y v_2 son los extremos de una potencial rama y a incorporar y n es el número de ramas elegidas para ingresar al tour hasta el momento)

```
VerificacionDeLasY = 0
```

(se comparan v_1 y v_2 con los extremos de todas las ramas $y's$ que han sido previamente seleccionadas para incorporar al tour)
(recordemos que la forma de las y_j es (vector$(2j)$, vector$(2j+1)$))

```
For i = 1 To n - 1
   If (v1 = vector(2 * i) And v2 = vector(2 * i + 1)) Or
       (v1 = vector(2 * i + 1) And v2 = vector(2 * i)) Then
```

(si la rama estudiada es igual a alguna de las ramas $y's$ anteriores se define VerificacionDeLasY $= 1$ y salimos de la subrutina)

```
      VerificacionDeLasY = 1
      Exit For
   End If
Next i
```

End Function

6) Function VerificacionYEstrellaDeX(v_1, v_2) Entradas: v_1, v_2, tour Salida: VerificacionYEstrellaDeX

(v_1 y v_2 son los extremos de una rama que quiere considerarse como y^*)

```
VerificacionYEstrellaDeX = 0
```

(se controla si v_1 y v_2 coinciden con alguna de las ramas del tour original)
(recordemos que la forma de las ramas del tour es (t, tour(j)))

```
For i = 1 To CantidadDeCiudades
   If (v1 = i And v2 = tour(i)) Or (v1 = tour(i) And v2 = i) Then
```

(si la y^* en estudio es alguna de las ramas del tour original se define VerificacionYEstrellaDeX = 1 y salimos de la subrutina)

```
      VerificacionYEstrellaDeX = 1
      Exit For
   End If
Next i
```

End Function

7) Sub BuscoY1() Entradas: vector, tour, TourMenosUno y costo Salida: PosiblesY1 y PesosY1

```
PosiblesY1 = 1

For i = 1 To CantidadDeCiudades
```

(la rama y_1 tiene por extremos al vector(2) y al i que estamos buscando, este i no puede ser el vector(2) pues carece de sentido, tampoco tour(vector(2)) ni TourMenosUno(vector(2)) pues sería una rama del tour)

```
   If i <> tour(vector(2)) And i <> TourMenosUno(vector(2)) And
       i <> vector(2) Then
```

(se calcula $g_1 =\mid x_1 \mid - \mid y_1 \mid$)

```
      g1 = costo(vector(1), vector(2)) - costo(vector(2), i)
      If g1 > 0 Then
```

(en una matriz llamada PesosY1 guardo todas las posibilidades para y_1, en la primera fila los posibles extremos y en la segunda fila sus respectivos pesos)

```
         PesosY1(1, PosiblesY1) = i
         PesosY1(2, PosiblesY1) = costo(vector(2), i)
```

(PosiblesY1 es un contador, indica la cantidad de ramas y_1 a considerar)

```
            PosiblesY1 = PosiblesY1 + 1
         End If
      End If
   Next i
```

End Sub

8) Sub OrdenarY1porPesos() Entradas: PosiblesY1 y PesosY1 Salida: OrdenarY1

(creamos una matriz llamada OrdenarY1 en donde guardaremos en la primera fila los extremos posibles para y_1, y en la segunda fila los respectivos pesos)

```
   For i = 1 To PosiblesY1 - 1
      OrdenarY1(1, i) = PesosY1(1, i)
      OrdenarY1(2, i) = PesosY1(2, i)
   Next i

   For t = 1 To PosiblesY1 - 1
      If PosiblesY1 > 2 Then
         For i = 1 To PosiblesY1 - 2
```

(si el peso de la rama de la posición i es mayor al peso de la rama que está en la posición $i+1$, las cambiamos)

```
            If OrdenarY1(2, i) >= OrdenarY1(2, i + 1) Then
               T1 = OrdenarY1(1, i)
               T2 = OrdenarY1(2, i)
               M1 = OrdenarY1(1, i + 1)
               M2 = OrdenarY1(2, i + 1)
               OrdenarY1(2, i) = M2
               OrdenarY1(1, i) = M1
               OrdenarY1(2, i + 1) = T2
               OrdenarY1(1, i + 1) = T1
            End If
         Next i
      End If
```

(repetimos el procedimiento tantas veces como la cantidad de columnas que tiene la matriz que queremos ordenar)

```
   Next t
```

End Sub

9) Sub BuscoY2QueQuiebreX3() Entradas: vector, tour, TourMenosUno, costo y Check Salida: PosiblesY2, PesosY2

```
PosiblesY2 = 1

For t = 1 To CantidadDeCiudades
```

(hasta aquí, se fijaron las ramas x_1, y_1, x_2 = (vector(3), vector(4)) y buscamos una rama y_2. No puede ser el vector(4) pues no tiene sentido, ni tampoco el vector(1) pues sería imposible el quiebre de una x_3. También descartamos tour(vector(4)) y TourMenos-Uno(vector(4)) pues las ramas $y's$ no pueden pertenecer al tour original. Por último, verificamos que la que elegimos no coincida con y_1 a través de VerificacionDeLasY)

```
If t <> tour(vector(2 * 2)) And t <> TourMenosUno(vector(2 * 2)) And
    t <> vector(2 * 2) And t <> vector(1) Then
   If VerificacionDeLasY(t, vector(2 * 2), 2) = 0 Then
```

(se calcula $G_2 = g_1 + g_2 = (\mid x_1 \mid - \mid y_1 \mid) + (\mid x_2 \mid - \mid y_2 \mid))$

```
G = CalcularG(1) + costo(vector(2 * 2 - 1), vector(2 * 2)) -
       costo(vector(2 * 2), t)
If G > 0 Then
```

(el y_2 tiene que asegurar el quiebre de una x_3, vamos a intentar en un principio con el extremo vector(6)=tour(t))

```
vector(2 * 2 + 1) = t
vector(2 * 2 + 2) = tour(t)
```

(el vector(6) no puede ser el vector(1), ya que, no existiría y^*. Se verifica que la rama x_3 elegida no sea una de las anteriores mediante VerificacionDeLasX. La rama y^* = (vector(6), vector(1)) no puede pertenecer al tour original por eso controlamos que esto no suceda con VerificacionYEstrellaDeX)

```
If tour(t) <> vector(1) Then
   If VerificacionDeLasX(tour(t), t, 2) = 0 And
       VerificacionYEstrellaDeX(tour(t), vector(1)) = 0 Then
```

(se estudia si al extraer las ramas x_1, x_2, x_3 y al agregar las ramas y_1, y_2, y^* lo que se obtiene es un tour o no)

```
RamasQueQuedan (2 + 1)
CargandoRamas (2 + 1)
```

(si el intento con vector(6) = tour(t) es bueno, tenemos que guardar la información)

```
If Check() = 1 Then
```

(creamos una matriz llamada PesosY2 en donde en la primera fila guardamos los posibles y_2, en la segunda fila los respectivos pesos y en la tercera fila el x_3 a quebrar en cada opción)

```
                        PesosY2(1, PosiblesY2) = t
                        PesosY2(2, PosiblesY2) = costo(vector(2 * 2), t)
                        PesosY2(3, PosiblesY2) = vector(2 * 2 + 2)
                        PosiblesY2 = PosiblesY2 + 1
                     Else
                        GoTo uuuu
                     End If 'Check
                  Else
                     GoTo uuuu
                  End If 'VerificacionDeLasX
               Else
uuuu:
```

(el intento con vector(6) = tour(t) fracasó, consideramos la otra opción vector(6) = TourMenosUno(t))

```
                  vector(2 * 2 + 2) = TourMenosUno(t)

                  If TourMenosUno(t) <> vector(1) Then
                     If VerificacionDeLasX(TourMenosUno(t), t, 2) = 0 And
                     VerificacionYEstrellaDeX(TourMenosUno(t), vector(1)) = 0
                     Then
                        RamasQueQuedan (2 + 1)
                        CargandoRamas (2 + 1)
                        If Check() = 1 Then
                           PesosY2(1, PosiblesY2) = t
                           PesosY2(2, PosiblesY2) = costo(vector(2 * 2), t)
                           PesosY2(3, PosiblesY2) = vector(2 * 2 + 2)
                           PosiblesY2 = PosiblesY2 + 1
                        End If 'del Check
                     End If 'de VerificacionDeLasX
                  End If 'de TourMenosUno
               End If 'de tour
            End If 'de G > 0
         End If 'de VerificacionDeLasY
      End If 'del t<>
   Next t
```

End Sub

10) Sub OrdenarY2PorPesos() Entradas: PosiblesY2 y PesosY2 Salida: OrdenarY2

(creamos una matriz llamada OrdenarY2 en donde guardamos en la primera fila los extremos posibles para y_2, en la segunda fila los respectivos pesos y en la tercera fila el x_3 a quebrar en cada elección)

```
For i = 1 To PosiblesY2 - 1
   OrdenarY2(1, i) = PesosY2(1, i)
   OrdenarY2(2, i) = PesosY2(2, i)
   OrdenarY2(3, i) = PesosY2(3, i)
Next i

For t = 1 To PosiblesY2
   If PosiblesY2 > 2 Then
      For i = 1 To PosiblesY2 - 2
```

(si el peso de la rama de la posición i es mayor al peso de la rama que está en la posición $i + 1$, las cambiamos)

```
            If OrdenarY2(2, i) >= OrdenarY2(2, i + 1) Then
               T1 = OrdenarY2(1, i)
               T2 = OrdenarY2(2, i)
               T3 = OrdenarY2(3, i)
               M1 = OrdenarY2(1, i + 1)
               M2 = OrdenarY2(2, i + 1)
               M3 = OrdenarY2(3, i + 1)
               OrdenarY2(2, i) = M2
               OrdenarY2(1, i) = M1
               OrdenarY2(3, i) = M3
               OrdenarY2(2, i + 1) = T2
               OrdenarY2(1, i + 1) = T1
               OrdenarY2(3, i + 1) = T3
            End If
         Next i
      End If
```

(repetimos el procedimiento tantas veces como la cantidad de columnas que tiene la matriz que queremos ordenar)

```
   Next t
```

End Sub

11) Sub BuscoYnQueQuiebreXSiguiente(n) Entradas: vector, tour, TourMenosUno, costo y Check Salida: PosiblesYn, vector

(n representa el número de la rama y que se busca)

```
   PosibleYn = 0
   R = MaxNum

   For t = 1 To CantidadDeCiudades
```

(recordemos que la rama y_n que queremos hallar es de la forma (vector($2n$), vector($2n+1$)) por lo tanto no tiene sentido que t sea igual a vector($2n$) ni el vector(1) pues no podría quebrar una x_{n+1}. También sacamos como opciones TourMenosUno(vector($2n$)) y tour(vector($2n$)) pues las ramas y no pueden ser ramas del tour. Por último, al pedir que VerificacionDeLasY(t, vector($2n$), n) $= 0$ se intenta que no sea ninguna de las ramas y elegidas anteriormente)

```
If t <> tour(vector(2 * n)) And t <> TourMenosUno(vector(2 * n)) And
    t <> vector(2 * n) And t <> vector(1) Then
   If VerificacionDeLasY(t, vector(2 * n), n) = 0 Then
      If costo(vector(2 * n), t) < R Then
```

