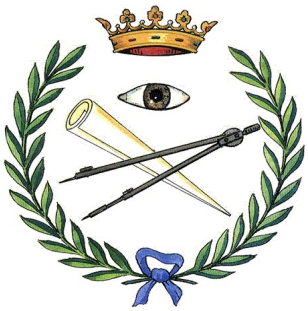


Universidad de Santiago de Compostela

Máster en Técnicas Estadísticas

Trabajo Fin de Máster

El problema del viajante

Francisco José Veiga Losada

Director: Julio González Díaz

Santiago de Compostela - 2 de septiembre de 2013

Índice general

Introducción

En este trabajo abordamos el problema del viajante o *traveling salesman problem* (TSP), un problema clásico de optimización combinatoria.

La importancia del problema del viajante radica en que diversos problemas del mundo real que, aparentemente no tienen relación entre sí, pueden ser formulados como un caso particular del mismo. Por ejemplo, puede aplicarse: en áreas de logística de transporte, en robótica y en control y operación optimizada de semáforos.

Este trabajo está organizado en cuatro partes:

En la primera parte, enunciamos los principales conceptos teóricos que utilizaremos para desarrollar el trabajo. Veremos también la formulación matemática del problema del viajante de comercio así como unas breves notas de complejidad computacional.

En la segunda parte demostraremos, basándonos principalmente en el libro de Papamitrou (véase bibliografía), que la versión de reconocimiento del problema del viajante de comercio es NP-completo. Para la consecución de este objetivo hemos de demostrar otros resultados muy importantes como puede ser el *Teorema de Cook*.

En la tercera parte, proponemos y definimos una serie de algoritmos, heurísticos y exactos, con los que podemos solucionar algunos casos particulares del problema del viajante. Por ejemplo el algoritmo del árbol o el algoritmo de las hormigas.

Finalmente, comprobaremos el funcionamiento de los algoritmos comparando el tiempo de ejecución y sus soluciones, teniendo en cuenta que algunos de ellos son heurísticos, por lo que la solución no tiene porque ser exacta, todo ello a través de ejemplos aleatorios.

Una vez ejecutados todos los ejemplos, observando los resultados obtenidos, podemos analizarlos de forma breve y los representaremos mediante una tabla.

Capítulo 1
Bases Teóricas

Contenidos

En esta sección enunciaremos el problema del viajante de comercio de un modo más formal y veremos distintas maneras de plantearlo matemáticamente. Analizaremos también los conceptos teóricos básicos que utilizaremos posteriormente para demostrar que el problema del viajante de comercio (TSP) es un problema que pertenece a la clase de complejidad computacional NP-completo. En capítulos posteriores del trabajo, trataremos casos particulares del TSP, para los que hallaremos su solución utilizando diferentes algoritmos, los cuales también programaremos en un computador.

1.1. El problema del viajante: Definición y variaciones.

"Un viajante de comercio tiene que visitar una serie de ciudades pasando solo una vez por cada una, partiendo de su ciudad de origen y volviendo a esta. ¿Cuál es la ruta óptima que debe elegir?", esto sería, de modo informal, una posible descripción del *problema del viajante de comercio*, conocido como TSP que son sus siglas en inglés (Traveling Salesman Problem).

Este es uno de los problemas de mayor importancia y más estudiados en el desarrollo del diseño de algoritmos y de la teoría de complejidad computacional. Aunque sabemos que tiene solución (ya que siempre podemos evaluar todas las posibles rutas) la resolución es verdaderamente complicada a pesar de que en su definición parezca muy sencillo. Como veremos más adelante, si buscamos la solución óptima para este problema, la dificultad reside en que este pertenece a la clase de complejidad NP-duro, no es ni NP-completo, esto quiere decir que no existen algoritmos para solucionarlo en tiempo polinomial ni tampoco tenemos algoritmos que nos permitan verificar su solución en tiempo polinomial. Otro motivo por el que el TSP es muy estudiado es porque está presente en muchos otros problemas más generales que se encuentran en la práctica de forma habitual, por ejemplo en los problemas de enrutamiento, por lo que si se encontrara una solución para el TSP obtendríamos un progreso significativo en el estudio de estos otros problemas.

Un ejemplo práctico del problema del viajante en un caso real, recopilado del documento [5], se desarrolló en Atlanta (USA) en 1982 por *Bartholdi* y *Platzman*, en donde el modelo del problema del viajante se aplicó para proveer de comida caliente a personas mayores o enfermas que no podían salir de casa para hacer la compra o no podían cocinar. Como la comida debía repartirse utilizando una única camioneta de reparto, se utilizó este modelo para generar diariamente la ruta que debía seguir la camioneta, tratando que esta fuera siempre lo más corta posible. Además el sistema para calcular la ruta debía de ser barato, eficiente y su coste de mantenimiento debía ser mínimo ya que era para una institución benéfica que no tenía prácticamente ningún ingreso. Al sistema también se le podían añadir nuevos domicilios sin que esto provocara ninguna alteración en la gestión del mismo.

Una definición bastante genérica y nada matemática pero a mi parecer bastante clara, que podemos encontrar en el libro de Davendra [1], es la siguiente:

Definición 1.1. Dado un conjunto de ciudades y el coste (o distancia) de viajar entre cada par posible de estas ciudades, el *problema del viajante de comercio* consiste en encontrar el mejor camino posible que visite todas las ciudades una única vez y vuelva al punto de partida minimizando el coste del viaje.

En esta definición, Davendra está suponiendo que estamos en un grafo completo, ya que dice que todas las ciudades pueden conectarse entre si. La definición de grafo completo la daremos posteriormente.

Previamente a dar una definición más formal de este problema, veremos unos cuantos conceptos:

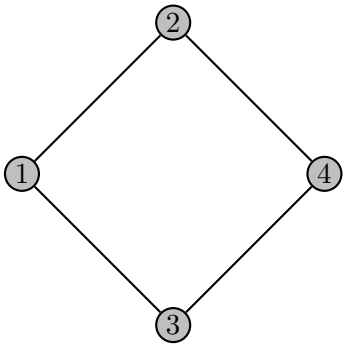
- Un *grafo* G es un par (N, M) consistente en un conjunto N de elementos llamados nodos o vértices y un conjunto M cuyos elementos llamaremos arcos o aristas. Cada arco o arista representa una conexión directa entre dos nodos o elementos de N.
- Un grafo *orientado* es aquel en el que los arcos son pares ordenados; el arco (i, j) empieza en el nodo i y termina en el nodo j. En este caso $M \subseteq N \times N$ y, si $i \neq j$, $(i, j) \neq (j, i)$.
- Un grafo *no orientado*, es un grafo en el que (i, j) y (j, i) representan el mismo arco.
- Un grafo *completo* es un grafo (orientado o no) en el que todas las aristas posibles están presentes.
- Un *camino* es una secuencia de aristas en las que todos los vértices son distintos.
- Un *circuito* es una secuencia de aristas en la que el primer vértice y el último son el mismo y además no hay más vértices coincidentes que estos dos.
- Un *circuito hamiltoniano* es un circuito que recorre todos los nodos del grafo una única vez.
- Un *camino hamiltoniano* es un camino que recorre todos los nodos del grafo una única vez.

En la Figura 1.1 podemos ver un ejemplo de un grafo no orientado, de un grafo orientado y de un grafo completo, donde se aprecia de forma clara la diferencia existente entre ellos.

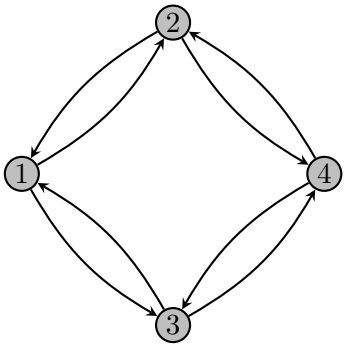
El problema del viajante puede definirse de muchas formas según tomemos unas u otras restricciones en él, algunas de estas derivan en casos particulares que veremos más adelante. Para la siguiente definición, suponemos que tenemos un grafo no orientado, o lo que es lo mismo, que el camino para ir y volver de una ciudad a otra es el mismo y por tanto tiene el mismo coste asociado. Estas condiciones las mantendremos a lo largo de todo el trabajo.

Definición 1.2. El *problema del viajante de comercio* en un grafo no orientado G con costes c consiste en encontrar un circuito hamiltoniano en G de coste mínimo.

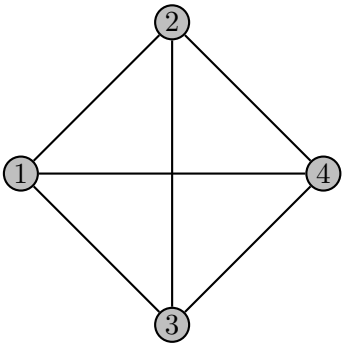
En el ejemplo de la Figura 1.2, se puede ver claramente que no puede haber un circuito hamiltoniano que contenga al arco $(2, 3)$, ya que si lo hubiera esto obligaría a que el circuito pasara por otro nodo más de una vez además del nodo inicial y por tanto el circuito que obtendríamos ya no sería un circuito hamiltoniano. Por consiguiente, el circuito representado en la figura es la única solución para este problema del viajante en un grafo G no orientado.



(a) Grafo no orientado.

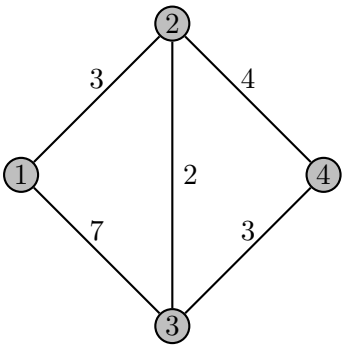


(b) Grafo orientado.



(c) Grafo completo.

Figura 1.1: Tipos de grafos.



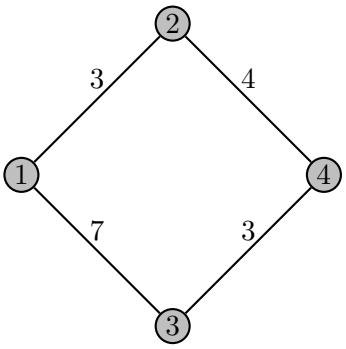


Figura 1.2: Problema del viajante y su solución.

1.1.1. Variaciones del TSP

Hay muchas clasificaciones diferentes del TSP, una de ellas es la que encontramos en el libro de Davendra [1] y que distingue tres variaciones del TSP, el *problema del viajante simétrico* (sTSP), el *problema del viajante asimétrico* (aTSP) y el *problema del viajante con múltiples viajantes* (mTSP).

- **sTSP:** Sea $V = \{v_1, ..., v_n\}$ el conjunto de ciudades (nodos), $A = \{(r, s) : r, s \in V\}$ el conjunto de arcos y d_{rs} el coste asociado a cada arco $(r, s) \in A$, donde $d_{rs} = d_{sr}$. El sTPS es el problema que consiste en encontrar un circuito de mínimo coste que visite cada ciudad una única vez, a excepción de la primera ciudad que será la primera y la última en ser visitada. Además, cuando d_{rs} es la distancia euclidiana entre la ciudad r y la ciudad s, tenemos un sTSP euclideano.

- **aTSP:** Supongamos que nos encontramos en las mismas condiciones del problema sTPS, con la diferencia de que $d_{rs} \neq d_{sr}$, para algún arco del grafo, entonces nuestro problema TSP es un problema aTSP.