(se calcula $G_n = g_1 + ... + g_n = (\mid x_1 \mid - \mid y_1 \mid) + ... + (\mid x_n \mid - \mid y_n \mid)$)

```
G = CalcularG(n - 1) + costo(vector(2 * n - 1), vector(2 * n)) -
       costo(vector(2 * n), t)
If G > 0 Then
```

(el y_n tiene que asegurar el quiebre de un x_{n+1}, vamos a intentar en un principio con el extremo vector($2n+2$) $=$ tour(t))

```
vector(2 * n + 1) = t
vector(2 * n + 2) = tour(t)
```

(la rama x_{n+1} no puede ser una de las ramas $x's$ previas, por eso se pide que la VerificacionDeLasX(tour(t), t, n) $= 0$. No puede ser el vector(1) pues no tendría sentido lo de la rama y^*, además esta última rama no puede ser una del tour pues es una rama de las y, es eso lo que se corrobora en VerificacionYEstrellaDeX)

```
If tour(t) <> vector(1) Then
   If VerificacionDeLasX(tour(t), t, n) = 0 And
       VerificacionYEstrellaDeX(tour(t), vector(1)) = 0
       Then
```

(se estudia si al extraer las ramas x_1, ..., x_{n+1} y al agregar y_1, ..., y_n, y^* lo que resulta es un tour o no)

```
RamasQueQuedan (n + 1)
CargandoRamas (n + 1)
```

(si el intento con el vector($2q+2$) $=$ tour(t) es bueno, guardamos toda la información -extremos y peso. Se busca la rama de menor peso)

```
If Check() = 1 Then
   VectorDosNMasUno = t
   VectorDosNMasDos = tour(t)
   R = costo(vector(2 * n), t)
   PosibleYn = 1
```

```
                        Else
                           GoTo uuuu
                        End If
                     Else
                        GoTo uuuu
                     End If
                  Else
```

uuuu:

(el intento con el vector$(2n+2)$ = tour(t) fracasó, vamos a probar con el otro extremo)

```
                     vector(2 * n + 2) = TourMenosUno(t)

                     If TourMenosUno(t) <> vector(1) Then
                        If VerificacionDeLasX(TourMenosUno(t), t, n) = 0 And
                     VerificacionYEstrellaDeX(TourMenosUno(t), vector(1)) = 0
                     Then
                           RamasQueQuedan (n + 1)
                           CargandoRamas (n + 1)
                           If Check() = 1 Then
                              VectorDosNMasUno = t
                              VectorDosNMasDos = TourMenosUno(t)
                              R = costo(vector(2 * n), t)
                              PosibleYn = 1
                           End If 'Check
                        End If 'VerificacionDeLasX
                     End If 'TourMenosUno
                  End If 'tour(t)
               End If 'G > 0
            End If 'costo(vector(2*n),t)
         End If 'VerificacionDeLasY
      End If 't<>
   Next t

   vector(2 * n + 1) = VectorDosNMasUno
   vector(2 * n + 2) = VectorDosNMasDos
```

End Sub

12) Sub CopiarATourNuevo() Entradas: TourTemporal Salida: TourNuevo

(si se obtienen mejoras, guardamos en el vector TourNuevo el tour con el cual fue posible realizarlas)

```
   For i = 1 To CantidadDeCiudades
```

```
      TourNuevo(i) = TourTemporal(i)
   Next i
```

(la forma en que cargamos el tour en el vector es la siguiente: TourNuevo(1), TourNuevo(2), ..., TourNuevo(CantidadDeCiudades) y la rama que une el último vértice y el primero del camino expuesto anteriormente)
(recordemos que a TourTemporal lo definimos en la subrutina Check, y que si llegamos hasta la subrutina en que nos hallamos es porque al chequear la posible existencia de un tour se obtuvo un sí como respuesta)
End Sub

13) Sub CopiarATourOriginal() Entradas: TourNuevo Salida: tour y TourMenosUno

(una vez que se tiene la seguridad de que el tour guardado en TourNuevo no puede ser mejorado se lo almacena en los vectores tour y TourMenosUno)

```
   For i = 1 To CantidadDeCiudades - 1
      tour(TourNuevo(i)) = TourNuevo(i + 1)
      TourMenosUno(TourNuevo(i + 1)) = TourNuevo(i)
   Next i
   tour(TourNuevo(42)) = TourNuevo(1)
   TourMenosUno(TourNuevo(1)) = TourNuevo(42)
```

(se reinicia el algoritmo con este nuevo tour inicial)
End Sub

14) Function CalcularG(n) Entradas: vector Salida: G

```
   G = 0
```

(se calcula $G_n = g_1 + ... + g_n = (\mid x_1 \mid - \mid y_1 \mid) + ... + (\mid x_n \mid - \mid y_n \mid)$)

```
   For i = 1 To n
      G = G + costo(vector(2 * i - 1), vector(2 * i)) -
              costo(vector(2 * i), vector(2 * i + 1))
   Next i

   CalcularG = G
```

End Function

15) GUnoMenosMasGYEstrella(n) Entradas: vector Salida: GConYEstrella

(se calcula $G_{n-1} + g_n^* = (\mid x_1 \mid - \mid y_1 \mid) + ... + (\mid x_{n-1} \mid - \mid y_{n-1} \mid) + (\mid x_n \mid - \mid y_n^* \mid)$)

```
   GConYEstrella = CalcularG(n - 1) + (costo(vector(2 * n - 1), vector(2 * n)) -
                                costo(vector(2 * n), vector(1)))

   GUnoMenosMasGYEstrella = GConYEstrella
```

End Function

Estructura principal

```
Empezarotravez:
Gast = 0
For w = 1 To CantidadDeCiudades
```

(se fijan los extremos de la rama x_1)

```
   vector(1) = w
   vector(2) = tour(w)

' Busco Y1, si no encuentro, backtracking

CambioX1:
   BuscoY1
```

(si las posibilidades para y_1 son mayores o iguales a dos, las ordenamos según el peso y nos quedamos con la más liviana)

```
   If PosiblesY1 >= 2 Then
      OrdenarY1porPesos
      GoTo InicioX2
FinX2:
   End If
```

(si no encontramos y_1, vamos a cambiar la rama x_1 utilizando la otra posibilidad del tour original, consideramos el extremo TourMenosUno(vector(1)) e intentamos hallar y_1 nuevamente)

```
   If vector(2) = tour(vector(1)) Then
      vector(2) = TourMenosUno(vector(1))
      GoTo CambioX1
   End If
```

(si llegamos hasta aquí es porque ya probamos con las dos chances para x_1 y no hallamos y_1, lo único que resta es cambiar el nodo inicial)

```
CambioT1:
Next w

Exit Sub

'***************************** Busco X2 e Y2, si no encuentro, backtracking

InicioX2:
vector(3) = OrdenarY1(1, 1)
```

(calculamos $g_1 = |\ x_1\ | - |\ y_1\ |$ y buscamos la rama x_2)

```
g1 = CalcularG(1)
ContadorY1 = 1

BuscoX2:
```

(intentamos con tour(vector(3)) que es una de las alternativas para x_2)

```
vector(4) = tour(vector(3))
tnc = 0
CambioX2:
```

(constatamos que la rama y^* no sea una del tour, pedimos que el vector(4) no sea el vector(1) porque no tendría sentido lo de y^*)

```
If VerificacionYEstrellaDeX(vector(4), vector(1)) = 0 And
    vector(4) <> vector(1) Then
   RamasQueQuedan (2)
   CargandoRamas (2)
```

(analizamos la existencia de tour luego de extraer y agregar las ramas correspondientes)

```
   If Check() = 1 Then
```

(si hay tour y $g_1 + g^*$ es mayor a G^* entonces actualizamos G^*, guardamos este tour con el que logramos mejoras en el vector TourNuevo y vamos en busca de y_2)

```
      If GUnoMenosMasGYEstrella(2) > Gast Then
         Gast = GUnoMenosMasGYEstrella(2)
         CopiarATourNuevo
      End If
```

(no pudimos actualizar G^*, continuamos buscando y_2)

```
buscoy2:
      BuscoY2QueQuiebreX3
```

(si las posibilidades para y_2 son mayores o iguales a dos, las ordenamos según el peso y nos quedamos con la más liviana)

```
      If PosiblesY2 >= 2 Then
         OrdenarY2PorPesos
         GoTo InicioYyXqueSigue
      End If
```

(no encontramos ningún y_2, si G^* es positivo cargamos el tour que tenemos en TourNuevo en el tour original y volvemos a empezar)

```
      If Gast > 0 Then
         CopiarATourOriginal
         GoTo Empezarotravez
      End If
FinYyXqueSigue:
```

(no encontramos ningún y_2, como G^* es 0 hacemos backtracking)
(se utiliza la otra opción para x_2 aunque rompemos momentáneamente el criterio de factibilidad y vamos a buscar y_2)

```
      If vector(4) = tour(vector(3)) And tnc = 0 Then
         vector(4) = TourMenosUno(vector(3))
         tnc = 1
         GoTo buscoy2
      Else
         If tnc = 0 Then
            vector(4) = tour(vector(3))
            tnc = 1
            GoTo buscoy2
         End If
      End If
```

(cambiamos la rama y_1, vamos utilizando, una por una, las posibilidades que tenemos ordenadas en la primera fila de la matriz OrdenarY1)

```
      If PosiblesY1 > 2 Then
         If ContadorY1 < PosiblesY1 - 1 Then
            vector(3) = OrdenarY1(1, ContadorY1 + 1)
            g1 = CalcularG(1)
            ContadorY1 = ContadorY1 + 1
            GoTo BuscoX2
         End If
      End If
```

(si no encontramos y_1, realizamos bactracking, estudiamos si podemos cambiar x_1)

```
      GoTo FinX2
   End If 'check
```

(el intento con vector(4) = tour(vector(3)) fracasó porque al extraer y al agregar las ramas correspondientes no existe tour, consideramos la otra posibilidad para vector(4))

```
End If 'VerificacionYEstrellaDeLosX
```

(el intento con vector(4) = tour(vector(3)) fracasó porque el vector(4) es igual al vector(1) o y^* es alguna rama del tour, probamos suerte con TourMenosUno(vector(3)))

```
vector(4) = TourMenosUno(vector(3))
GoTo CambioX2

'**************************** Busco Yn y Xn+1 (n >= 3), si no encuentro,
backtracking

InicioYyXqueSigue:
ContadorY2 = 1
```

(tenemos la rama y_2 y x_3)

```
vector(5) = OrdenarY2(1, 1)
vector(6) = OrdenarY2(3, 1)
```

(calculamos $G_2 = g_1 + g_2 = (\mid x_1 \mid - \mid y_1 \mid) + (\mid x_2 \mid - \mid y_2 \mid))$
(si G es menor o igual a G^* detenemos la búsqueda, cargamos el tour que teníamos en TourNuevo en el tour original y volvemos a empezar el proceso)

```
If CalcularG(2) <= Gast Then
   CopiarATourOriginal
   GoTo Empezarotravez
Else
```