- **mTSP:** Dados un conjunto de nodos donde tenemos m vendedores localizados en un único nodo al que llamaremos almacén, y al resto de nodos (ciudades) que tiene que visitar les llamaremos nodos intermedios. EL mTSP consiste en encontrar circuitos de coste mínimo para los m vendedores, que empiecen y terminen en el nodo almacén y de manera que cada nodo intermedio se visita una única vez. El coste de cada arco puede ser la distancia entre ciudades, el tiempo de ir de una ciudad a otra, etc. Según las variables que obtengamos podemos tener diferentes variaciones de este problema, que son las siguiente:

  - *Uno o varios almacenes.* Si sólo tenemos un almacén, todos los vendedores finalizarán su recorrido en él, mientras que si tenemos más nodos que sean almacenes, los vendedores pueden regresar al almacén inicial o a cualquiera de los otros almacenes, respetando que el número de vendedores en cada almacén después del recorrido tiene que ser el mismo que el número de vendedores que había antes de iniciar el circuito.
  - *Número de vendedores.* El número de vendedores del problema puede ser fijo o puede venir dado por una variable que esté acotada
  - *Coste.* Cuando el número de vendedores no es fijo, cada uno de ellos tiene asociado un coste fijo en el que se incurre cuando este vendedor entra en el problema. En este caso, otro objetivo del problema será minimizar el coste de los vendedores.
  - *Ventanas de tiempo.* A veces, un nodo tiene que ser visitado en un periodo determinado de tiempo al cual llamamos ventana de tiempo, este es un caso particular del mTSP que llamamos *problema del viajante múltiple con plazo especificado* (mTSPTW). Este problema es típico de problemas relacionados con la aeronáutica.
  - *Restricciones.* Las restricciones pueden estar en el número de nodos que cada vendedor visita, minimizar o maximizar la distancia que el vendedor tiene que recorrer, o alguna restricción de otro tipo.

El mTSP es generalmente tratado como un problema de enrutamiento de vehículos relajado (VRP) donde no hay restricciones de capacidad. Por tanto la formulación del problema y los métodos de solución para el VRP son válidos para el problema mTSP si asignamos una capacidad grande a los vendedores. Si solo tenemos un vendedor, el problema mTSP se reduce al problema TSP.

1.1.2. Formulación Matemática

El problema del viajante de comercio, además de plantearse como un problema de optimización en redes como lo llevamos haciendo en este capítulo, véase la Definición 1.2, también podemos pensarlo como un problema de optimización general, siendo en este caso tratado como un *problema de programación lineal entera.*

Consideramos el problema del viajante de comercio con un conjunto de nodos (ciudades) N, compuesto por $n+1$ nodos numerados del $0, 1, ..., n$. Las distancias entre los nodos o las ciudades viene dada por los parámetros c_{ij}, donde i es el nodo de origen y j el nodo de destino y almacenamos los valores en la matriz C (la matriz C no es necesariamente simétrica, es decir, no tiene que ocurrir que $c_{ij} = c_{ji}$).El problema de optimización consistirá en definir un grafo a través de las variables x_{ij}, donde $x_{ij} = 1$ si el arco (i, j) está en el grafo y $x_{ij} = 0$ si el arco no está en el grafo. Podemos definir el TSP como sigue:

$$\text{minimizar} \quad \sum_{i,j=0}^{n} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{sujeto a} \quad (1) \quad & \sum_{i=0}^{n} x_{ij} = 1 && j = 0, ..., n \\
(2) \quad & \sum_{j=0}^{n} x_{ij} = 1 && i = 0, ..., n \\
(3) \quad & \sum_{i,j}^{n} x_{ij} \leq |K| - 1 && 1 \leq i \neq j \leq n, K \subsetneq N \\
& 0 \leq x_{ij} \leq 1, && x_{ij} \in Z, \forall i, j
\end{aligned}$$

Las desigualdades en (1) nos piden que en cada nodo entre un único arco y las desigualdades en (2) nos piden que de cada nodo salga un único arco (de las desigualdades (1) y (2) se sigue que del nodo 0 también tiene que salir un único arco). De las desigualdades de (1) y (2) ya nos obligan a que de cada nodo salga y entre un único arco que es lo que estamos buscando, pero con esto no nos llega, ya que como podemos ver en la Figura 1.3, se pueden cumplir estas condiciones y obtener soluciones con varios subcircuitos, cosa que no nos interesa, ya que en el TSP el viajante debe llegar de nuevo al nodo de origen visitando todos los siguientes, y así no se cumple. Para solucionar este problema añadimos la condición (3), que nos dice que no puede haber un subgrafo con un número mayor de aristas que el número de nodos menos uno, siempre que el conjunto de nodos sea distinto que el número de nodos total.

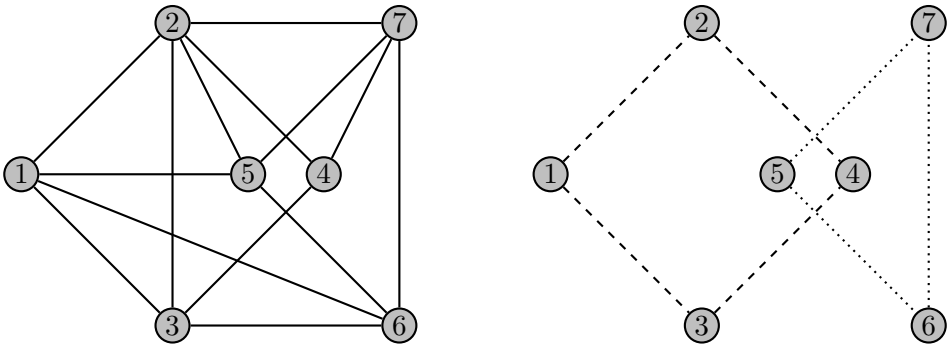


Figura 1.3: Subcircuitos factibles para una solución del problema del viajante si no imponemos las condiciones de (3).

El problema que tiene esta formulación a la hora de intentar programarla es que no es eficiente, ya que en la condición (3) tenemos tantas restricciones como el número de subconjuntos de nuestro conjunto de nodos, es decir, un total de $2^{n+1} - 2$ restricciones. Una formulación más práctica a la hora de atacar este problema es la siguiente debida a *Tucker*, que podemos encontrar en el Capítulo 13 del libro de Papadimitrou [4], y que esencialmente lo único que modifica es la restricción (3).

$$\text{minimizar} \quad \sum_{i,j=0}^{n} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{sujeto a} \quad (1) \quad & \sum_{i=0}^{n} x_{ij} = 1 && j = 0, ..., n \\
(2) \quad & \sum_{j=0}^{n} x_{ij} = 1 && i = 0, ..., n \\
(3) \quad & u_i - u_j + nx_{ij} \leq n - 1 && 1 \leq i \neq j \leq n \\
& 0 \leq x_{ij} \leq 1, && x_{ij} \in Z, \forall i, j
\end{aligned}$$

Donde u_i, con $i = 1, ..., n$, son variables reales de libre disposición, llamadas condiciones de *Miller-Tucker*, que evitan que una posible solución del problema sea con subcircuitos y mejora la anterior ya que reduce el número de restricciones considerablemente. Probaremos ahora que está última formulación define el problema del viajante.

Proposición 1.1. *Las restricciones anteriores* (1)*,* (2) *y* (3) *definen el TSP.*

Demostración. Primero demostraremos que toda solución factible de este problema es un circuito hamiltoniano. Para ver esto mostraremos que todo circuito pasa a través de la ciudad 0, por tanto no puede ser que una solución factible tenga dos subcircuitos en ella ya que los dos tendrían que contener a la ciudad 0 y eso no es posible por las restricciones de (1) y (2). Supongamos que no, supongamos que la secuencia de ciudades $i_1, ..., i_k$ es un circuito que excluye al nodo 0. Podemos escribir las restricciones de (3) a lo largo del camino como:

$$u_{ij} - u_{ij_{+1}} + n \leq n-1 \quad \text{con} \quad j = 1, ..., k-1$$
$$u_{ik} - u_{i1} + n \leq n-1.$$

Sumando estas restricciones, obtenemos una contradicción, ya que llegamos a que $kn \leq k(n-1)$, y esto no es cierto si $k > 0$.

Para acabar la demostración vamos a demostrar que para todo circuito hamiltoniano factible, hay valores de u_i que satisfacen las restricciones de (3). De hecho, sea $u_0 = 0$ y $u_i = t$ si la ciudad i es la t-ésima ciudad en ser visitada en el camino, $t = 1, ..., n$. Si $x_{ij} = 0$, tenemos que

$$u_i - u_j \leq n-1 \quad con \quad 1 \leq i \neq j \leq n$$

que siempre se cumple ya que si $u_i = n$ y $u_j = 0$, como el circuito ha de retornar al nodo 0, tenemos que el arco $(i, 0)$ estará en el circuito, y por lo tanto $x_{i0} = 1$. Si $x_{ij} = 1$ tenemos:

$$u_i - u_j + n \leq n-1$$

que se cumplen porque $u_i - u_j = -1$ si i y j son ciudades consecutivas en el camino (darse cuenta que el caso $u_i = n$ y $u_j = 0$ esta excluido por las restricciones de (3)).

Esta formulación implica a las variables x_{ij}, las cuales están obligadas a ser enteras, y las variables u_i no lo son. Tal problema se llama *problema de programación lineal entera mixta.*

□

1.2. Algoritmos y complejidad computacional.

Cuando decimos que queremos calcular una solución para el TSP, lo que queremos es buscar un algoritmo que nos permita alcanzar una solución para este problema. Un algoritmo es un procedimiento matemático, paso a paso, descrito a través de instrucciones totalmente inequívocas, que empieza en unas condiciones iniciales especificadas y, eventualmente, finaliza devolviendo el resultado deseado (o una "aproximación" del mismo).

De forma muy general, podemos decir que tenemos dos tipos de algoritmos; los algoritmos *determinísticos*, que son aquellos que fijado un problema, todas las ejecuciones del algoritmo producirán el mismo resultado final e incluso los valores intermedios antes de llegar a esa solución serán los mismos. Los algoritmos *no determinísticos* en los que uno introduce cierta aleatoriedad en el proceso de búsqueda de la solución. Otra posible clasificación de los algoritmos atendiendo a la precisión de los mismos sería: *algoritmos exactos*, que siempre devuelven la solución óptima del problema; *algoritmos aproximados*, que producen soluciones que están dentro de un determinado ratio de la solución óptima y los *algoritmos heurísticos*, que producen soluciones sin ninguna garantía de optimalidad pero que a cambio, suelen tener un tiempo de ejecución menor que el de los algoritmos en las otras clases.

1.2.1. Complejidad Computacional

La *complejidad computacional* surge de plantearse si un algoritmo es o no efectivo, es decir si calcula la solución de forma precisa o con un error despreciable en un tiempo razonable, o por el contrario los costes para llegar a esa solución, por ejemplo de tiempo, son tan altos que no merece la pena utilizar ese algoritmo para llegar a la solución del problema. La idea de complejidad computacional comienza a estudiarse en los primeros años de la década de los 70 y lo que se intenta es intentar acotar el número de operaciones que el algoritmo necesita en función del tamaño de los datos necesarios para especificar el problema.

Para esto, diremos que un algoritmo tiene una velocidad $O(f(n))$ si existen constantes k y n_0 tales que el número de operaciones efectuadas por el algoritmo para resolver cualquier problema de tamaño $n \geq n_0$ es, como mucho $kf(n)$. En la práctica, interesa encontrar algoritmos cuya función f sea una función polinomial de n, ya que los algoritmos no polinomiales pueden ser muy poco útiles en la práctica para estudiar problemas grandes.

Para medir el rendimiento de un algoritmo tenemos tres enfoques: el *análisis empírico* donde estimamos el comportamiento del algoritmo testándolo con diferentes ejemplos, el *análisis del caso promedio* en el que estimamos el número medio de pasos que necesita el algoritmo y por último el *análisis del peor caso* que consiste en encontrar cotas superiores para el número de operaciones que va a necesitar el algoritmo. Este último caso es en el que nos basaremos nosotros a la hora de exponer los siguientes resultados teóricos.