(si G es mayor a G^* continuamos buscando y_3)

```
   m = 3
   GoTo BuscoYyXqueSigue
End If

BuscoYyXqueSigue:
```

(anteriormente encontramos x_m, ahora verificamos si con la y^* se producen mejoras para actualizar G^* y guardar el tour con el que fue posible realizarlas)

```
If GUnoMenosMasGYEstrella(m) > Gast Then
   Gast = GUnoMenosMasGYEstrella(m)
   RamasQueQuedan (m)
   CargandoRamas (m)
   Check
   CopiarATourNuevo
End If
```

(no actualizamos G^*, continuamos con el procedimiento)

```
BuscoYm:
BuscoYnQueQuiebreXSiguiente (m)
If PosibleYn = 1 Then
```

(logramos hallar y_m, si G es menor o igual a G^*, entonces cargamos el tour que tenemos en TourNuevo en el tour original y volvemos a empezar)

```
   If CalcularG(m) <= Gast Then
      CopiarATourOriginal
      GoTo Empezarotravez
   Else
```

(tenemos y_m, G es mayor a G^*, continuamos el proceso)

```
      m = m + 1
      GoTo BuscoYyXqueSigue
   End If
End If
```

(nos fue imposible obtener una y_m, G^* es positivo, cargamos el tour que tenemos en TourNuevo en el tour original y volvemos a empezar)

```
If Gast > 0 Then
   CopiarATourOriginal
   GoTo Empezarotravez
End If
```

(G^* es cero, cambiamos la rama y_2, vamos utilizando una por una las posibilidades que tenemos ordenadas en la primera fila de la matriz OrdenarY2)

```
If PosiblesY2 > 2 Then
   If ContadorY2 < PosiblesY2 - 1 Then
      vector(5) = OrdenarY2(1, ContadorY2 + 1)
      vector(6) = OrdenarY2(3, ContadorY2 + 1)
      ContadorY2 = ContadorY2 + 1
      m = 3
      GoTo BuscoYyXqueSigue
   End If
End If
```

(no es posible hallar y_2, bactracking a x_2)

```
GoTo FinYyXqueSigue
```

Fin de la Estructura principal

Utilizamos tres tipos diferentes de tours iniciales para testear los diferentes problemas:
a) Tipo I
For ciudad = 1 To CantidadDeCiudades-1
tour(ciudad) = ciudad + 1

TourMenosUno(ciudad + 1) = ciudad
Next ciudad
tour(CantidadDeCiudades) = 1
TourMenosUno(1) = CantidadDeCiudades

b) Tipo II
For ciudad = 1 To CantidadDeCiudades/2 - 1 (o CantidadDeCiudades/2 -1 si CantidadDeCiudades es impar)
tour(2 * ciudad) = 2 * ciudad + 2
TourMenosUno(2 * ciudad + 2) = 2 * ciudad
Next ciudad
For ciudad = 1 To CantidadDeCiudades/2 - 1 (o CantidadDeCiudades/2 si CantidadDeCiudades es impar)
tour(2 * ciudad + 1) = 2 * ciudad - 1
TourMenosUno(2 * ciudad - 1) = 2 * ciudad + 1
Next ciudad
tour(1) = 2
TourMenosUno(2) = 1
tour(CantidadDeCiudades-1) = CantidadDeCiudades
TourMenosUno(CantidadDeCiudades) = CantidadDeCiudades-1

c) Tipo III
For ciudad = 1 To CantidadDeCiudades-1
tour(ciudad) = ciudad + 1
TourMenosUno(ciudad + 1) = ciudad
Next ciudad
tour(CantidadDeCiudades) = 1
TourMenosUno(1) = CantidadDeCiudades
tour(1) = 3
TourMenosUno(3) = 1
tour(3) = 5
TourMenosUno(5) = 3
tour(CantidadDeCiudades) = 2
TourMenosUno(2) = CantidadDeCiudades
tour(2) = 4
TourMenosUno(4) = 2
tour(4) = 1
TourMenosUno(1) = 4

Resumimos los resultados obtenidos en la siguiente tabla:

	Tours Iniciales						Peso del mejor tour conocido
	Tipo I		Tipo II		Tipo III		
	Nº iterac.	Peso inicial / Peso final	Nº iterac.	Peso inicial / Peso final	Nº iterac.	Peso inicial / Peso final	
DFJ42	1	699 / 699	21	1211 / 704	3	773 / 699	699
LK105	66	36480 / 14514	73	57459 / 14940	57	38709 / 14401	14383
LK318	189	119872 / 42822	233	191971 / 43033	169	122120 / 43081	41345

3.4.4. Refinamientos

El algoritmo básico limita la búsqueda usando las siguientes cuatro reglas:
(1) sólo cambios secuenciales están permitidos
(2) la ganancia transitoria debe ser positiva
(3) el tour tiene que ser "cerrado" (con una excepción, $i = 2$)
(4) una rama extraída previamente no puede ser agregada, y una rama adicionada anteriormente no puede ser eliminada

Para limitar la búsqueda aún más, Lin y Kernighan le realizaron algunos refinamientos al algoritmo introduciendo las siguientes reglas:
(5) la búsqueda de una rama que será agregada al tour $y_i = (t_{2i}, t_{2i+1})$ se limita a los cinco vecinos más cercanos de t_{2i}
(6) para $i \geq 4$ ninguna rama x_i del tour debe eliminarse si es una rama común de un número pequeño $(2 - 5)$ de tours soluciones
(7) la búsqueda de mejoras se detiene si el tour en curso es el mismo a algún tour solución previo

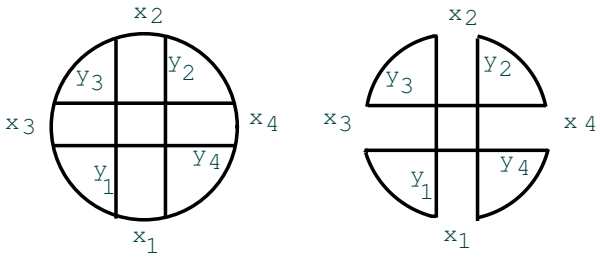
Las reglas 5 y 6 son heurísticas, ahorran tiempo de ejecución pero en algunas ocasiones a expensas de no alcanzar la mejor solución posible, en cambio, la regla 7 también economiza tiempo de ejecución pero no tiene influencia sobre la calidad de las soluciones a encontrar. Si un tour es el mismo a un tour solución previo, es sabido que en ningún nodo podrá mejorarse, el tiempo necesario para chequear esto (checkout time) puede evitarse.

Además de estos refinamientos cuyos propósitos son limitar la búsqueda, Lin y Kernighan agregaron otros que se orientan a dirigirla. En aquellos lugares donde el algoritmo tiene la opción de distintas alternativas, se utilizan reglas heurísticas para dar prioridad a las mismas. En el caso donde sólo una de ellas puede ser elegida, se elige aquella que tiene mayor prioridad. En cambio, cuando varias alternativas pueden ser probadas, se las considera en orden descendiente de prioridad. Para ser más específicos:

(8) cuando se elige la rama y_i (i $\geq$ 2) se tiene que tratar de maximizar | x_{i+1} | - | y_i |
(9) si hay dos alternativas para x_4, se opta por aquella que maximice | x_4 |

La regla 8 es heurística, la prioridad para y_i es el tamaño de la próxima rama a quebrar x_{i+1} (lookahead). Se trata de quebrar una rama pesada incluyendo una rama liviana. La regla 9 trata una situación especial donde hay dos opciones para x_4, dándosele preferencia a la de mayor peso. En otros casos (elección de x_1, x_2 y a veces x_3) hay dos alternativas posibles, en estas situaciones el algoritmo examina las dos opciones usando backtracking (a menos que un tour mejor fuese encontrado). Lin y Kernighan no especifican las secuencias en que las alternativas son examinadas.

Como un último refinamiento Lin y Kernighan incluyen una limitada defensa frente a las situaciones donde sólo un cambio no secuencial puede conducir a una solución mejor. Después de que un óptimo local ha sido encontrado, el algoritmo examina entre las ramas a quebrar si es posible hacer una mejora a través de un cambio no secuencial 4-opt.



Como comentario final, queremos agregar que K. Helsgaun escribió un paper más reciente en donde se describe la implementación de una nueva versión modificada del algoritmo heurístico de Lin y Kernighan. Habiéndosele realizado pruebas computacionales, se concluye que es altamente efectiva, logrando resolver una instancia de 13509 ciudades y mejorando la solución conocida hasta el momento de otra de 85900 ciudades, de la que se desconoce el óptimo [9].

Capítulo 4

Secuencia de trabajos

4.1. Descripción del problema

Vamos a considerar el problema de secuenciar N trabajos $J_1, ..., J_N$ en una máquina [10]. Cada uno de ellos tiene asociados dos números A_i y B_i. Para comenzar el trabajo i, la máquina debe estar en el estado A_i y cuando el mismo se complete el estado habrá cambiado automáticamente a B_i. Para realizar el trabajo J_j a continuación de J_i, el estado de la máquina debe transformarse a A_j. El costo de este cambio es c_{ij} y se define como

$$c_{ij} = \begin{cases} \displaystyle\int_{B_i}^{A_j} f(x)dx & \text{si } B_i \leq A_j \\ \displaystyle\int_{A_j}^{B_i} g(x)dx & \text{si } A_j < B_i \end{cases}$$

donde $f(x)$ y $g(x)$ son funciones integrables que satisfacen $f(x) + g(x) \geq 0$. Podemos interpretar a $f(x)$ como el costo de incrementar la variable estado y $g(x)$ el costo de disminuirla. La restricción $f(x) + g(x) \geq 0$ implica que no se obtiene ganancia si se incrementa y luego se disminuye al estado original la variable estado.Veremos un método para encontrar el orden en el que deben realizarse los trabajos J_i de forma tal que se minimice el costo total.

Este problema se relaciona con el problema del viajante, los J_i juegan el rol de los nodos o ciudades y c_{ij} representa el costo de ir del nodo i al nodo j. Buscamos el camino de mínimo costo que pase una sola vez por cada nodo. Sin embargo, en el TSP se quiere hallar un ciclo hamiltoniano de mínimo costo.

Si agregamos un nuevo trabajo J_0, existe una correspondencia entre los tours $J_0 J_{i_1} ... J_{i_N} J_0$ y las secuencias $J_{i_1} ... J_{i_N}$ del problema original. Si asumimos que inicialmente la máquina se encuentra en un estado B_0, luego se realizan los N trabajos y al concluirse, la máquina debe quedar en el estado A_0, podemos pensar al problema de secuenciar los

trabajos como el problema del viajante. Por lo tanto, podemos minimizar sobre tours y luego descartando J_0 tendremos la secuencia de menor costo.