Diremos que una clase de problemas de optimización es *fácil* si se puede desarrollar un algoritmo que resuelva cualquier problema de la clase en tiempo polinomial. Los algoritmos en tiempo polinomial se llaman algoritmos *eficientes*. Hay muchos problemas de programación de los que no sabemos si existe algún algoritmo eficiente que los resuelva. Antes de entrar a definir formalmente las clases de complejidad computacional que nos interesan para nuestro trabajo, tenemos que hablar de otro concepto muy importante que influirá de forma determinante en el desarrollo de este trabajo, el de la *máquina de Turing*, ya que nuestras definiciones, se basarán en este concepto, además de los diferentes resultados teóricos que utilizaremos a lo largo de todo el estudio.

En esencia, una máquina de Turing es un dispositivo que manipula símbolos sobre una tira de cinta de acuerdo a una tabla de reglas, como vemos en el dibujo de la Figura 1.4. Aunque el funcionamiento de la máquina de Turing es muy simple, la verdadera importancia de esta es que puede ser adaptada para simular la lógica de cualquier algoritmo de computador, con lo cual podemos estudiar matemáticamente la eficiencia de distintos algoritmos estudiando el tiempo de resolución necesario en una máquina de Turing. Alan Turing introdujo el concepto de máquina de Turing en el trabajo *On computable numbers, with an application to the Entscheidungsproblem* [6], publicado en la revista *Proceedings of the London Mathematical Society* por la Sociedad Matemática de Lóndres en 1936.

La máquina de Turing modela matemáticamente a una máquina que opera mecánicamente sobre una cinta. En esta cinta hay símbolos que la máquina puede leer y escribir, uno cada vez, usando un cabezal lector/escritor de cinta. La operación está completamente determinada por un conjunto finito de instrucciones elementales tales como “en el estado 42, si el símbolo visto es 0 , escribe un 1; si el símbolo visto es 1, cambia al estado 17; en el estado 17, si el símbolo

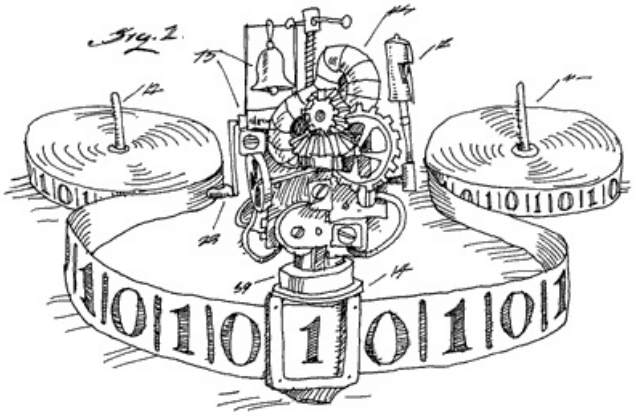


Figura 1.4: Ilustración de una máquina de Turing.

visto es 0, escribe un 1 y cambia al estado 6; etc".

Podemos clasificar las máquinas de Turing en dos tipos, *las máquinas de Turing deterministas*, DTM, y *las máquinas de Turing no deterministas*, NTM.

En una DTM, el conjunto de reglas prescribe como mucho la realización de una acción en cada situación. Es una función de transición que para cada estado y cada símbolo que lee, especifica tres cosas: el símbolo a escribir, la dirección en la que moverse y el nuevo estado interno.

Por el contrario, una NTM puede prescribir un conjunto de acciones. Se puede especificar más de una acción, las transiciones ya no son únicas. Una forma de interpretar esto es que una NTM tiene la capacidad de "ramificarse" en múltiples copias en cada paso, realizando las distintas acciones. Si alguna rama termina el algoritmo todas terminan.

Supongamos que tenemos dos clases de problemas, P_1 y P_2. Decimos que P_1 se *reduce* a P_2 en tiempo polinomial si cualquier ejemplo de la clase P_1 se puede transformar en tiempo polinomial en un ejemplo de la clase P_2.

Con estos conceptos, ya podemos entrar a definir las clases de complejidad que nosotros queríamos, que son P, NP, NP-*completo* y NP-*duro*, y sus relaciones como podemos ver en la Figura 1.5.

- ***P*:** Un problema pertenece a la clase de complejidad P, si existe un algoritmo que resuelve cualquier ejemplo de este problema en un tiempo polinomial. De modo equivalente, un problema pertenece a la clase de complejidad P si cualquier ejemplo de este problema puede ser resuelto en tiempo polinomial usando una máquina de Turing determinista (DTM).
- ***NP*:** Un problema pertenece a la clase de complejidad NP, si existe un algoritmo que puede verificar cualquier solución de un ejemplo de esta clase en tiempo polinomial. De modo equivalente, un problema pertenece a la clase de complejidad NP si cualquier ejemplo de este problema puede ser resuelto en una máquina de Turing no determinista (NTM).

- ***NP*-completo:** Un problema P_1 pertenece a al clase de complejidad NP-completo si cualquier problema de la clase NP se puede reducir a P_1 en tiempo polinomial. Todo problema que está en NP-completo está también en NP.
- ***NP*-duro:** Un problema es NP-duro si cualquier problema de la clase NP se puede reducir a él en tiempo polinomial. La diferencia que existe entre este problema y un problema NP-completo es que un problema puede ser NP-duro sin pertenecer a NP.

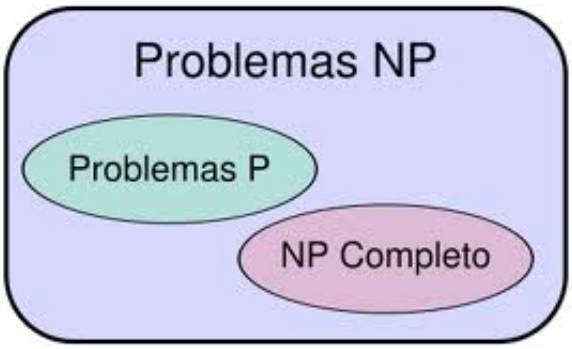


Figura 1.5: Relaciones de las clases de complejidad.

Una vez dadas estas definiciones, destacar que una de las mayores incógnitas de la complejidad computacional es poder decir si las clases P y NP son iguales, $P = NP$. Aunque existe un amplio acuerdo en que estas clases de complejidad computacional tienen que ser distintas, nunca nadie fue capaz de probarlo. Se puede extraer por la definición de ambas que $P \subset NP$, pero el problema está en la otra implicación. Es más, este problema quedaría probado si podemos encontrar un algoritmo que resuelva un problema de la clase NP-completo en tiempo polinomial, ya que, al poder reducir cualquier problema de la clase NP a él en tiempo polinomial, estaríamos probando que podemos encontrar un algoritmo que resuelva cualquier problema de clase NP en tiempo polinomial, y por tanto estaríamos probando que $NP \subset P$, y en consecuencia tendríamos probado que $P = NP$. Si esto se pudiera probar, sería un gran hallazgo que tendría mucha repercusión en multitud de campos. Por ejemplo, uno de los principales problemas que encontraríamos, es que estaríamos diciendo que existe un algoritmo polinomial para factorizar en números primos, lo que implicaría que muchos sistemas de seguridad de todo el mundo se verían comprometidos de forma muy importante.

Una vez vistos los anteriores conceptos teóricos, ya tenemos la base para poder demostrar que el problema del viajante pertenece a la clase de complejidad computacional NP-completo, que veremos en el siguiente capítulo.

Capítulo 2

El problema del viajante es NP-completo

Contenidos

En este capítulo vamos a demostrar que el problema del viajante pertenece a la clase de complejidad computacional NP-completo.

En el capítulo 2 afirmamos que el problema del viajante es NP-duro y que por lo tanto es muy difícil llegar a calcular una solución general para este problema. Ahora decimos que vamos a demostrar que es un problema NP-completo. Ambas afirmaciones son ciertas aunque parezca que pueden ser contradictorias. El problema del viajante es un problema de optimización combinatoria y por tanto según como nos planteemos el problema tenemos tres versiones del mismo: *la versión de optimización*, *la versión de evaluación* y *la versión de reconocimiento*. La complejidad computacional de cada una de estas versiones es diferente, así se explica la posible contradicción. En la primera parte de este capítulo explicaremos este posible conflicto basándonos en el libro de Papadimitrou [4] (Capítulo 15, Sección 2).

Luego, daremos una serie conceptos y de resultados previos, como por ejemplo el *teorema de Cook*, que serán la base del objetivo de este capítulo.

2.1. El problema del viajante: NP-duro y NP-completo.

Intentaremos dar una respuesta concisa y fácil de entender a la aparente contradicción mencionada en el final de la sección anterior. El problema del viajante es un problema de optimización combinatoria que, para la realización de este trabajo y en particular de este tema, enfocaremos desde el punto de vista computacional.

Un *problema de optimización* lo definimos como un par (F, c), donde F es el conjunto de soluciones factibles y c es la función de coste, $c : F \longrightarrow R$. Como vamos a tratar estos problemas desde el punto de vista de la teoría de la computación, fijamos una manera de formular estos problemas que nos sea útil a la hora de introducirlos en un ordenador. Se podría hacer un listado de todas las soluciones posibles con su respectivo coste c. Esto no es práctico ya que hay problemas para los que sería inviable de hacer por el gran número de soluciones posibles. Por ello, en lo que sigue vamos a suponer que F viene dado implícitamente por un algoritmo que llamaremos $\mathcal{A}_F$ y c por otro algoritmo al que llamaremos $\mathcal{A}_c$ de la siguiente manera. Dada una posible solución f y un conjunto de parámetros S, el algoritmo $\mathcal{A}_\mathcal{F}$ decide si f pertenece o no a la región factible, F. Por otro lado, dada una solución factible f y otro conjunto de parámetros Q, $\mathcal{A}_c$ devuelve el coste de la solución f, $c(f)$.

Un problema de optimización combinatoria lo podemos formular de tres maneras diferentes, mediante la *versión de optimización*, la *versión de evaluación* o la *versión de reconocimiento*. Estas son las siguientes:

- **Versión de optimización**: Teniendo en cuenta las representaciones de los parámetros S y Q para los algoritmos $\mathcal{A}_F$ y $\mathcal{A}_c$ respectivamente, encontrar la solución óptima.
- **Versión de evaluación**: Dados S y Q, encontrar el coste de la solución óptima.
- **Versión de reconocimiento**: Dado un problema representado por los parámetros S y Q, y un entero L, ¿Hay una solución factible $f \in F$ tal que $c(f) \leq L$?.[1]

[1] Este mismo problema podemos plantearlo con $c(f) \geq L$, si queremos maximizar

Notar que la versión de reconocimiento del problema de optimización la podemos resolver únicamente contestando con un *sí* o un *no* a la pregunta formulada.

La pregunta que nos planteamos ahora es si la complejidad computacional de las tres versiones es la misma, es decir, si resolver estos tres problemas es igual de difícil.

Para esto asumiremos que tanto $\mathcal{A}_F$ como $\mathcal{A}_c$ son algoritmos en tiempo polinomial, algo que es muy frecuente. Se puede demostrar que la versión de evaluación del problema se puede resolver de manera eficiente siempre que la versión de reconocimiento también lo sea. Por otra parte, decir que no se conoce ningún método general que resuelva ninguna de estas versiones de este problema.

De lo anterior podemos afirmar que las versiones de evaluación y reconocimiento no son más difíciles que la versión de optimización, y que la versión de reconocimiento no es más difícil que la versión de evaluación. Si ordenamos estas versiones según su nivel de dificultad obtenemos que: versión de *optimización* > *evaluación* > *reconocimiento*.

A nosotros nos interesa particularmente el problema del viajante. De lo dicho anteriormente deducimos que la versión de optimización del problema del viajante consiste en encontrar un circuito hamiltoniano de longitud mínima (coste mínimo). Encontrar la solución a esto es muy complicado, es más esta versión es NP-duro. Por otro lado, la versión de reconocimiento del problema del viajante que consiste en que dado un problema del viajante y un circuito hamiltoniano (solución factible) ver si su longitud (coste) es menor (mayor) que un cierto L prefijado de antemano. Responder a está pregunta, como vimos antes no es tan difícil, es más esta versión es NP-completa.