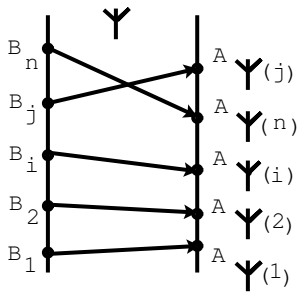
Por lo comentado anteriormente, nos concentraremos en el problema de encontrar un tour de mínimo costo. Lo plantearemos en términos de hallar una permutación ψ para la cual el costo total $c(\psi)$,

$$c(\psi) = \sum_{i=1}^{N} c_{i\psi(i)},$$

es mínimo sujeto a la condición de que ψ sea un tour (es decir, que $\psi(S) \neq S$ para todo subconjunto $S \subset \{1, 2, ..., N\}$). El método para resolver el problema será el siguiente. Primero, encontraremos una permutación φ que minimice la sumatoria $\sum_{i=1}^{N} c_{i\varphi(i)}$. Luego, a través de una serie de intercambios convertiremos a la permutación φ en un tour ψ.

4.2. Intercambios. Costo de los intercambios

Vamos a asumir que los B_i están numerados de forma que si $j > i$ entonces $B_j \geq B_i$. Pensaremos que se hallan ordenados sobre una recta real en la posición correspondiente a sus valores. Resultaría natural disponer los A_i sobre la misma recta pero para observar mejor a la permutación ψ, los ubicamos sobre otra en forma separada.



Las ramas unen el estado final B_i del trabajo J_i con el estado inicial $A_{\psi(i)}$ del trabajo posterior $J_{\psi(i)}$.

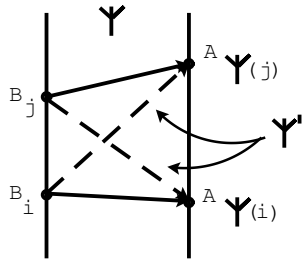
A continuación, definiremos lo que entenderemos por intercambio y calcularemos su efecto sobre el costo $c(\psi)$. Un intercambio α_{ij} es una permutación dada por

$$\begin{cases} \alpha_{ij}(i) = j, \\ \alpha_{ij}(j) = i, \\ \alpha_{ij}(k) = k, \ k \neq i, j. \end{cases}$$

Si aplicamos un intercambio a ψ obtenemos una nueva permutación, $\psi' = \psi\alpha_{ij}$. Claramente,

$$\begin{cases} \psi'(i) = \psi(j), \\ \psi'(j) = \psi(i), \\ \psi'(k) = \psi(k),\ k \neq i, j. \end{cases}$$

El efecto de α_{ij} sobre ψ es el intercambio de los sucesores de i y j.



Definimos el costo $c_\psi(\alpha_{ij})$ de aplicar el intercambio α_{ij} a la permutación ψ como

$$c_\psi(\alpha_{ij}) = c(\psi\alpha_{ij}) - c(\psi).$$

Daremos una fórmula para $c_\psi(\alpha_{ij})$, en donde se utilizarán intervalos $[a, b]$, $a \leq b$ sobre la recta real. Para esos intervalos, se usará la siguiente notación

$$|[a, b]|_f = \int_a^b f(x)dx,$$

$$|[a, b]|_g = \int_a^b g(x)dx,$$

$$\|[a, b]\| = \int_a^b (f(x) + g(x))dx.$$

Asumamos que

$$\begin{cases} B_j \geq B_i, \\ A_{\psi(j)} \geq A_{\psi(i)}. \end{cases}$$

Tenemos que

$$c_{i\psi(j)} = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]|_f + |(-\infty, B_i] \bigcap [A_{\psi(j)}, +\infty)|_g \tag{4.1}$$

pues si $A_{\psi(j)} \geq B_i$,

$$c_{i\psi(j)} = \int_{B_i}^{A_{\psi(j)}} f(x)dx = |[B_i, A_{\psi(j)}]|_f = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]|_f$$

y como $(-\infty, B_i] \bigcap [A_{\psi(j)}, +\infty) = \emptyset$ se tiene que

$$c_{i\psi(j)} = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]|_f + |(-\infty, B_i] \bigcap [A_{\psi(j)}, +\infty)|_g;$$

en cambio, si $A_{\psi(j)} < B_i$,

$$c_{i\psi(j)} = \int_{A_{\psi(j)}}^{B_i} g(x)dx = |[A_{\psi(j)}, B_i]|_g = |(-\infty, B_i] \bigcap [A_{\psi(j)}, +\infty)|_g$$

y al ser $[B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}] = \emptyset$ se tiene que

$$c_{i\psi(j)} = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]|_f + |(-\infty, B_i] \bigcap [A_{\psi(j)}, +\infty)|_g.$$

De la misma manera se observa que

$$c_{i\psi(i)} = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}]|_f + |(-\infty, B_i] \bigcap [A_{\psi(i)}, +\infty)|_g. \tag{4.2}$$

Restando (4.1) menos (4.2),

$$c_{i\psi(j)} - c_{i\psi(i)} = \int_{[B_i,+\infty) \bigcap (-\infty,A_{\psi(j)}]} f(x)\ dx + \int_{(-\infty,B_i] \bigcap [A_{\psi(j)},+\infty)} g(x)\ dx$$
$$- \int_{[B_i,+\infty) \bigcap (-\infty,A_{\psi(i)}]} f(x)\ dx - \int_{(-\infty,B_i] \bigcap [A_{\psi(i)},+\infty)} g(x)\ dx.$$

Como $[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \subset [B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]$ vale que,

$$([B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]) - ([B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}]) =$$
$$([B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]) \bigcap ([B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}])^c =$$
$$([B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]) \bigcap ([B_i, +\infty)^c \bigcup (-\infty, A_{\psi(i)}]^c) =$$
$$([B_i, +\infty) \bigcap (-\infty, A_{\psi(j)}]) \bigcap ((-\infty, B_i) \bigcup (A_{\psi(i)}, +\infty)) =$$
$$[B_i, +\infty) \bigcap (A_{\psi(i)}, A_{\psi(j)}];$$

análogamente, $((-\infty, B_i] \bigcap [A_{\psi(i)}, +\infty)) - ((-\infty, B_i] \bigcap [A_{\psi(j)}, +\infty)) = (-\infty, B_i] \bigcap [A_{\psi(i)}, A_{\psi(j)})$, por lo que finalmente se obtiene que

$$c_{i\psi(j)} - c_{i\psi(i)} = \int_{[B_i,+\infty) \bigcap [A_{\psi(i)},A_{\psi(j)}]} f(x)\ dx - \int_{(-\infty,B_i] \bigcap [A_{\psi(i)},A_{\psi(j)}]} g(x)\ dx.$$

En consecuencia,

$$c_{i\psi(j)} - c_{i\psi(i)} = |[B_i, +\infty) \bigcap [A_{\psi(i)}, A_{\psi(j)}]|_f - |(-\infty, B_i] \bigcap [A_{\psi(i)}, A_{\psi(j)}]|_g.$$

Similarmente

$$c_{j\psi(i)} - c_{j\psi(j)} = -|[B_j, +\infty) \bigcap [A_{\psi(i)}, A_{\psi(j)}]|_f + |(-\infty, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]|_g.$$

Sumando,

$$c_\psi(\alpha_{ij}) = c(\psi\alpha_{ij}) - c(\psi) = c_{i\psi(j)} + c_{j\psi(i)} - c_{i\psi(i)} - c_{j\psi(j)} = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]\|.$$

$$\boxed{c_\psi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]\|}$$

Como $f(x) + g(x) \geq 0$ entonces

$$c_\psi(\alpha_{ij}) \geq 0.$$

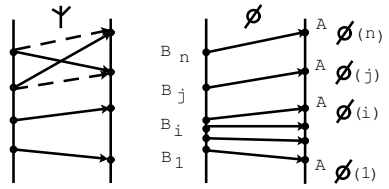
Recordemos que estábamos asumiendo que $B_j \geq B_i$ y que $A_{\psi(j)} \geq A_{\psi(i)}$; si esto no valiera y el orden de $A_{\psi(j)}$ y $A_{\psi(i)}$ fuese contrario al orden de B_j y B_i, la última expresión estaría precedida por un signo menos, o sea,

$$c_\psi(\alpha_{ij}) = -\|[B_i, B_j] \bigcap [A_{\psi(j)}, A_{\psi(i)}]\| \leq 0.$$

Teorema 1: Sea φ una permutación que ranquea los A, es decir, si $j > i$ vale que $A_{\varphi(j)} \geq A_{\varphi(i)}$; entonces

$$c(\varphi) = \text{mín}_\psi\, c(\psi).$$

Demostración: Si una permutación ψ contiene un orden contrario entre B_j, B_i y $A_{\psi(j)}$, $A_{\psi(i)}$, entonces el costo de aplicar α_{ij} es negativo o cero. Como $c(\psi\alpha_{ij}) - c(\psi) = c_\psi(\alpha_{ij}) \leq 0$, deducimos que $c(\psi\alpha_{ij})) \leq c(\psi)$. Aplicando en forma sucesiva distintos intercambios, cualquier ψ es reducida a la permutación φ que ranquea los A; por lo anterior, resulta que φ tiene costo menor o igual que $c(\psi)$.



4.3. Tours y árboles

Analizaremos ahora el efecto de un intercambio α_{ij} sobre los ciclos de una permutación ψ.

Lema 1: Si ψ es una permutación cuyos ciclos disjuntos son $C_1 ... C_p$ y α_{ij} es un intercambio con $i \in C_r$ y $j \in C_s$, $r \neq s$, entonces $\psi\alpha_{ij}$ contiene los mismos ciclos que ψ excepto C_r y C_s que son reemplazados por un único ciclo que contiene todos sus nodos.

Demostración: Llamemos $\overline{C_v}$ al conjunto de nodos de C_v para $v = 1, ..., p$. Si $v \neq r, s$ vale que;

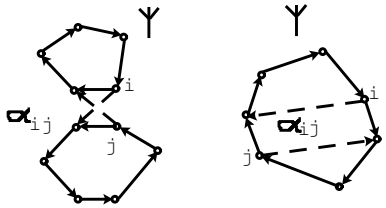
$$\psi\alpha_{ij}(\overline{C_v}) = \overline{C_v}.$$

Pensemos al ciclo C_r como $i \to a_1 \to a_2 \to ... \to a_h \to ... \to a_{m_r} \to i$ y al ciclo C_s como $j \to b_1 \to b_2 \to ... \to b_l \to ... \to b_{m_s} \to j$. Observemos lo que sucede cuando

tomamos i y le aplicamos en forma sucesiva la permutación $\psi\alpha_{ij}$: $i \to b_1 \to b_2 \to ... \to b_l \to ... \to b_{m_s} \to j \to a_1 \to a_2 \to ... \to a_h \to ... \to a_{m_r} \to i$.

Podemos concluir que se forma un único ciclo que incluye todos los nodos de C_r y C_s.

Si i y j son del mismo ciclo, el efecto es contrario y se separa.