De ahora en adelante vamos a trabajar con la versión de reconocimiento del problema del viajante y demostraremos que esta versión del problema es NP-completa.

2.2. Clase de problemas auxiliares.

Antes de meternos a fondo en el teorema de Cook vamos a describir una serie de problemas que nos resultarán de gran ayuda a la hora de llevar a cabo su demostración. El problema que vamos a ver será el *problema de satisfacibilidad*. Este es la base del teorema de Cook y será clave a la hora de demostrar que el problema del viajante de comercio es NP-completo. Hablaremos brevemente de lo que son las variables y las fórmulas *booleanas*, ya que el problema de satisfacibilidad involucra directamente a este tipo de variables.

2.2.1. Variables booleanas

Una **variable booleana** x es una variable que solo puede tomar dos valores, *sí* o *no*. Las podemos combinar utilizando los siguientes operadores lógicos: ***o***, representado con el símbolo $+$, quiere decir que ocurre una cosa o la otra; ***y***, representado con el símbolo $\cdot$, significa que ocurre una cosa y la otra; y ***no***, representado con $\bar{x}$, nos indica justo lo opuesto del valor de la variable x. Combinando las variables booleanas con los operadores lógicos que acabamos de presentar obtenemos las **fórmulas booleanas**. Por ejemplo:

$$\bar{x}_3 \cdot (x_1 + \bar{x}_2 + x_3) \tag{2.1}$$

Dado un valor *t(x)* para cada variable x evaluaremos la fórmula booleana de la misma manera que lo hacemos con una expresión algebraica. Si miramos el ejemplo anterior 2.1, y lo evaluamos con el siguiente conjunto de valores (llamados *asignación de verdad*) $t(x_1) = sí$, $t(x_2) = sí$ y $t(x_3) = no$, obtenemos que la fórmula booleana del ejemplo es verdadera.

Las fórmulas booleanas que son ciertas para alguna asignación de verdad las llamamos *satisfactibles*. No todas las fórmulas booleanas son satisfactibles. Hay algunas que no son ciertas para ninguna asignación de verdad ya que ellas mismas incurren en alguna contradicción. Consideramos el siguiente ejemplo:

$$(x_1 + x_2 + x_3) \cdot (x_1 + \bar{x}_2) \cdot (x_2 + \bar{x}_3) \cdot (x_3 + \bar{x}_1) \cdot (\bar{x}_1 + \bar{x}_2 + \bar{x}_3) \tag{2.2}$$

Para que esta fórmula 2.2 sea cierta todas las subfórmulas que aparecen en paréntesis (llamadas *cláusulas*) deben ser ciertas. La primera cláusula nos dice que al menos una de las tres variables tiene que ser un *sí*. Las tres clausulas siguientes fuerzan a que las variables sean iguales. Supongamos que x_1 es un *no*, como la segunda cláusula debe ser verdadera, la variable x_2 tendrá que ser un *no*. Por la misma razón aplicada a la tercera clausula, la variable x_3 debe ser también un *no*. Esto contradice lo que nos pedía la primera cláusula, ya que ninguna de las tres variables puede ser un *sí*. De forma más general, si una variable asume un valor cualquiera; la segunda, tercera y cuarta cláusula obligan a que las otras variables tomen el mismo valor. Si a esto añadimos que la primera cláusula me obliga a que al menos una sea un *sí*, y para que la última cláusula sea cierta una de ellas tiene que ser un *no*, llegamos a una contradicción. La formula es *insatisfactible*.

2.2.2. El problema de satisfacibilidad

El problema de satisfacibilidad es el siguiente:

"*Dadas m cláusulas C_1, ... ,C_m, implicando a las variables x_1, ... ,x_n. ¿La fórmula dada por $C_1 \cdot ... \cdot C_m$ es satisfactible?*"

El problema de satisfacibilidad es el problema central de la lógica matemática. Es de gran interés intentar construir algoritmos eficientes para calcular sus soluciones. Un posible algoritmo que calcule la solución a este problema sería aquel que pruebe con todas las asignaciones de verdad posibles y vea si al menos una de ellas satisface la ecuación. Este algoritmo no es eficiente. Tenemos 2^n posibles asignaciones de verdad, donde n es el número de variables. Esto implica que al aumentar el número de variables, el número de posibles asignaciones de verdad aumenta considerablemente. Hasta el momento no existe ningún algoritmo eficiente que resuelva el problema de satisfacibilidad.

Es interesante observar que este problema se puede formular como un problema de programación lineal entera (ILP). Esto es inmediato, únicamente identificaremos 1 con "*sí*" y 0 con "*no*". Por otro lado, "*o*" se convierte en una suma ordinaria +, y $\bar{x}$ pasa a ser $1-x$. Es necesario que, para cada cláusula C, al menos una de sus variables sea cierta o, lo que es lo mismo:

$$\sum_{x \in C} x + \sum_{\bar{x} \in C} (1 - x) \geq 1$$

Veamos ahora como quedaría enunciado el ejemplo 2.2 como un problema de programación lineal:

$$\begin{array}{rcl} x_1 + x_2 + x_3 & \geq & 1 \\ x_1 + (1 - x_2) & \geq & 1 \\ x_2 + (1 - x_3) & \geq & 1 \\ x_3 + (1 - x_1) & \geq & 1 \\ (1 - x_1) + (1 - x_2) + (1 - x_3) & \geq & 1 \\ x_1, x_2, x_3 & \leq & 1 \\ x_1, x_2, x_3 & \geq & 0 \\ x_1, x_2, x_3 \in Z & & \end{array} \tag{2.3}$$

Es fácil ver como las restricciones del ejemplo 2.2 y del problema de programación lineal entera 2.3 definen un mismo problema. Sabiendo esto, si tenemos una fórmula booleana con varias clausulas conectadas, *diremos que ese problema es satisfactible si el correspondiente problema de programación lineal entera tiene una solución factible.*

Para que el problema 2.3 sea un problema de programación lineal entera nos hace falta una función objetivo. Para solucionar esto simplemente añadimos la inecuación $x_1 + x_2 + x_3 \geq y$ al sistema. Ahora el objetivo del problema será maximizar el valor de la variable y. Así ya tenemos nuestra función objetivo y por tanto enunciado un problema de programación lineal entera.

Este problema pertenece a una clase especial muy importante de problemas de programación lineal entera que solo admiten soluciones que sean cero o uno. Estos tipos de problemas de programación lineal entera se llaman *problemas de programación lineal binaria* o *problemas de programación lineal cero-uno*. La antepenúltima y penúltima fila del sistema normalmente se escriben como $x_j \in \{0, 1\}$, $j = 1, ..., n$ o con la igualdad $x_j^2 = x_j$ que son equivalentes.

2.3. Teorema de Cook.

Para probar que un problema es NP-completo debemos probar que:

a) El problema pertenece a la clase NP.

b) Cualquier otro problema que pertenezca a NP se puede transformar en nuestro problema en tiempo polinomial.

Para probar el segundo punto se demuestra que un problema NP-completo conocido se puede transformar en nuestro problema en tiempo polinomial. Con esto y utilizando la propiedad transitiva de las transformación tendríamos el segundo punto probado de la siguiente forma:

Sabemos que una transformación en tiempo polinomial es transitiva, es decir si $A \longrightarrow B$ y $B \longrightarrow C$ entonces $A \longrightarrow C$. Supongamos que B es NP-completo, entonces $\forall A \in NP$, existe una transformación en tiempo polinomial $A \longrightarrow B$. Tomamos un $A \in NP$ y consideramos la transformación en tiempo polinomial $A \longrightarrow B$ que sabemos que existe por lo dicho anteriormente.

Suponemos que tenemos probada que existe una transformación en tiempo polinomial entre el problema NP-completo B y el problema que nosotros queremos demostrar que es NP-completo, C. Esta transformación será $B \longrightarrow C$.

Ahora aplicando la propiedad transitiva de la transformación en tiempo polinomial llegamos a la conclusión de que existe una transformación en tiempo polinomial entre A y C, $A \longrightarrow C$. Como esto se cumple para todo problema $A \in NP$, ya tenemos demostrado que nuestro problema C es NP-completo.

En este apartado trabajaremos con la versión de reconocimiento del problema del viajante. Para esto utilizaremos una definición diferente (aunque equivalente) de la que habíamos dado de lo que es un problema NP. Esa definición es la siguiente:

Definición 2.1. Decimos que un problema de reconocimiento A pertenece a la clase de complejidad computacional NP si existe un polinomio $p(n)$ y un algoritmo $\mathcal{A}$ (algoritmo de certificación) de tal manera que lo siguiente es cierto:

La cadena x es un ejemplo *si* de A si y solo si existe una cadena de símbolos en Σ, $c(x)$ (el certificado), tal que $|c(x)| \leq p(|x|)$, con la propiedad de que $\mathcal{A}$, si se le da la entrada $x\$c(x)$, llega a la respuesta *si* después de, a lo sumo $p(|x|)$ pasos.

Formalizaremos la idea de algoritmo de certificación. Este algoritmo puede ser considerado como un dispositivo que lee y modifica los símbolos de una cadena. Varía una posición cada vez a través de una cabeza de lectura y escritura (de forma muy semejante al funcionamiento de una máquina de Turing) como podemos ver en la figura 2.1.

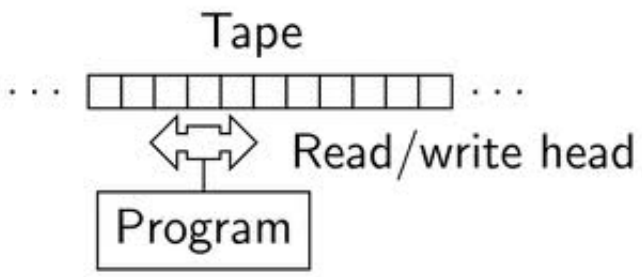


Figura 2.1: Algoritmo de certificación.

2.3.1. Funcionamiento del algoritmo de certificación

Inicialmente, la cabeza está escaneando la posición más a la izquierda de la cadena, y el programa está a punto de ejecutar su primera instrucción. Las instrucciones del programa son de la forma, l: si σ entonces $(\sigma^{'}; o; l^{'})$. Donde l y $l^{'}$ son números de instrucciones (etiquetas), σ y $\sigma^{'}$ son símbolos del alfabeto Σ y o es uno de los números 1, 0 o -1. El significado estos símbolos es el siguiente:

Si el símbolo analizado es σ, entonces borrarlo y escribe $\sigma^{'}$ en su lugar, moverse o posiciones a la derecha y ejecutar la instrucción número $l^{'}$; en caso contrario continuar a la siguiente instrucción.

Moverse $o = 0$ lugares a la derecha significa quedarse en la misma posición en la que se encuentra, y moverse $o = -1$ lugares a la derecha significa moverse una posición a la izquierda. Si denotamos por $|A|$ al tamaño del programa, es decir, al número total de instrucciones, entonces la última instrucción, $|A|$, será "aceptar".

Decimos que una cadena $x\$c(x)$ es aceptada por $\mathcal{A}$, si el algoritmo comenzó correctamente en la cadena y llegó a la última instrucción antes de que se cumplieran el número máximo de pasos $p(|x|)$. Si el algoritmo se ejecuta el número máximo de instrucciones $p(|x|)$ sin llegar a completar la cadena, o si la cabeza cae de la cadena, entonces decimos que $x\$c(x)$ es rechazada.

Veremos ahora tres propiedades de este algoritmo de certificación:

1. El funcionamiento del algoritmo parece estar limitado al espacio proporcionado por la cadena, y no hay un "papel de borrador" extra. Este problema puede solucionarse fácilmente. Podemos definir el certificado para que contenga en sí mismo un espacio adicional para uso del algoritmo. Debido a esto, ahora debemos asumir que que para todos las cadenas x de la misma longitud, el tamaño de $c(x)$ es $|c(x)| = p(|x|) - |x| - 1$. Por tanto el tamaño del conjunto $x\$c(x)$ es igual a $p(|x|)$. Esto no es una perdida de generalidad ya que si el certificado es más pequeño siempre se pueden completar con espacios en blanco que rellenen hasta ese tamaño. Además, si las entradas son mas largas que $p(|x|)$ no son significativas para $\mathcal{A}$, ya que $\mathcal{A}$ no puede inspeccionar toda la longitud de esa entrada dentro de los $p(|x|)$ pasos.

2. El repertorio de instrucciones disponible es muy limitado. Pero a pesar de ser tan limitado, puede ser usado para simular instrucciones más complicadas solamente introduciendo un único factor multiplicativo constante en el número total de pasos, pues esto no afectará al carácter polinomial del algoritmo.

3. Aunque no vamos a detenernos en esto, otra propiedad importante de este modelo es que es capaz de hacer direccionamientos. Por ejemplo, mover el decimoquinto símbolo de la cadena, que recuerda a los ordenadores de acceso aleatorio.

Apoyándonos en lo que acabamos de ver, vamos a dar el siguiente resultado, el teorema de Cook, que será transcendental en la consecución de nuestro objetivo. Nos basaremos en la demostración que se da en el libro de Papadimitou [4] (Capítulo 15, Sección 5).

Ejemplo: Problema de Satisfacibilidad y el algoritmo de certificación

El problema de Satisfacibilidad anteriormente analizado es NP. Dado un conjunto de cláusulas C_1, C_2,..., C_m que involucran a las variables booleanas x_1, x_2,..., x_n, donde el problema es satisfactible.El algoritmo de certificación aseguraría simplemente que las cláusulas C_i están bien definidas y que todas las cláusulas son verdad bajo la asignación de verdad, que nosotros tomamos como certificado.

2.3.2. Teorema de Cook

Teorema 2.1. *Satisfacibilidad es NP-completo*

Demostración. Antes vimos que el problema de *Satisfacibilidad* es NP. Nos queda por probar la segunda parte, que si $A \in NP$, entonces A se puede transformar en tiempo polinomial en un problema de Satisfacibilidad. Dada una cadena x debemos construir una fórmula $F(x)$ usando únicamente el hecho de que $A \in NP$, de tal manera que x es un ejemplo *sí* de A sí y sólo sí $F(x)$ es satisfactible. Vamos a considerar el algoritmo de certificación $\mathcal{A}$ para A, el cual, debido a que $A \in NP$ opera dentro de un polinomio $p(n)$.

La fórmula $F(x)$ será una fórmula booleana (esto es claro ya que estamos tratando el problema de satisfacibilidad). Ya la explicamos anteriormente en el punto 3.2 y usará las variables booleanas que definiremos en las siguientes lineas.

1. La variable booleana $\boldsymbol{x_{ij\sigma}}$ con $0 \leq i, j \leq p(|x|)$ y $\sigma \in \Sigma$. El significado de esta variable cuando vale 1 es el siguiente: en el instante i, la posición j de la cadena contiene el símbolo σ. En el caso de que su valor sea 0, nos dirá que en ese instante la cadena no está en esa posición.

2. La variable booleana $\boldsymbol{y_{ijl}}$ con $0 \leq i \leq p(|x|)$, $0 \leq j \leq p(|x|)+1$ y $1 \leq l \leq |G|$, donde $|G|$ es el número de instrucciones en el algoritmo G. El significado de esta variablen cuando su valor es 1 es el siguiente: En el instante i, leemos la j-ésima posición y ejecutamos la instrucción l-ésima. Notar, que si $j = 0$ o si $j = p(|x|)+1$, significa que el lector ha salido de la cadena y por tanto el cálculo no tiene éxito. Si el valor de esta variable es 0 significa que en el instante i no leemos la j-ésima posición y por tanto no se ejecuta la l-ésima posición.

Estas variables booleanas las combinamos formando la fórmula booleana $F(x)$, que cumple que es satisfactible si y solo si x es un ejemplo *sí* de A. La fórmula $F(x)$ se descompone en cuatro partes como sigue:

$$F(x) = U(x) \cdot S(x) \cdot W(x) \cdot E(x)$$

En lo que sigue descompondremos y analizamos cada una de las cuatro partes de $F(x)$:

- El propósito de $\boldsymbol{U(x)}$ es asegurarse de que en cada instante i con $0 \leq i \leq p(|x|)$, cada posición de la cadena contiene un único símbolo. El lector escanea una única posición dentro de los límites de la cadena, y el programa ejecuta una única declaración.

$$\begin{aligned} U(x) \;=\; & \left(\prod_{\substack{0 \leq i,j \leq p(|x|) \\ \sigma \neq \sigma'}} \left(\bar{x}_{ij\sigma} + \bar{x}_{ij\sigma'} \right) \right) \cdot \left(\prod_{\substack{0 \leq i \leq p(|x|) \\ j \neq j' \; o \; l \neq l'}} \left(\bar{y}_{ijl} + \bar{y}_{ij'l'} \right) \right) \cdot \\ & \cdot \left(\prod_{\substack{0 \leq i \leq p(|x|) \\ 1 \leq l \leq |\mathcal{A}|}} \left(\bar{y}_{i0l} \cdot \bar{y}_{i,p(|x|),l} \right) \right) \cdot \left(\prod_{0 \leq i \leq p(|x|)} \left(\left(\prod_{0 \leq j \leq p(|x|)} \sum_{\sigma \in \Sigma} x_{ij\sigma} \right) \cdot \sum_{\substack{1 \leq j \leq p(|x|) \\ 1 \leq l \leq |\mathcal{A}|}} y_{ijl} \right) \right) \end{aligned}$$

- La fórmula $\boldsymbol{S(x)}$ nos indica que el algoritmo $\mathcal{A}$ comienza correctamente. En el instante 0 el símbolo más a la izquierda $|x|+1$ es $x\$$, la cabeza de lectura y escritura explora el símbolo más a la izquierda de la cadena y la primera instrucción de $\mathcal{A}$ está a punto de ser ejecutada.

$$S(x) \quad = \quad \left(\prod_{j=1}^{|x|} x_{0jx(j)} \right) . x_{0,|x|+1.\$} . y_{011}$$

  ($x(j)$ representa el símbolo j-ésimo de la cadena x)

- La fórmula $\boldsymbol{W(x)}$ indica que el algoritmo G funciona bien, de acuerdo con las instrucciones del programa. $W(x)$ es el conjunto de fórmulas $W_{ij\sigma l}$, una para cada $0 \leq i \leq p(|x|)$, $1 \leq j \leq p(|x|)$, $\sigma \in \Sigma$ y $1 \leq l \leq |\mathcal{A}|$, de tal manera que la l-ésima instrucción de G es:

$$l : \text{si } \sigma \text{ entonces } (\sigma^{'}; o; l^{'})$$

  La fórmula $W_{ij\sigma l}$ está definida de la siguiente manera:

$$\begin{aligned} W_{ij\sigma l} \quad = \quad & \left(\bar{x}_{ij\sigma} + \bar{y}_{ijl} + x_{i+1,j,\sigma'} \right) . \left(\bar{x}_{ij\sigma} + \bar{y}_{ijl} + y_{i+1,j+o,l'} \right) . \\ & . \prod_{\tau \neq \sigma} \left((\bar{x}_{ij\tau} + \bar{y}_{ijl} + x_{i+1,j,\tau}) . (\bar{x}_{ij\tau} + \bar{y}_{ijl} + y_{i+1,j,l+1}) \right) \end{aligned}$$

  Esto significa que siempre que $x_{ij\sigma}$ e y_{ijl} sean ambas verdaderas, en el siguiente instante las variables x e y, que indicaron que $\mathcal{A}$ hizo un movimiento correcto, deben de ser también verdaderas. Para la última instrucción de $\mathcal{A}$, tenemos que para cada i, j y σ:

$$W_{ij\sigma|\mathcal{A}|} \quad = \quad \left(\bar{x}_{ij\sigma} + \bar{y}_{ij|G|} + y_{i+1,j,|\mathcal{A}|} \right)$$

  Indica que una vez que el algoritmo alcance la instrucción *aceptar*, permanezca en ella. Finalmente, $W(x)$ contiene las clausulas:

$$\prod_{\substack{0 \leq i \leq p(|x|) \\ \sigma \in \Sigma \\ 1 \leq l \leq |\mathcal{A}| \\ j \neq j^{'}}} \left(\bar{x}_{ij\sigma} + \bar{y}_{ij^{'}l} + x_{i+1,j,\sigma} \right)$$

  Significa que cada vez que $\mathcal{A}$ esté escaneando una posición distinta de la j-ésima, el símbolo de esa posición permanece sin cambios.

- La última parte de $F(x)$ indica únicamente que $\mathcal{A}$ termina correctamente con el programa de ejecución de la instrucción aceptar. Esto consiste sólo en una clausula:

$$E(x) = \sum_{j=1}^{p(|x|)} y_{p(|x|),j,|\mathcal{A}|}$$

Con esto queda completada la construcción de $F(x)$. Esta construcción solo requiere de un numero de operaciones que es una función polinomial $|x|$. Esto se deduce del hecho de que la longitud total de $F(x)$ es $O(p^3(|x|)log(p(|x|)))$. Por tanto, lo único que nos queda por demostrar para confirmar que esta es una transformación polinomial del problema A al problema de satisfacibilidad es la siguiente afirmación:

"$F(x)$ es satisfactible si y solo si x es un ejemplo *sí* de A"

Para demostrar la implicación *solo si* suponemos que $F(x)$ es satisfactible. Por tanto $U(x)$, $S(x)$, $W(x)$ y $E(x)$ son también satisfactibles por la misma asignación de verdad t. Como t satisface $U(x)$, para todo i y j solo una variable $x_{ij\sigma}$ debe ser cierta, es decir que la j-ésima celda contiene a σ en el instante i. También para todo i, exactamente una de las variables y_{ijl} es cierta, es decir, en el instante i la j-ésima posición de la cadena es examinada y se ejecuta la orden l. Por último, ninguna de las variables y_{i0l} e $y_{i,p(|x|)+1,l}$ pueden ser ciertas, por lo que el lector nunca saldrá de la cadena.

La asignación de verdad t describe algunas secuencias de cadenas, posiciones del lector e instrucciones. Vamos a demostrar ahora que esta secuencia es un cálculo de aceptación válido para $\mathcal{A}$ con una entrada $x\$c(x)$ para algún certificado $c(x)$.

Dado que $S(x)$ también debe ser satisfecho por t la secuencia comienza correctamente. Los primeros $|x| + 1$ lugares ocupados por la cadena correcta $x\$$, y con el primer símbolo de x escaneado mientras se ejecuta la primera instrucción.

$W(x)$ también es satisfecha por t. Esto significa que la secuencia cambia con las reglas de ejecución del algoritmo $\mathcal{A}$. Finalmente, t satisface $E(x)$ solo si el algoritmo termina en su última instrucción de aceptación.

Consecuentemente, si $F(x)$ es satisfactible existe una cadena de longitud apropiada de tal manera que $\mathcal{A}$ acepta $x\$c(x)$, entonces x es un *si* de A.

Para la otra implicación asumimos que x es un *si* de A. Entonces existe una cadena $c(x)$ de longitud $p(|x|) - |x| - 1$, de tal manera que $\mathcal{A}$ acepta $x\$c(x)$. Esto significa que existe una sucesión de cadenas de $p(|x|)$ (con $x\$c(x)$ primero), números de instrucción y posiciones de escaneo que son válidas en $\mathcal{A}$ y terminan con la aceptación de $x\$c(x)$. Esta sucesión define una asignación de verdad t que necesariamente satisface $F(x)$. Lo que completa la prueba de esta afirmación.