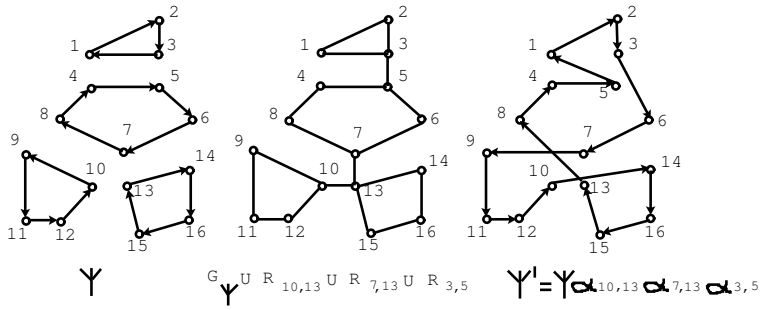
Para cualquier permutación ψ definiremos el grafo no dirigido G_ψ que tiene N nodos y ramas del nodo i al nodo $\psi(i)$ para $i = 1, ..., N$. Claramente existe una correspondencia entre los ciclos de ψ y las componentes conexas de G_ψ. Para cada ciclo de ψ hay una componente conexa de G_ψ que contiene el mismo conjunto de nodos. Esta correspondencia puede mantenerse bajo ciertos cambios. Supongamos que a G_ψ le agregamos un arco R_{ij} que conecta los nodos i y j en componentes diferentes. El efecto será dejar todas las otras componentes iguales y unir las dos componentes en cuestión. Como i y j se encuentran en ciclos diferentes de ψ, el Lema 1 nos dice que las componentes de $G_\psi \bigcup R_{ij}$ se corresponden con los ciclos de $\psi\alpha_{ij}$.

Comenzando con un grafo G_ψ podemos agregar un conjunto de arcos para convertirlo en conexo. Si G_ψ tiene p componentes, el mínimo número de arcos para conectarlo es $p-1$. Llamaremos a un conjunto minimal de arcos adicionales que conecta al grafo G_ψ un "spanning tree".

Teorema 2: Sean $\alpha_{i_1 j_1}, \alpha_{i_2 j_2}, ..., \alpha_{i_{p-1} j_{p-1}}$ los intercambios correspondientes a los arcos de un spanning tree de G_ψ (los arcos pueden ser considerados en cualquier orden). Entonces la permutación ψ' dada por

$$\psi' = \psi\alpha_{i_1 j_1}\alpha_{i_2 j_2}...\alpha_{i_{p-1} j_{p-1}}$$

es un tour.



Demostración: Necesitamos demostrar que la correspondencia entre ciclos y componentes conexas se mantiene, porque como $G_\psi \bigcup R_{i_1 j_1} \bigcup R_{i_2 j_2} \bigcup ... \bigcup R_{i_{p-1} j_{p-1}}$ es conexo por la definición de spanning tree, esa correspondencia permite sólo un ciclo para ψ'. Sabemos que la correspondencia se mantiene si cada arco conecta dos componentes. Sin embargo, como estamos usando los arcos de un spanning tree esta propiedad de dos componentes es automática: si para uno de esos arcos, cuando lo agregamos, se conectan nodos de una misma componente entonces los arcos restantes sin aquél deben hacer conexo a G_ψ, lo que contradice la minimalidad del conjunto conector.

4.4. El costo de un árbol. Propiedad especial del mínimo árbol

Habiendo establecido la conexión entre árboles y tours, estudiaremos el costo de los arcos de los árboles. Empezaremos con el grafo G_φ que resulta del uso de la permutación mínima φ. Para el arco R_{ij} correspondiente al intercambio α_{ij}, asignamos el costo $c_\varphi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\varphi(i)}, A_{\varphi(j)}]\|$. Por el costo $c_\varphi(\tau)$ de un árbol τ entendemos a la suma de los costos de los intercambios correspondientes a sus arcos

$$c_\varphi(\tau) = \sum_{R_{ij} \in \tau} c_\varphi(\alpha_{ij}).$$

Hallar un árbol de mínimo costo es sencillo, sólo tenemos que aplicar el método de Kruskal [**?**] considerando las componentes de G_φ como puntos. Se puede hallar un mínimo spanning tree con una propiedad adicional:

Lema 2: Existe un spanning tree de mínimo costo para G_φ que contiene sólo arcos $R_{i,i+1}$.

Demostración: Supongamos que τ es un árbol mínimo que contiene el arco R_{ij}, $j > i+1$. Como $A_{\varphi(j)}$ y $A_{\varphi(i)}$ están en el mismo orden que B_j y B_i tenemos

$$c_\varphi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\varphi(i)}, A_{\varphi(j)}]\| = \|\{\bigcup_{p=i}^{j-1} [B_p, B_{p+1}]\} \bigcap \{\bigcup_{p=i}^{j-1} [A_{\varphi(p)}, A_{\varphi(p+1)}]\}\|.$$

Como

$$\bigcup_{p=i}^{j-1} \{[B_p, B_{p+1}] \bigcap [A_{\varphi(p)}, A_{\varphi(p+1)}]\} \subset \{\bigcup_{p=i}^{j-1} [B_p, B_{p+1}]\} \bigcap \{\bigcup_{p=i}^{j-1} [A_{\varphi(p)}, A_{\varphi(p+1)}]\},$$

vale que,

$$\|\{\bigcup_{p=i}^{j-1} [B_p, B_{p+1}]\} \bigcap \{\bigcup_{p=i}^{j-1} [A_{\varphi(p)}, A_{\varphi(p+1)}]\}\| \geq \|\bigcup_{p=i}^{j-1} \{[B_p, B_{p+1}] \bigcap [A_{\varphi(p)}, A_{\varphi(p+1)}]\}\| =$$

$$\sum_{p=i}^{j-1} \|[B_p, B_{p+1}] \bigcap [A_{\varphi(p)}, A_{\varphi(p+1)}]\|,$$

puesto que $f(x) + g(x) \geq 0$. Por lo tanto,

$$c_\varphi(\alpha_{ij}) \geq \sum_{p=i}^{j-1} \|[B_p, B_{p+1}] \bigcap [A_{\varphi(p)}, A_{\varphi(p+1)}]\|.$$

Como φ tiene la propiedad de preservar el orden $\|[B_p, B_{p+1}] \bigcap [A_{\varphi(p)}, A_{\varphi(p+1)}]\| = c_\varphi(\alpha_{p,p+1})$; luego

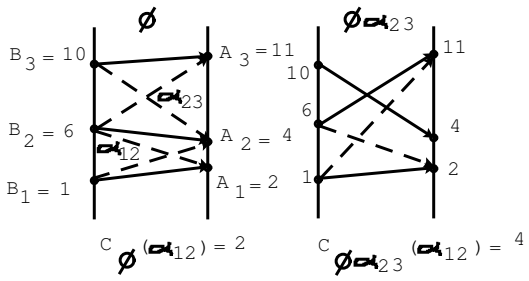
$$c_\varphi(\alpha_{ij}) \geq \sum_{p=i}^{j-1} c_\varphi(\alpha_{p,p+1}).$$

Los arcos $R_{p,p+1}$, $p = i, ..., j-1$, forman una cadena que une los nodos i y j. Si extraemos R_{ij} de τ y lo sustituimos por estos arcos, el grafo sigue siendo conexo y el costo total de los arcos es menor o igual. Removiendo arcos superfluos en el nuevo conjunto, obtenemos un nuevo árbol τ' de costo menor o igual que $c_\varphi(\tau)$ que no contiene al arco R_{ij}.

De ahora en más, cada vez que hablemos de mínimo spanning tree será de uno formado por arcos de la forma $R_{i,i+1}$.

4.5. Costo de los árboles y tours

Si comenzamos con la permutación mínima φ con costo $c(\varphi)$ y le aplicamos intercambios de un árbol mínimo τ en algún orden, obtenemos un tour ψ. Sin embargo, el costo $c(\psi)$ generalmente no es $c(\varphi) + c_\varphi(\tau)$. Esto es porque, por lo general, practicar un intercambio afecta el costo de un intercambio futuro. Por ejemplo, como podemos apreciar en la siguiente figura, si $f(x) = 1$ y $g(x) = 0$ tenemos que $c_\varphi(\alpha_{12}) = \|[1, 6] \bigcap [2, 4]\| = \|[2, 4]\| = \int_2^4 1 \, dx = 2$ pero si aplicamos α_{23} y luego α_{12} se llega a que $c_{\varphi\alpha_{23}}(\alpha_{12}) = \|[1, 6] \bigcap [2, 11]\| = \|[2, 6]\| = \int_2^6 1 \, dx = 4 \neq c_\varphi(\alpha_{12})$.



Sin embargo, hay algunos casos en donde no se producen efectos sobre los intercambios futuros. En el ejemplo anterior se puede verificar que $c_{\varphi\alpha_{12}}(\alpha_{23}) = c_\varphi(\alpha_{23})$ pues $c_{\varphi\alpha_{12}}(\alpha_{23}) = \|[6, 10] \bigcap [2, 11]\| = \|[6, 10]\| = \int_6^{10} 1 \, dx = 4$ y $c_\varphi(\alpha_{23}) = \|[6, 10] \bigcap [4, 11]\| = \|[6, 10]\| = \int_6^{10} 1 \, dx = 4$.

Definición: Un nodo i es de tipo I relativo a la permutación ψ si $B_i \leq A_{\psi(i)}$. Si en cambio, $B_i > A_{\psi(i)}$ el nodo i es de tipo II.

Definición: Una permutación ψ *preserva el orden* sobre el par (i, j) si $B_j > B_i \Rightarrow A_{\psi(j)} \geq A_{\psi(i)}$.

Lema 3: Supongamos que tenemos una permutación ψ que preserva el orden sobre (i, j) y (p, q). Sean α_{ij} y α_{pq} intercambios con $i < j$ y $p < q$. Si $\psi' = \psi\alpha_{pq}$, se tiene que $c_\psi(\alpha_{ij}) = c_{\psi'}(\alpha_{ij})$ y ψ' preserva el orden sobre (i, j) si vale cualquiera de los siguientes cuatro casos:
a) $p > j$,
b) $q < i$,
c) $p = j$ y el nodo j es de tipo I relativo a ψ ($i < j = p < q$ y $B_j \leq A_{\psi(j)}$),
d) $q = i$ y el nodo i es de tipo II relativo a ψ ($p < q = i < j$ y $B_i > A_{\psi(i)}$).

Demostración: En los casos a) y b) $\psi'(i) = \psi(i)$ y $\psi'(j) = \psi(j)$ por lo tanto el orden no cambia y los intervalos que involucra la fórmula del costo tampoco lo hacen ($c_{\psi'}(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi'(i)}, A_{\psi'(j)}]\| = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]\| = c_\psi(\alpha_{ij})$). En el caso c) tenemos que $\psi'(i) = \psi(i)$ y $\psi'(j) = \psi(q)$. El orden se preserva pues $A_{\psi'(j)} = A_{\psi(q)} \geq A_{\psi(p)} = A_{\psi(j)} \geq A_{\psi(i)} = A_{\psi'(i)}$. La fórmula del costo no se altera ya que la única diferencia entre $c_\psi(\alpha_{ij})$ y $c_{\psi'}(\alpha_{ij})$ es el reemplazo de $A_{\psi(j)}$ por $A_{\psi'(j)}$: Como se asume que j es de tipo I, $A_{\psi(j)}$ es mayor o igual que B_j por lo que al cambiarlo por un número mayor, $A_{\psi'(j)}$, no se altera el intervalo de intersección que aparece en la fórmula del costo; más formalmente,

$$c_\psi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]\| = \|[\text{máx}(B_i, A_{\psi(i)}), \text{mín}(B_j, A_{\psi(j)})]\|$$

y al ser j de tipo I, $B_j \leq A_{\psi(j)}$, vale que

$$c_\psi(\alpha_{ij}) = \|[\text{máx}(B_i, A_{\psi(i)}), B_j]\|.$$

Por otro lado,

$$c_{\psi'}(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi'(i)}, A_{\psi'(j)}]\| = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(q)}]\|$$

y como $B_j \leq A_{\psi(j)} \leq A_{\psi(q)}$, resulta

$$c_{\psi'}(\alpha_{ij}) = \|[\text{máx}(B_i, A_{\psi(i)}), B_j]\|.$$

El caso d) es similar al c). Tenemos que $\psi'(j) = \psi(j)$ y $\psi'(i) = \psi(p)$. Se preserva el orden porque $A_{\psi'(i)} = A_{\psi(p)} \leq A_{\psi(q)} = A_{\psi(i)} \leq A_{\psi(j)} = A_{\psi'(j)}$. Nuevamente, el único cambio es el reemplazo de $A_{\psi(i)}$ por $A_{\psi'(i)} = A_{\psi(p)}$. Como $A_{\psi'(i)} \leq A_{\psi(i)}$ y $A_{\psi(i)} < B_i$ pues i es de tipo II, la intersección no se modifica: en efecto, observemos que

$$c_\psi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]\| = \|[\text{máx}(B_i, A_{\psi(i)}), \text{mín}(B_j, A_{\psi(j)})]\|$$

y al ser i de tipo II, $B_i > A_{\psi(i)}$, vale que

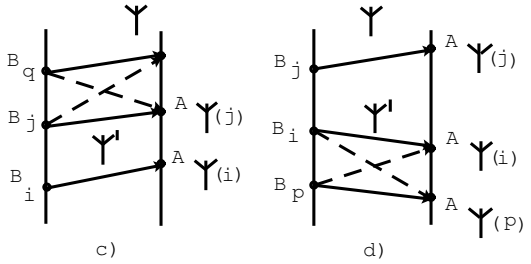
$$c_\psi(\alpha_{ij}) = \|[B_i, \text{mín}(B_j, A_{\psi(j)})]\|$$

y,

$$c_{\psi'}(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi'(i)}, A_{\psi'(j)}]\| = \|[B_i, B_j] \bigcap [A_{\psi(p)}, A_{\psi(j)}]\|$$

y como $B_i > A_{\psi(i)} \geq A_{\psi(p)}$, tenemos que

$$c_{\psi'}(\alpha_{ij}) = \|[B_i, \text{mín}(B_j, A_{\psi(j)})]\|.$$



Lema 4: En el caso c) del lema 3 el nodo j es de tipo I relativo a ψ' y en el caso d) el nodo i es de tipo II relativo a ψ'.

Demostración: En c) probamos que $A_{\psi'(j)} \geq A_{\psi(j)} \geq B_j$ por ser j de tipo I relativo a ψ y en d) vimos que $A_{\psi'(i)} \leq A_{\psi(i)} < B_i$ pues i es de tipo II relativo a ψ.

Lema 5: Sea φ la permutación de costo mínimo, y sean $\alpha_{i_1 i_1+1}, \alpha_{i_2 i_2+1}, ..., \alpha_{i_m i_m+1}$ una serie de intercambios con $i_1 < i_2 < ... < i_m$. Luego, existe una permutación ψ', obtenida al ejecutar los $\alpha_{i_p i_p+1}$ $(1 \leq p \leq m)$ en un orden particular, que cumple que

$$c(\psi') = c(\varphi) + \textstyle\sum_{p=1}^{m} c_\varphi(\alpha_{i_p i_{p+1}}).$$

Demostración: Llamaremos a un intercambio, un intercambio de tipo I si su menor nodo es de tipo I, y se denominará un intercambio de tipo II si su menor nodo es de tipo II. El orden de ejecución de los intercambios es el siguiente. Primero, aplicaremos todos los intercambios de tipo I en orden de índice decreciente. Luego, todos los intercambios de tipo II en orden creciente de índice.

Consideremos un intercambio de tipo I, α_{pp+1}, encontrado en la primera parte. Pensemos que cumple el rol del α_{pq} del lema 3. Consideremos los intercambios restantes no ejecutados cuyo costo puede ser afectado por ejecutar α_{pp+1}. Si un intercambio de los que resta no tiene nodos en común con α_{pp+1}, estamos en los casos a) o b) del lema 3. Los intercambios con nodos en común con α_{pp+1} son α_{p+1p+2} y α_{p-1p}. Si α_{p+1p+2} no fue realizado, debe ser de tipo II (porque los de tipo I se realizan en orden decreciente de índice), y estamos en el caso d) del lema. Si α_{p-1p} no fue realizado, como α_{pp+1} es de tipo I, estamos en el caso c). Consecuentemente, durante la primera parte, el costo de los intercambios restantes no es afectado. Es correcto utilizar el lema 3 sucesivamente pues los intercambios con los que se comienza cumplen que $B_j > B_i \Rightarrow A_{\varphi(j)} \geq A_{\varphi(i)}$ y

el lema 3 demuestra que esta propiedad sigue valiendo tras realizar cada intercambio, mientras que el lema 4 demuestra que los tipos de los intercambios no ejecutados siguen siendo lo que eran al principio.

El razonamiento con respecto a los intercambios de tipo II es similar. Nuevamente, si uno de los intercambios restantes no tiene nodos en común con el intercambio α_{pp+1} podemos aplicar el caso b) del lema 3, ya que, el intercambio debe tener primer índice mayor que $p+1$. Similarmente, el único intercambio restante con un nodo en común puede ser α_{p+1p+2} y debe ser de tipo II. Luego, recurriríamos al caso d).

Teorema 3: Sea τ un spanning tree de mínimo costo de G_φ. Sean $\alpha^1_{i_1 i_1+1}, ..., \alpha^1_{i_l i_l+1}$ los intercambios de tipo I correspondientes a los arcos de τ con $i_1 > i_2 > ... > i_l$ y sean $\alpha^2_{j_1 j_1+1}, ..., \alpha^2_{j_m j_m+1}$ los intercambios de tipo II con $j_1 < j_2 < ... < j_m$. Si

$$\psi^* = \varphi\alpha^1_{i_1 i_1+1}...\alpha^1_{i_l i_l+1}\alpha^2_{j_1 j_1+1}...\alpha^2_{j_m j_m+1},$$

entonces ψ^* es un tour con costo

$$c(\psi^*) = c(\varphi) + c_\varphi(\tau).$$

Demostración: El hecho de que ψ^* sea un tour se deriva del teorema 2 y la afirmación sobre el costo se desprende de los lemas 2 y 4. El orden seguido para ejecutar los intercambios es el mismo que se especifica en la demostración del lema 5.

La permutación ψ^* es candidata a ser el tour de mínimo costo. En las próximas dos secciones lo probaremos.

4.6. Una subestimación para el costo de una permutación

Podemos calcular el costo de tener $\psi(i)$ a continuación de i de la siguiente forma:

$$c_{i\psi(i)} = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}]|_f + |(-\infty, B_i] \bigcap [A_{\psi(i)}, +\infty)|_g.$$

Definimos el intervalo P_i, independiente de ψ, como

$$P_i = [B_i, B_{i+1}] \bigcap [A_{\varphi(i)}, A_{\varphi(i+1)}]$$

(recordemos que φ es la permutación que ranquea los A) y el conjunto P, unión de intervalos disjuntos, como

$$P = \bigcup_{i=1}^{N-1} P_i.$$

Usando P, definimos un nuevo costo c^*_{ij}

$$c^*_{ij} = |[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap P|_f + |(-\infty, B_i] \bigcap [A_{\psi(i)}, +\infty) \bigcap P|_g$$

que es la misma fórmula anterior excepto por la presencia de P. Usando el nuevo costo podemos definir un nuevo costo $c^*(\psi)$ para la permutación ψ como

$$c^*(\psi) = \sum_{i=1}^{N} c^*_{i\psi(i)},$$

y para intercambios

$$c^*_\psi(\alpha_{ij}) = c^*(\psi\alpha_{ij}) - c^*(\psi).$$

La fórmula para $c^*_\psi(\alpha_{ij})$ se obtiene en la misma forma que $c_\psi(\alpha_{ij})$,

$$c^*_\psi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}] \bigcap P\|,$$

siempre que ψ preserve el orden sobre el par (i, j), la única diferencia es la presencia de P. Observar que, en este caso, $c^*_\psi(\alpha_{ij}) \leq c_\psi(\alpha_{ij})$.

Nos referiremos a la contribución al costo $c^*_{i\psi(i)}$ realizada por un intervalo P_q, definiendo $c^*_{ij}(q)$ por

$$c^*_{ij}(q) = |[B_i, +\infty) \bigcap (-\infty, A_j] \bigcap P_q|_f + |(-\infty, B_i] \bigcap [A_j, +\infty) \bigcap P_q|_g, \tag{4.3}$$

que es similar a c^*_{ij}, salvo que a P se lo reemplaza por P_q. Claramente,

$$\sum_{q=1}^{N-1} c^*_{i\psi(i)}(q) = c^*_{i\psi(i)}.$$

Lema 6: Para cualquier i, j y q, cada vez que los intervalos

$$[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap P_q,$$

$$[A_{\psi(j)}, +\infty) \bigcap (-\infty, B_j] \bigcap P_q,$$

tienen interior no vacío, son P_q. Si P_q tiene interior no vacío, las intersecciones anteriores son iguales a P_q si y sólo si valen

- *a)* $i \leq q < \varphi^{-1}\psi(i)$,
- *b)* $\varphi^{-1}\psi(j) \leq q < j$

respectivamente.

Demostración: Si $[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap [B_q, B_{q+1}] \bigcap [A_{\varphi(q)}, A_{\varphi(q+1)}]$ tiene interior no vacío entonces $[B_i, +\infty) \bigcap [B_q, B_{q+1}]$ tiene interior no vacío y $(-\infty, A_{\psi(i)}] \bigcap [A_{\varphi(q)}, A_{\varphi(q+1)}]$ tiene interior no vacío, con lo cual $q + 1 > i$ y $\varphi(q) < \psi(i)$. Entonces $q \geq i$ y $q < \varphi^{-1}(\psi(i)) \Rightarrow q \geq i$ y $q + 1 \leq \varphi^{-1}(\psi(i))$. Por lo tanto, vale que $[B_i, +\infty) \bigcap [B_q, B_{q+1}] = [B_q, B_{q+1}]$ y que $(-\infty, A_{\psi(i)}] \bigcap [A_{\varphi(q)}, A_{\varphi(q+1)}] = [A_{\varphi(q)}, A_{\varphi(q+1)}]$.

Con lo que se concluye que si $[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap P_q$ tiene interior no vacío, entonces $[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap P_q = P_q$.