La afirmación que acabamos de probar nos dice que $F(x)$ es un *si* de satisfacibilidad si y solo si x es un *si* del problema A. Por tanto, lo que escribimos es una transformación polinómica de A a satisfacibilidad. Además, como A es un problema arbitrario dentro de NP, concluimos que el teorema queda demostrado.

□

2.4. Resultado principal.

Demostraremos ahora que el problema del viajante de comercio es NP-completo. Para ello demostraremos unos resultados previos y veremos que existe una transformación polinómica de una problema NP-completo al problema del viajante. Esto lo podemos ver en el libro de Papadimitrou [4] (Capítulo 15, sección 6).

Un problema de satisfacibilidad al que le restringimos el número de variables que hay en cada clausula a 3, diremos que es un problema de 3-satisfacibilidad. Este problema sigue siendo tan difícil como el problema de satisfacibilidad.

2.4.1. 3-satisfatibilidad es NP-completo

Teorema 2.2. *3-satisfacibilidad es NP-completo*

Demostración. Como 3-satisfacibilidad es un caso particular del problema de satisfacibilidad está en NP. Para demostrar que es NP-completo vamos a demostrar que existe una transformación polinómica de satisfacibilidad a 3-satisfacibilidad. Veremos que satisfacibilidad se transforma polinómicamente en 3-satisfacibilidad.

Consideramos cualquier fórmula F con las clausulas $C_1, ..., C_m$ y construimos una nueva fórmula $F^{'}$ que contenga únicamente 3 variables por clausula, de tal forma que $F^{'}$ es satisfactible si y solo si F lo es. Para esto examinaremos cada una de las clausulas de F por separado. Reemplazaremos cada C_i por un conjunto conjunto equivalente de clausulas donde cada una de ellas tendrá 3 variables. Distinguiremos tres casos:

1. Si C_i tiene 3 variables no hacemos nada.

2. Si C_i tiene más de tres variables, como por ejemplo $C_i = (\lambda_1 + \lambda_2 + ... + \lambda_k)$, $K > 3$, reemplazamos C_i por $k-2$ clausulas como sigue, $(\lambda_1 + \lambda_2 + x_1)\ .\ (\bar{x}_1 + \lambda_3 + x_2)\ .\ (\bar{x}_2 + \lambda_4 + x_3)\ .\ ...\ .\ (\bar{x}_{k-3} + \lambda_{k-1} + \lambda_k)$, donde $x_1, ..., x_{k-3}$ son nuevas variables. Es fácil ver que estas nuevas clausulas son satisfactibles si y solo si C_i lo es. Veamos un caso particular, supongamos que $C_i = (\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4)$ por tanto las nuevas clausulas quedarían, $(\lambda_1 + \lambda_2 + x_1)\,.\,(\bar{x}_1 + \lambda_3 + \lambda_4)$. Es trivial ver que si las nuevas clausulas son ciertas es estrictamente necesario que al menos una de las λ_i sea un *si*. Esto implica que la clausula C_i es cierta. Por otra parte, para que la clausula C_i sea cierta es necesario que al menos una de las variables λ_i sean un *si*. Si dos o más de las λi son un *si*, entonces es trivial que las nuevas clausulas son ciertas sea cual sea el valor de la variable auxiliar x_1. Si sólo un λ_i es un *si*, las nuevas clausulas son ciertas siempre que la variable que corresponde al *si* de x_1 o $\bar{x}_1$ esté en la clausula en la que no se encuentre ese λ_i.

3. Si $C_i = (\lambda)$, reemplazamos C_i por $(\lambda + y + z)$, y si $C_i = \left(\lambda + \lambda^{'}\right)$, lo reemplazamos por $\left(\lambda + \lambda^{'} + y\right)$. Luego añadimos las clausulas

$$(\bar{z} + \alpha + \beta)\,.\,(\bar{z} + \bar{\alpha} + \beta)\,.\,\left(\bar{z} + \alpha + \bar{\beta}\right)\,.\,\left(\bar{z} + \bar{\alpha} + \bar{\beta}\right)$$
$$.\,(\bar{y} + \alpha + \beta)\,.\,\left(\bar{y} + \alpha + \bar{\beta}\right)\,.\,(\bar{y} + \bar{\alpha} + \beta)\,.\,\left(\bar{y} + \bar{\alpha} + \bar{\beta}\right)$$

a la fórmula, donde y, z, α y β son nuevas variables. Esta adición fuerza a que las variables z e y sean un *no* en cualquier asignación de verdad que satisfaga $F^{'}$. Por tanto las clausulas (λ) y $\left(\lambda + \lambda^{'}\right)$ son equivalentes en sus reemplazamientos.

Con los cambios que acabamos de hacer, hemos reemplazado todas las clausulas C_i por otras equivalentes con tres variables en cada clausula. Además hemos demostrado que la fórmula final $F^{'}$ es satisfactible si y solo si F lo es. Como la construcción de $F^{'}$ se puede realizar en tiempo polinomial, podemos decir que hemos descrito una transformación en tiempo polinomial de satisfacibilidad a 3-satisfacibilidad. Por lo tanto podemos afirmar que 3-*satisfacibilidad es NP-completo.*

□

2.4.2. Circuito hamiltoniano es NP-completo

Utilizando el resultado anterior vamos a demostrar que encontrar un circuito hamiltoniano es un problema NP-completo. Esto será la base para lograr nuestro objetivo principal, demostrar que el problema del viajante de comercio es un problema NP-completo.

Teorema 2.3. *Circuito hamiltoniano es NP-completo*

Demostración. Sabemos que el problema de encontrar un circuito hamiltoniano está en NP. Vamos a demostrar que 3-satisfacibilidad, que acabamos de demostrar que es un problema NP-completo, tiene una transformación polinómica a este problema. Con esto quedaría demostrado el teorema. Entonces dada una fórmula booleana F que contiene m clausulas $C_1, ..., C_m$ y que involucra a n variables $x_1, ..., x_n$, construimos un grafo $G = (V, E)$ tal que G contiene un circuito hamiltoniano si y solo si F es satisfactible. Para la realización de esta demostración, vamos a considerar las siguientes construcciones especiales que únicamente utilizaremos para esta demostración.

Vamos a considerar primero el grafo A que se muestra en la figura 2.2 (a). Supongamos que este grafo es un subgrafo de algún otro grafo G tal que:

1. Los únicos extremos de A son u, $u^{'}$, v y $v^{'}$.
2. G tiene un circuito hamiltoniano c.

Teniendo en cuenta los dos puntos anteriores podemos afirmar que c atraviesa A por alguno de los dos caminos mostrados en la figura 2.2 (b) y (c). Para ver esto, lo primero es decir que los ocho bordes verticales de A tienen que pertenecer siempre a c, ya que si no lo hicieran es imposible que el circuito c pueda pasar por los cuatro nodos z_1, z_2, z_3 y z_4. Además, cualquiera combinación de bordes horizontales que no sean los que se muestran en esas dos figuras no pueden ser parte de un circuito hamiltoniano. Con todo esto esto, podemos decir que realmente el subgrafo A se comporta como si se tratara de un par de bordes $[u, u^{'}]$ y $[v, v^{'}]$ de G. Si entramos en A por el nodo u tenemos que salir obligatoriamente por el nodo $u^{'}$, y si lo hacemos por el nodo v saldremos por el nodo $v^{'}$, con la restricción adicional de que un circuito hamiltoniano en G debe atravesar obligatoriamente uno de ellos. Para representar el subgrafo A lo haremos como se muestra en la figura 2.2 (d).

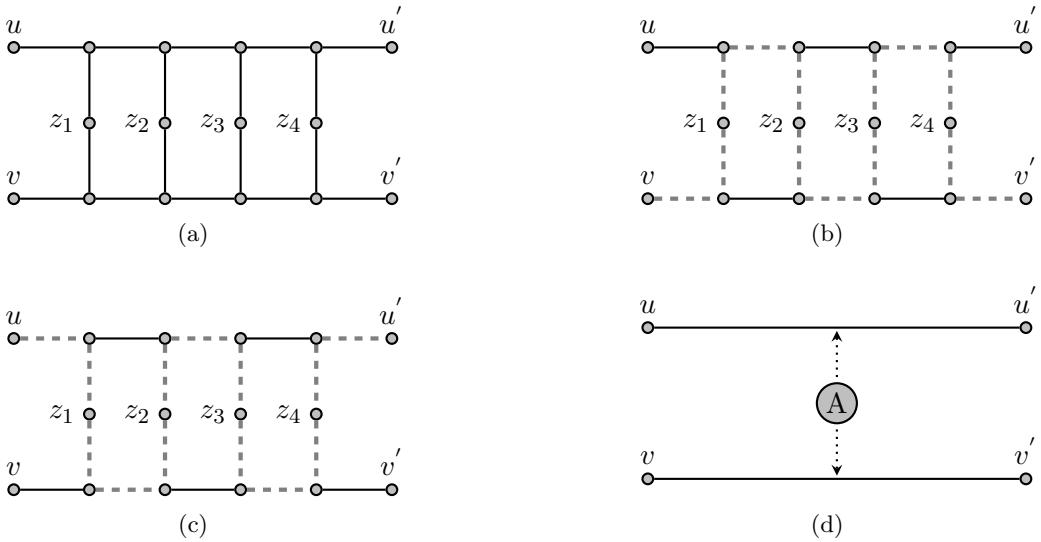


Figura 2.2: Subgrafo A

El grafo B que se muestra en la figura 2.3 (a) tiene una propiedad similar. Si B es un subgrafo del grafo G, de tal manera que solo los únicos nodos de B en los que inciden aristas de G son los nodos u_1 y u_4. Si además G tiene un circuito hamiltoniano c, entonces c no puede atravesar a la vez las aristas $[u_1, u_2]$, $[u_2, u_3]$ y $[u_3, u_4]$. Por otra parte, cualquier subconjunto propio de estas aristas puede ser una parte de un circuito hamiltoniano de G a través de las configuraciones mostradas en las figuras 2.3 (a), (b), (c) y (d) (y más configuraciones que no se muestran en esta imagen). Para representar este subgrafo, lo haremos como se ve en la figura 2.3 (e).

El grafo $G = (V, E)$ consistirá principalmente en copias de los subgrafos A y B. Para cada una de las m clausulas $C_1, ..., C_m$; nosotros tenemos m copias del subgrafo B unidas en serie (ver la figura 2.4 para una ilustración de la construcción de G. Darse cuenta que estamos construyendo una transformación polinómica de $3 - SAT$ a circuito hamiltoniano y por tanto cada una de las clausulas de F ha de tener 3 variables). Además para cada variable x_i tenemos dos nodos v_i y w_i, y dos aristas que son la copia derecha de $[v_i, w_i]$ y su correspondiente copia izquierda. También tenemos las aristas $[w_i, v_{i+1}]$ para $i = 1, ..., n - 1$, y las aristas $[u_{11}, v_1]$ y $[u_{m4}, w_n]$ donde u_{ij} denota la i-ésima copia de u_j (ver la figura 2.4 donde están los nodos marcados).

Darse cuenta que únicamente utilizamos los parámetros m y n de nuestra fórmula en la construcción de G de momento. Puesto que G pretende captar la complejidad de la cuestión de satisfacibilidad para F, ahora tenemos que tener en cuenta la naturaleza exacta de las clausulas de finalización. Así conectamos (a través del subgrafo A, A-conector) la arista $[u_{ij}, u_{ij_{+1}}]$ con la copia izquierda de $[v_k, w_k]$ en el caso de que la j-ésima variable de C_i sea x_k y con la copia derecha de $[v_k, w_k]$ si es $\bar{x}_k$ (ver la figura 2.4). Con esto tenemos completa la construcción del grafo G.

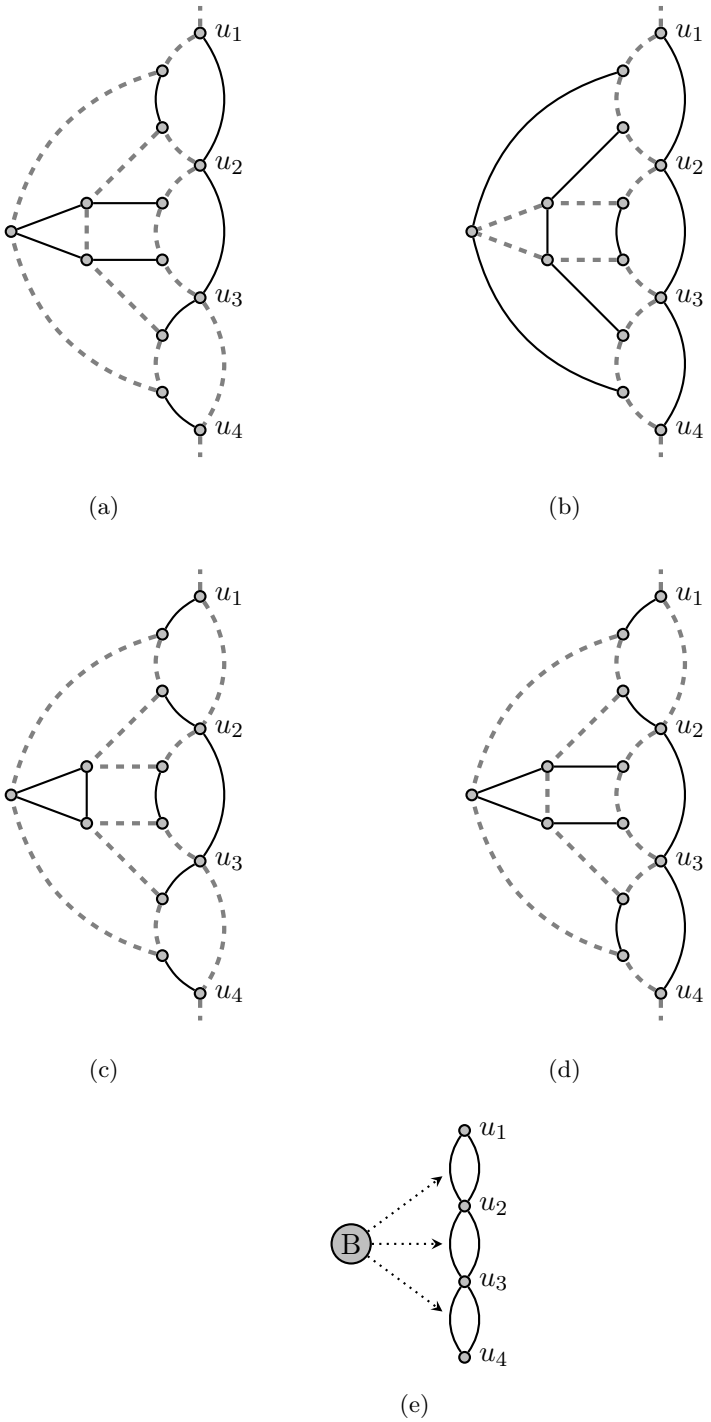


Figura 2.3: Subgrafo B

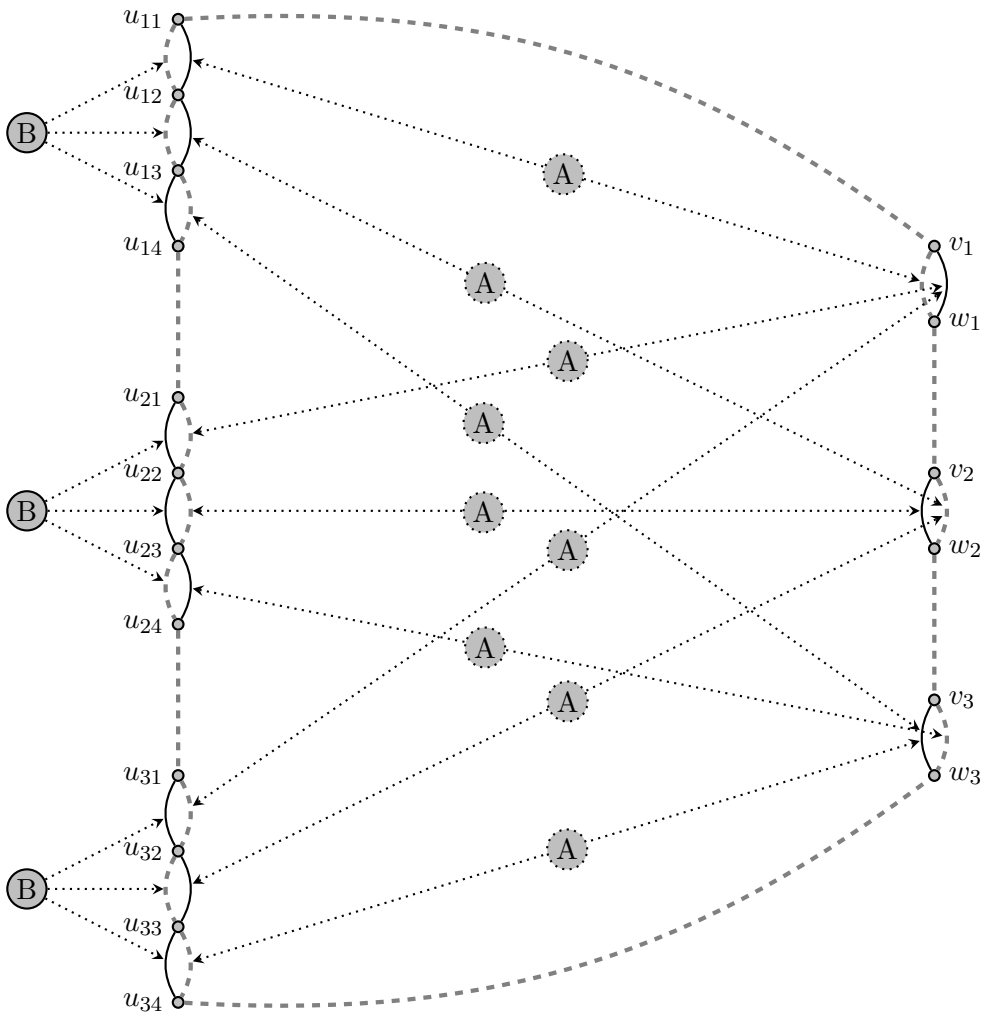


Figura 2.4: Ejemplo de un circuito hamiltoniano en el grafo G con $m = 3$ y $n = 3$.

Para este ejemplo la función F quedaría de la siguiente forma:

$$F = (x_1 + \bar{x}_2 + x_3).(\bar{x}_1 + x_2 + \bar{x}_3).(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)$$

El circuito hamiltoniano que se nos muestra corresponde a la siguiente asignación de verdad:

$$t(x_1) = si;\ t(x_2) = no \text{ y } t(x_3) = no$$

Ahora vamos a argumentar que la construcción de G descrita por F anteriormente es tal que G tiene un circuito hamiltoniano si y solo si F es satisfactible. Para la dirección *solo si* de la afirmación suponemos que G tiene un circuito hamiltoniano c. c tiene que tener una estructura especial: atraviesa $[u_{11}, v_1]$ y luego todos los nodos v y w de arriba hacia abajo, eligiendo una de las copias de $[v_i, w_i]$ para cada $i = 1, ..., n$. Luego atraviesa $[w_n, u_{m4}]$ y finalmente atraviesa todas las copias de B de abajo hacia arriba. Darse cuenta que el hecho de que c elija la copia izquierda de $[v_i, w_i]$ significa que x_i toma el valor *si*. Por otro lado si la copia elegida es la derecha significa que la variable x_i toma el valor *no*. La función resultado es una asignación de verdad válida porque c debe atravesar exactamente una de las dos copias. Las aristas $[u_{ij}, u_{ij_{+1}}]$ para cada clausula C_i se comporta de forma similar; son atravesadas si y solo si la copia correspondiente de la arista $[v_k, w_k]$ no lo es, es decir, si la correspondiente variable es un *no*. Por otra parte, como las aristas $[u_{ij}, u_{ij_{+1}}]$ son partes de un grafo B no pueden ser las tres atravesadas por c. Equivalentemente la correspondiente clausula C_i es satisfecha por la correspondiente asignación de verdad. Además, como esto es válido para todas las clausulas C_i seguimos que todas las clausulas lo verifican y por tanto F es satisfactible.

Para la implicación *si* suponemos que F es satisfactible para alguna asignación de verdad t. Está claro que podemos construir un circuito hamiltoniano para G simplemente siguiendo las reglas anteriores, es decir, atravesar la copia izquierda de $[w_i, v_i]$ si y solo si x_i es un *si* mediante t, y atravesar la arista $[u_{ij}, u_{ij_{+1}}]$ si y solo si la j-ésima variable de C_i es un *no* mediante t. Esto siempre será posible sin atravesar las tres aristas $[u_{ij}, u_{ij_{+1}}]$ para cualquier clausula ya que se supone que t satisface F.

Hemos demostrado que existe una transformación polinómica de 3-satisfacibilidad a circuito hamiltoniano, es decir, 3-satisfacibilidad se transforma polinómicamente en circuito hamiltoniano. Además, como vimos en un resultado anterior que 3-satisfacibilidad es NP-completo, podemos afirmar que el problema de encontrar un circuito hamiltoniano es NP-completo.

□

2.4.3. TSP es NP-completo

Con todo los datos que tenemos, ya podemos enunciar y demostrar el resultado central de este trabajo, el problema del viajante de comercio es NP-completo.

Corolario 2.1. *El problema del viajante de comercio (TSP) es NP-completo*

Demostración. El problema de encontrar un circuito hamiltoniano, que acabamos de demostrar que es un problema que pertenece a la clase NP-completo, es un caso particular del problema del viajante de comercio. Sea $G = (V, E)$ un grafo cualquiera, construimos un ejemplo del TSP

con $|V|$ ciudades, siendo $d_{ij} = 1$ si $[v_i, v_j] \in E$, y $d_{ij} = 2$ en otro caso. Supongamos que el presupuesto L es igual al $|V|$. Es inmediato ver que existe un circuito menor o igual que L si y solo si existe un circuito hamiltoniano en G. Por tanto la versión de reconocimiento del problema del viajante es NP-completo.

□

Capítulo 3
Diseño de algoritmos para el TSP

Contenidos

En este capítulo del trabajo, vamos a definir y explicar el funcionamiento de algunos algoritmos que podemos utilizar a la hora de intentar calcular la solución para un problema del viajante de comercio. Algunos de estos algoritmos los programaremos nosotros mismos.

La información que utilizamos para la realización de este capítulo está tomada en una gran parte de lo apuntes de técnicas de optimización da xestión [2] realizados por el profesor Julio González Díaz.

3.1. Algoritmo de programación lineal y entera; ramificación y acotación.

La primera forma de atacar el problema del viajante es utilizar un algoritmo de programación lineal y entera. Este es un algoritmo exacto pero con un coste de tiempo muy elevado. En muchos casos, la utilización de este programa es inviable debido a que el coste de tiempo es muy alto a pesar de que llegue siempre a obtener la solución del problema.

Decimos que es la primera forma de atacar el problema ya que es la manera más intuitiva de hacerlo. Lo único que deberíamos de hacer es coger la fórmula vista en el capítulo 2, que nos dice como se podía expresar el TSP como un problema de programación lineal entera y programarlo. Ya hablamos en ese mismo capítulo de algunas propiedades y justificamos que estaba bien definida. Recordamos que la fórmula es la siguiente:

$$\text{minimizar} \quad \sum_{i,j=0}^{n} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{sujeto a} \quad (1) \quad & \sum_{i=0}^{n} x_{ij} = 1 && j = 0, ..., n \\
(2) \quad & \sum_{j=0}^{n} x_{ij} = 1 && i = 0, ..., n \\
(3) \quad & u_i - u_j + nx_{ij} \leq n - 1 && 1 \leq i \neq j \leq n \\
& 0 \leq x_{ij} \leq 1, && x_{ij} \in Z, \forall i, j
\end{aligned}$$

3.2. Algoritmo del árbol.

El siguiente algoritmo que vamos a tratar es el del *árbol*. Este es un algoritmo aproximado que puede llegar a cometer, en el peor de los casos, errores de hasta el 100 %. Aunque pueda cometer este error, este algoritmo no tiene el problema del alto coste de tiempo que tiene el problema de programación lineal y entera.