$\Leftarrow$) Si $i \leq q < \varphi^{-1}(\psi(i))$ entonces $i \leq q < q+1 \leq \varphi^{-1}(\psi(i))$, con lo cual $B_i \leq B_q \leq B_{q+1}$ y $A_{\varphi(q)} \leq A_{\varphi(q+1)} \leq A_{\psi(i)}$. Por lo tanto, $[B_i, +\infty) \bigcap [B_q, B_{q+1}] = [B_q, B_{q+1}]$ y $(-\infty, A_{\psi(i)}] \bigcap [A_{\varphi(q)}, A_{\varphi(q+1)}] = [A_{\varphi(q)}, A_{\varphi(q+1)}] \Rightarrow [B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap P_q = P_q$

$\Rightarrow$) Supongamos que $i > q$ o que $q \geq \varphi^{-1}(\psi(i)) \Rightarrow i > q,\ i \geq q+1$ o $q \geq \varphi^{-1}(\psi(i))$ $\Rightarrow B_i \geq B_q,\ B_i \geq B_{q+1}$ o $A_{\varphi(q)} \geq A_{\psi(i)}$. En consecuencia $[B_i, +\infty) \bigcap [B_q, B_{q+1}] = \emptyset$ o $\{B_{q+1}\}$ o $(-\infty, A_{\psi(i)}] \bigcap [A_{\varphi(q)}, A_{\varphi(q+1)}] = \emptyset$ o $\{A_{\varphi(q)}\}$, de donde $[B_i, +\infty) \bigcap (-\infty, A_{\psi(i)}] \bigcap P_q$ tiene interior vacío. Lo que resulta absurdo por la hipótesis.

El mismo argumento puede usarse para demostrar el segundo caso.

Ahora sí podemos probar el teorema que nos da la subestimación.

Teorema 4: Para cualquier permutación ψ,

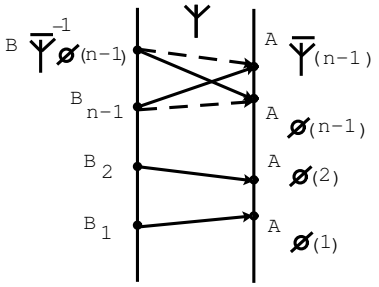
$$c(\psi) \geq c(\varphi) + c^*(\psi).$$

Demostración: En un principio, vamos a establecer el teorema 4 para el caso $\psi = \varphi$, demostrando que $c^*(\varphi) = 0$.

Aplicando el lema para el caso $\psi = \varphi$, como no se satisfacen las desigualdades, teniendo en cuenta la ecuación (4.3) vale que $c^*_{i\varphi(i)}(q) = 0$ para todo i y q, por lo tanto, $c^*(\varphi) = 0$.

Definimos la altura h de una permutación ψ como el primer índice en donde $\psi(i)$ difiere de $\varphi(i)$ si $\psi \neq \varphi$, o bien $h = N$ si $\psi = \varphi$. Si $h = N$, $\psi = \varphi$ y el teorema vale. Asumamos que el teorema vale para todas las permutaciones de altura h, $h \geq n$. Probaremos que es cierto para todas las permutaciones de altura $n-1$.

Sea $\overline{\psi}$ con altura $n-1$ y sea ψ definida por $\psi = \overline{\psi}\alpha_{ij}$, con $i = n-1$, $j = \overline{\psi}^{-1}\varphi(n-1)$.



Entonces

$$\psi(n-1) = \overline{\psi}[\overline{\psi}^{-1}\varphi(n-1)] = \varphi(n-1);$$

por lo tanto, ψ es de altura mayor o igual que n y el teorema vale para ψ. La permutación $\overline{\psi}$ se puede obtener de ψ a través del mismo intercambio α_{ij}. Como $i < j$ (puesto que $j = \overline{\psi}^{-1}\varphi(n-1)$ y por hipótesis $\overline{\psi}(1) = \varphi(1)$, ..., $\overline{\psi}(n-2) = \varphi(n-2)$, $\overline{\psi}(n-1) \neq \varphi(n-1)$ con lo cual $\overline{\psi}(k) \neq \varphi(n-1)\ \forall k \leq n-1$) y

$$\varphi^{-1}\psi(i) = \varphi^{-1}(\psi(n-1)) = \varphi^{-1}(\varphi(n-1)) = n-1 < \varphi^{-1}\overline{\psi}(n-1) = \varphi^{-1}\psi(j)$$

Entonces $A_{\psi(i)} < A_{\psi(j)}$, es decir, ψ preserva el orden del par (i,j) y es por esto que las fórmulas para calcular $c_\psi(\alpha_{ij})$ y $c^*_\psi(\alpha_{ij})$ son: $c_\psi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}]\|$ y $c^*_\psi(\alpha_{ij}) = \|[B_i, B_j] \bigcap [A_{\psi(i)}, A_{\psi(j)}] \bigcap P\|$.

Por hipótesis inductiva

$$c(\psi) \geq c(\varphi) + c^*(\psi);$$

como además vale que

$$c_\psi(\alpha_{ij}) \geq c^*_\psi(\alpha_{ij}),$$

sumando encontramos que

$$c(\overline{\psi}) = c(\psi) + c_\psi(\alpha_{ij}) \geq c(\varphi) + c^*(\psi) + c^*_\psi(\alpha_{ij}) = c(\varphi) + c^*(\overline{\psi}),$$

con lo que queda demostrado el teorema.

4.7. Una subestimación para el costo de un tour

Haremos una conexión entre la subestimación para el costo de una permutación y el costo del mínimo spanning tree.

Definamos el grafo de N nodos, G^*_ψ que contiene todos los arcos $R_{q\varphi(q)}$ de G_φ y los arcos no dirigidos R_{qq+1} para cada q tal que existe algún i, tal que $i \leq q < \varphi^{-1}\psi(i)$ o $\varphi^{-1}\psi(i) \leq q < i$.

En el siguiente lema se concluye que G^*_ψ contiene un spanning tree.

Lema 7: Si ψ es un tour, G^*_ψ es conexo.

Demostración: Supongamos que G^*_ψ no es conexo. Entonces es posible dividirlo en dos componentes disjuntas C_1 y C_2. Como G^*_ψ incluye todos los arcos de G_φ, C_1 y C_2 pueden ser sólo uniones de componentes de G_φ. Si $\overline{C_1}$ es el conjunto de nodos en C_1

$$\varphi\overline{C_1} = \overline{C_1}.$$

Supongamos que

$$\varphi^{-1}\psi\overline{C_1} = \overline{C_1}.$$

Entonces si aplicamos φ, obtenemos

$$\psi\overline{C_1} = \varphi\overline{C_1} = \overline{C_1},$$

lo que implicaría que ψ no es un tour; se concluye que no puede valer $\varphi^{-1}\psi\overline{C_1} = \overline{C_1}$.

Por lo tanto, debe existir al menos un i_0 tal que $i_0 \in C_1$ y el nodo $\varphi^{-1}\psi(i_0) \in C_2$. Si $i_0 < \varphi^{-1}\psi(i_0)$, sea q el menor índice para el cual

$$i_0 \leq q < q+1 \leq \varphi^{-1}\psi(i_0),$$

y el nodo $q+1 \in C_2$. Por definición, el arco $R_{qq+1} \in G^*_\psi$. Si $\varphi^{-1}\psi(i_0) < i_0$, sea q el mayor índice para el cual $\varphi^{-1}\psi(i_0) \leq q < q+1 \leq i_0$ y el nodo $q \in C_2$. Nuevamente, por definición, el arco $R_{qq+1} \in G^*_\psi$. Pero R_{qq+1} une las dos componentes C_1 y C_2 que supuestamente son disjuntas. Esta contradicción establece el lema.

Antes de probar la minimalidad del tour ψ^* definido en el teorema 3, necesitamos un lema más.

Lema 8: Dados q y una permutación ψ, la cantidad de valores para los cuales vale $i \leq q < \varphi^{-1}\psi(i)$ es la misma que la cantidad de valores para los cuales vale $\varphi^{-1}\psi(i) \leq q < i$.

Demostración: Definamos la función $\overline{R}(i)$ por

$$\overline{R}(i) = \begin{cases} 1 & \text{si } i > q \\ 0 & \text{si } i \leq q \end{cases}$$

Para cualquier permutación ψ, se tiene que

$$\sum_i \{\overline{R}(i) - \overline{R}[\varphi^{-1}\psi(i)]\} = 0,$$

pues $\varphi^{-1}\psi$ reordena los términos. El término individual

$$\{\overline{R}(i) - \overline{R}[\varphi^{-1}\psi(i)]\}$$

es 1 si $\varphi^{-1}\psi(i) \leq q < i$, -1 si $i \leq q < \varphi^{-1}\psi(i)$ o 0 en otro caso. Pero entonces la cantidad de 1 y -1 debe ser igual, lo que demuestra el lema.

Ahora sí podemos demostrar la minimalidad de ψ^*.

Teorema 5: El tour ψ^* descripto en el teorema 3 es un tour de mínimo costo.

Demostración: Consideremos la suma $c^*(\psi)$, que puede ser escrita:

$$c^*(\psi) = \sum_i c^*_{i\psi(i)} = \sum_i (\sum_q c^*_{i\psi(i)}(q)) = \sum_q (\sum_i c^*_{i\psi(i)}(q)).$$

Observemos al término que se encuentra entre paréntesis:

$$\sum_i c^*_{i\psi(i)}(q).$$

Por el lema 6, si $c^*_{i\psi(i)}(q)$ no es cero, entonces tiene que ser $|P_q|_f$ o $|P_q|_g$, la primera cuando vale $i \leq q < \varphi^{-1}\psi(i)$ y la segunda cuando vale $\varphi^{-1}\psi(i) \leq q < i$. Por el lema 8, podemos concluir que $|P_q|_f$ y $|P_q|_g$ ocurren con igual frecuencia en $\sum_i c^*_{i\psi(i)}(q)$; por lo tanto, la suma es siempre un múltiplo entero no negativo de $\|P_q\|$. Más aún, si R_{qq+1}

es un arco de G^*_ψ, necesariamente valen $i \leq q < \varphi^{-1}\psi(i)$ o $\varphi^{-1}\psi(j) \leq q < j$ para algún i o j; por el lema 8, valen ambas desigualdades para algún i y j y por consiguiente, del lema 6 se puede concluir que

$$\sum_i c^*_{i\psi(i)}(q) \geq \|P_q\|.$$

Entonces

$$c^*(\psi) \geq \sum_{q|R_{qq+1}\in G^*_\psi} \|P_q\|,$$

y como $\|P_q\| = c_\varphi(\alpha_{qq+1})$,

$$c^*(\psi) \geq \sum_{q|R_{qq+1}\in G^*_\psi} c_\varphi(\alpha_{qq+1}).$$

Como G^*_ψ es conexo, incluye algún árbol τ'. Por lo tanto,

$$c^*(\psi) \geq \sum_{q|R_{qq+1}\in G^*_\psi} c_\varphi(\alpha_{qq+1}) \geq \sum_{q|R_{qq+1}\in \tau'} c_\varphi(\alpha_{qq+1}) \geq c_\varphi(\tau').$$

Si τ es cualquier mínimo spanning tree tenemos que

$$c^*(\psi) \geq c_\varphi(\tau).$$

Del teorema 4, la desigualdad anterior y el teorema 3, concluimos que

$$c(\psi) \geq_{Teo\ 4} c(\varphi) + c^*(\psi) \geq c(\varphi) + c_\varphi(\tau) =_{Teo\ 3} c(\psi^*),$$

donde ψ^* es el candidato para el tour óptimo definido en el teorema 3.

4.8. El algoritmo

Descripción del algoritmo:

Comenzaremos dando una descripción de los pasos computacionales que se requieren en el algoritmo para la búsqueda del tour mínimo.

Los datos del problema son las funciones $f(x)$ y $g(x)$ que proporcionan el costo de aumentar y disminuir el estado, y la lista de N trabajos con sus respectivos valores de estado inicial, A_i y final B_i.

PASOS PRELIMINARES:

- P1) Ordenar los números B_i por tamaño en orden creciente y renumerar los trabajos de forma tal que con la nueva numeración

$$B_i \leq B_{i+1},\ i = 1, ..., N-1.$$

- P2) Ordenar los A_i por tamaño en orden creciente.
- P3) Encontrar $\varphi(p)$ para todo p. Definir la permutación φ por

  $$\varphi(p) = q,$$

  si q es tal que A_q es el p-ésimo más pequeño de los A_i.
- P4) Calcular los números $c_\varphi(\alpha_{ii+1})$ para $i = 1, ..., N-1$:
  $c_\varphi(\alpha_{ii+1}) = 0$ si $\text{máx}(B_i, A_{\varphi(i)}) \geq \text{mín}(B_{i+1}, A_{\varphi(i+1)})$,

  $$c_\varphi(\alpha_{ii+1}) = \int_{\text{máx}(B_i, A_{\varphi(i)})}^{\text{mín}(B_{i+1}, A_{\varphi(i+1)})} (f(x) + g(x))\ dx \text{ si } \text{máx}(B_i, A_{\varphi(i)}) < \text{mín}(B_{i+1}, A_{\varphi(i+1)}).$$

PASOS PARA SELECCIONAR UN CONJUNTO DE ARCOS:

- S1) Formar un grafo no dirigido con N nodos y arcos que conectan los nodos i y $\varphi(i)$, para $i = 1, ..., N$.
- S2) Si el grafo tiene una sola componente conexa, ir al paso T1). De lo contrario, seleccionar el menor valor $c_\varphi(\alpha_{ii+1})$ tal que i está en una componente e $i+1$ en otra. De existir dos iguales, optar por cualquiera de ellos.
- S3) Agregar el arco no dirigido R_{ii+1} al grafo usando el valor i seleccionado en S2). Volver a S2).
- T1) Dividir los arcos agregados en S3) en dos grupos. Aquellos R_{ii+1} para los cuales $A_{\varphi(i)} \geq B_i$ (relacionados con los intercambios de tipo I) los ponemos en el grupo I, y los que cumplen que $B_i > A_{\varphi(i)}$ (relacionados con los intercambios de tipo II) los ubicamos en el grupo II.
- T2) Hallar el mayor índice i_1 tal que $R_{i_1 i_1+1}$ esté en el grupo I. Luego buscar el que le sigue, i_2 y continuar así sucesivamente. Supongamos que hay l elementos en el grupo I.
- T3) Hallar el menor índice j_1 tal que $R_{j_1 j_1+1}$ esté en el grupo II. Luego buscar el que le sigue, j_2 y continuar así sucesivamente. Supongamos que hay m elementos en el grupo II.
- T4) El tour mínimo se obtiene mediante la permutación ψ^* definida por

  $$\psi^*(i) = \varphi\alpha_{i_1 i_1+1}\alpha_{i_2 i_2+2}...\alpha_{i_l i_l+1}\alpha_{j_1 j_1+1}\alpha_{j_2 j_2+2}...\alpha_{j_m j_m+1}(i),$$

  donde α_{pq} es la permutación definida por

$$\begin{cases} \alpha_{pq}(p) = q, \\ \alpha_{pq}(q) = p, \\ \alpha_{pq}(i) = i,\ i \neq p, q. \end{cases}$$

Este algoritmo resuelve el problema en tiempo polinomial para este caso particular del TSP.

4.9. Ejemplo numérico

Sea $f(x) = 1$ y $g(x) = 0$. Analicemos la situación en donde existen siete trabajos con los A 's y los B 's dados en la siguiente tabla:

B	A
31	34
19	45
3	4
40	18
26	22
15	16
1	7

P1) Ordenamos por tamaño los B_i y los numeramos.

N °	B
7	40
6	31
5	26
4	19
3	15
2	3
1	1

P2) Ordenamos por tamaño los A_i y guardamos en la columna de al lado el número del trabajo al cual corresponde

A	N °
45	4
34	6
22	5
18	7
16	3
7	1
4	2

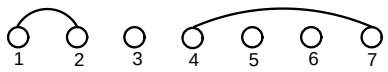
P3) Obtenemos φ:

$$\begin{array}{l}\varphi(1)=2,\\ \varphi(2)=1,\\ \varphi(3)=3,\\ \varphi(4)=7,\\ \varphi(5)=5,\\ \varphi(6)=6,\\ \varphi(7)=4.\end{array}$$

P4) Calculamos $c_\varphi(\alpha_{ii+1})$ para $i=1,...,6$. En principio, hallamos:

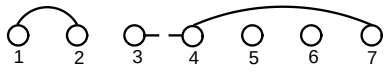
- $\text{máx}(B_1, A_{\varphi(1)}) = \text{máx}(1,4) = 4 \geq \text{mín}(B_2, A_{\varphi(2)}) = \text{mín}(3,7) = 3 \Rightarrow c_\varphi(\alpha_{12}) = 0$,
- $\text{máx}(B_2, A_{\varphi(2)}) = \text{máx}(3,7) = 7 \leq \text{mín}(B_3, A_{\varphi(3)}) = \text{mín}(15,16) = 15 \Rightarrow c_\varphi(\alpha_{23}) = \int_7^{15} 1\ dx = 8$,
- $\text{máx}(B_3, A_{\varphi(3)}) = \text{máx}(15,16) = 16 \leq \text{mín}(B_4, A_{\varphi(4)}) = \text{mín}(19,18) = 18 \Rightarrow c_\varphi(\alpha_{34}) = \int_{16}^{18} 1\ dx = 2$,
- $\text{máx}(B_4, A_{\varphi(4)}) = \text{máx}(19,18) = 19 \leq \text{mín}(B_5, A_{\varphi(5)}) = \text{mín}(26,22) = 22 \Rightarrow c_\varphi(\alpha_{45}) = \int_{19}^{22} 1\ dx = 3$,
- $\text{máx}(B_5, A_{\varphi(5)}) = \text{máx}(26,22) = 26 \leq \text{mín}(B_6, A_{\varphi(6)}) = \text{mín}(31,34) = 31 \Rightarrow c_\varphi(\alpha_{56}) = \int_{26}^{31} 1\ dx = 5$,
- $\text{máx}(B_6, A_{\varphi(6)}) = \text{máx}(31,34) = 34 \leq \text{mín}(B_7, A_{\varphi(7)}) = \text{mín}(40,45) = 40 \Rightarrow c_\varphi(\alpha_{67}) = \int_{34}^{40} 1\ dx = 6$,

S1) Armamos el grafo no dirigido G_φ (en el gráfico no se incluyen los arcos de tipo R_{ii})

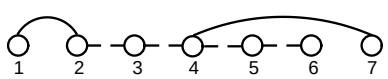


S2) El menor $c_\varphi(\alpha_{ii+1})$ tal que i esté en una componente e $i+1$ en otra es $c_\varphi(\alpha_{34})$

S3) Agregamos el arco R_{34} al grafo anterior



S2) y S3) De la misma forma continuamos agregando R_{45}, R_{56} y R_{23} y como ahora el grafo tiene una única componente vamos a T1).



T1) Dividimos los arcos agregados en los dos grupos: Grupo I= $\{R_{23}, R_{34}\}$ y Grupo II= $\{R_{45}, R_{56}\}$, pues $A_{\varphi(2)} = 7 \geq B_2 = 3$, $A_{\varphi(3)} = 16 \geq B_3 = 15$, $A_{\varphi(4)} = 18 < B_4 = 19$ y $A_{\varphi(5)} = 22 < B_5 = 26$.

T2) $i_1 = 3$, $i_2 = 2$.

T3) $j_1 = 4$, $j_2 = 5$.

T4) El tour mínimo es dado por la permutación

$$\psi^* = \varphi\alpha_{3,4}\alpha_{2,3}\alpha_{4,5}\alpha_{5,6}$$

$$\begin{gathered}\psi^*(1) = 2,\\ \psi^*(2) = 7,\\ \psi^*(3) = 1,\\ \psi^*(4) = 5,\\ \psi^*(5) = 6,\\ \psi^*(6) = 3,\\ \psi^*(7) = 4.\end{gathered}$$

Como $\psi^*(i)$ arroja el sucesor del trabajo i, el tour de mínimo costo es

$$1 \to 2 \to 7 \to 4 \to 5 \to 6 \to 3 \to 1.$$

Su costo es 34. Puede calcularse directamente ($c(\psi^*) = \sum_{i=1}^{7} c_{i\psi^*(i)} = c_{12} + c_{27} + c_{31} + c_{45} + c_{56} + c_{63} + c_{74} = 3+15+0+3+8+0+5 = 34$) o sumando el costo de φ y los costos de los intercambios ($c(\psi^*) = c(\varphi) + c_\varphi(\alpha_{34}) + c_\varphi(\alpha_{23}) + c_\varphi(\alpha_{45}) + c_\varphi(\alpha_{56}) = \sum_{i=1}^{7} c_{i\varphi(i)} + 2 + 8+3+5 = c_{12}+c_{21}+c_{33}+c_{47}+c_{55}+c_{66}+c_{74}+18 = 3+4+1+0+0+3+5+18 = 16+18 = 34$).

Bibliografía

[1] Cook, S.A.
The complexity of theorem-proving procedures.
Kibern. Sb., Nov. Ser. 12, 5-15 (1975).

[2] Garey, M.R.; Johnson D.S.
Computers and Intractability: "A Guide to the Theory of NP-Completeness".
W. H. Freeman and Company, 1979.

[3] Hurkens, C. A. J.; Woeginger, G. J.
On the nearest neighbor rule for the traveling salesman problem.
Oper. Res. Lett. 32, 1-4, 2004.

[4] Aho A. V.; Hopcroft J. E.; Ullman J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley Publishing Company, 1974.

[5] Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G.; Shmoys, D. B.
The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.
John Wiley and Sons, 1985.

[6] Lin, S.; Kernighan, B.W.
An effective heuristic algorithm for the traveling salesman problem.
[J] Oper. Res., Vol.21, No.2, pp. 498-516, 1973.

[7] Croes, G. A.
A Method for Solving Traveling Salesman Problems.
Oper. Res., Vol.6, No.6, pp. 791-812, 1958.

[8] Lin, S.
Computer Solutions of the Traveling Salesman Problem.
Bell Systems Technology Journal, Vol.44, pp. 2245-2269, 1965.

[9] Helsgaun, Keld

An effective implementation of the Lin-Kernighan traveling salesman heuristic.

Eur. J. Oper. Res. 126, No.1, 106-130 (2000).

[10] Gilmore, P.; Gomory, R.

Sequencing a one state-variable machine: a solvable case of the traveling salesman problem.

[J] Oper. Res. 12, 655-679 (1964).