Antes de explicar en que consiste el algoritmo del árbol vamos a hacer referencia a una serie de conceptos que son imprescindibles a la hora de entender este algoritmo.

3.2.1. Árbol de expansión mínima

El problema del árbol de extensión mínima es un problema con muchos usos tanto desde el punto de vista teóricos como desde el punto de vista práctico práctico. Este problema se define sobre grafos no orientados como podemos ver más adelante.

Una *cadena* en un grafo no orientado G es un conjunto de aristas dentro del mismo. De la misma forma, un grafo se dice que es *conexo* si para cada par de vértices existe una cadena que los une. Un grafo se dice que es un *árbol* si es conexo y no tiene ciclos.

Definición 3.1. Se dice que un árbol definido en un grafo $G = (N, M)$ es un *árbol de expansión* si contiene a todos los nodos de G.

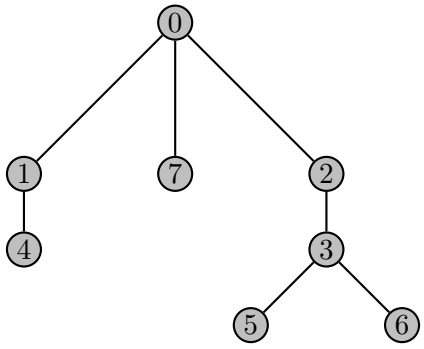


Figura 3.1: Un árbol.

Ahora, con estas definiciones anteriores ya podemos definir el problema del árbol de expansión mínima.

Definición 3.2. El *problema del árbol de expansión mínima* en un grafo no orientado G con costes c consiste en encontrar un árbol de espansión en G de coste mínimo.

Para encontrar un algoritmo que soluciones este problema necesitamos un resultado que veremos ahora. Dado un grafo no orientado $G(N, M)$, un *bosque* es una colección de árboles en G que no tienen ningún nodo en común. Diremos que es un *bosque de expansión* de G si todo nodo de G pertenece a algún árbol del bosque. El algoritmo que utilizaremos para llegar a una solución de este problema es consecuencia inmediata del siguiente resultado:

Proposición 3.1. *Tenemos un grafo no dirigido G, unos costes c y B un bosque de expansión de G. Fijemos un árbol A en B y supongamos que el arco (i, j) es el arco de mínimo coste de entre aquellos arcos que unen un nodo en A con algún nodo fuera de A. Entonces, de entre todos los árboles de expansión de G que contienen al bosque B habrá uno de mínimo coste que contenga al arco (i, j).*

Con este resultado podemos construir el siguiente algoritmo:

ALGORITMO PARA EL PROBLEMA DEL ÁRBOL DE EXPANSIÓN MÍNIMA

PASO 1

Elijamos un nodo arbitrario i y definamos el conjunto $X = \{i\}$.

PASO 2

Busquemos el nodo de $N \backslash X$ más cercano a X, digamos j.

- Añadamos el nodo j a X.
- Añadamos a nuestro árbol el arco correspondiente (el arco que une j a X a mínimo coste).
- Repetir el PASO 2 mientras $X \neq N$.

3.2.2. Cadena euleriana

Antes de nada damos la definición de cadena de euler:

Definición 3.3. Una *cadena de euler* en un grafo G es una cadena cerrada que contiene a todos los nodos al menos una vez y a todas las aristas una única vez.

Un grafo que contiene una cadena de euler se llama *grafo euleriano*. Un *multigrafo* $G = (N, M)$ es un grafo donde el conjunto M puede tener aristas repetidas. Todos los conceptos definidos para grafos podemos generalizarlos para multigrafos, así un multigrafo es *euleriano* si existe una cadena de euler.

Proposición 3.2. *Un multigrafo $G = (N, M)$ es euleriano sí y sólo sí:*

(I) G es conexo.

(II) Todos los nodos de N tienen grado par.

Demostración. Es inmediato que estad dos condiciones son necesarias. Si el grafo no es conexo no hay ninguna cadena cerrada que recorra todos los nodos y, además, en cualquier cadena cerrada tendremos que el número de arcos incidentes en cada nodo es par.

Para la suficiencia probaremos por inducción en el número de aristas del grafo. Un multigrafo con 0 aristas verificando (I) y (II) tendrá un único nodo y será euleriano. Supongamos ahora que tenemos un multigrafo G verificando (I) y (II) y tal que todos los multigrafos con menos aristas que G veirficando (I) y (II) son eulerianos. Elijamos ahora un nodo i de G y empecemos en él una cadena cerrada de aristas de G, que nunca repita dos veces la misma arista; claramente, (I) y (II) nos asegura que esto será posible. Ahora, eliminemos de G las aristas de esta cadena. Esto dará lugar a unas cuantas componentes conexas (subgrafos conexos del grafo original). Además, como a cada nodo le hemos eliminado una cantidad par de aristas en él, cada una de estas componentes conexas verificará (I) y (II), y la hipótesis de inducción nos dice que cada una de ellas será euleriana. Ahora podemos crear una cadena euleriana en el grafo original G sin más que concatenar las distintas cadenas eulerianas con la cadena de partida. □

Basándonos en el resultado anterior utilizamos el siguiente algoritmo a la hora de encontrar una solución al problema de encontrar una cadena euleriana.

ALGORITMO PARA ENCONTRAR UNA CADENA EULERIANA
(en un multigrafo verificando (I) y (II))

PASO 1:
Partimos de un nodo cualquiera $i \in N$.

PASO 2:
Construimos una cadena cerrada que empiece en i.

PASO 3:
Eliminamos del grafo las aristas de la cadena y construimos una nueva cadena cerrada para cada una de las componentes conexas de las que conste el nuevo grafo.

PASO 4:
Seguimos haciendo esto hasta que todos las aristas hayan sido añadidas a alguna cadena cerrada.

PASO 5:
Concatenamos todas las cadenas y obtenemos una cadena euleriana.

Vamos a ver ahora un resultado que nos será muy útil posteriormente en el algoritmo del árbol.

Proposición 3.3. *Sea $G = (N, M)$ un grafo completo con costes c verificando la desigualdad triangular. Sea $\bar{G} = (N, \bar{M})$ un multigrafo euleriano (con costes dados también por c). Entonces, existe un circuito hamiltoniano τ en G con coste asociado $c(\tau) \leq C(\bar{G})$. Además, podremos encontrar dicho circuito en tiempo $O(m)$.*

Demostración. Las hipótesis nos aseguran que $\bar{G}$ tiene una cadena euleriana ω. Como ω visita todos los nodos al menos una vez, podemos escribir $\omega = (\omega_0, i_1, \omega_1, i_2, ..., i_n, \omega_n)$ donde $\tau := (i_1, ..., i_n, i_1)$ es un circuito hamiltoniano (embebido en ω) y ω_0, $\omega 1$,..., ωn son cadenas (posiblemente vacías). La desigualdad triangular nos asegura que, para todo k, $c(i_k, i_{k+1})$ será menor o igual que la suma de los costes de las aristas en i_k, ω_k, i_{k+1}. Por tanto, la longitud del circuito τ, $c(\tau)$, será menor o igual que la longitud de la cadena ω, que será exactamente $C(\bar{G})$.

Por último, sabemos que la cadena euleriana la podemos encontrar en un tiempo $O(m)$, y la identificación de un circuito hamiltoniano dentro de ella es una tarea inmediata que se puede realizar de tal manera que el tiempo total sea $O(m)$.

□

3.2.3. El algoritmo del árbol

Una vez visto los resultados anteriores procedemos a plantear un algoritmo que nos será útil para resolver el problema del viajante. Este algoritmo, *algoritmo del árbol* permite calcular un circuito hamiltoniano en un grafo G con coste c. El algoritmo es el siguiente:

> ALGORITMO DEL ÁRBOL
>
> PASO 1
> Encontrar un árbol de expansión mínima T.
>
> PASO 2
> Crear un multigrafo $\bar{G}$ duplicando todas las aristas del árbol T.
>
> PASO 3
> Encontrar una cadena euleriana de $\bar{G}$ y un circuito hamiltoniano τ embebido en ella.

Con este algoritmo resolveremos una de la clase más importante de problemas del viajante, *los problemas del viajante métricos.*Trabajaremos con grafos no orientado, es decir, con $c_{ij} = c_{ji}$. Este tipo de problemas se caracterizan por dos cosas:

- El grafo $G = (N, M)$ es completo. Para todo par de nodos $i, j \in N$, $(i, j) \in M$.
- Desigualdad triangular. Dados tres nodos i, j y k tenemos que $c_{ij} \leq c_{ik} + c_{kj}$

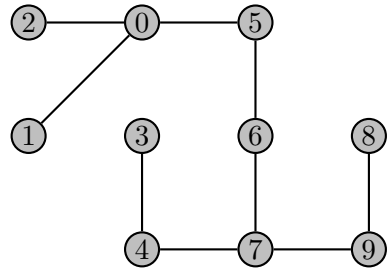
Mediante el siguiente resultado vamos a probar que el algoritmo del árbol está bien definido, es decir encuentra un circuito hamiltoniano y además es una 2-aproximación para el problema del viajante métrico.

Proposición 3.4. *El algoritmo del árbol es un algoritmo* 2*-aproximado del problema del viajante métrico (simétrico).*

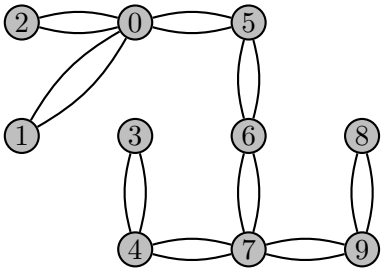
Demostración. El multigrafo $\bar{G}$ es conexo, ya que contiene a un árbol de expansión mínima. Además, todos sus nodos tienen grado par, ya que se ha obtenido duplicando todas las aristas de un árbol. Por tanto, la Proposición 3.2 nos asegura que $\bar{G}$ es euleriano. Entonces, podremos encontrar una cadena euleriana y un circuito hamiltoniano τ embebido en ella.

Probemos ahora la cota para el error. Denotemos por $\hat{c}$ el coste del circuito hamiltoniano de coste mínimo. Queremos probar que $c(\tau) \leq 2\hat{c}$. La Proposición 3.3 nos asegura que $c(\tau) \leq C(\bar{G})$. Claramente, $C(\bar{G}) = 2c(T)$, donde $c(T)$ es el coste del árbol T. Pero $c(T) \leq \hat{c}$, ya que cualquier circuito hamiltoniano puede transformarse en un árbol de expansión sin más que eliminar una arista cualquiera. Por tanto, $c(\tau) \leq C(\bar{G}) = 2c(T) \leq 2\hat{C}$. □

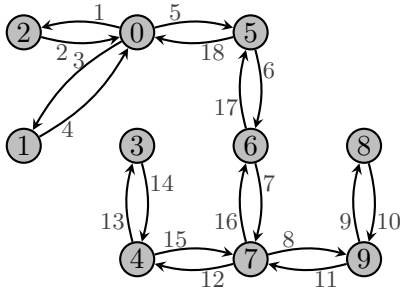
En la Figura 3.2 podemos ver una ilustración del algoritmo del árbol mediante un ejemplo partiendo de un árbol de expansión mínima. No pondremos los costes para simplificar el ejemplo.



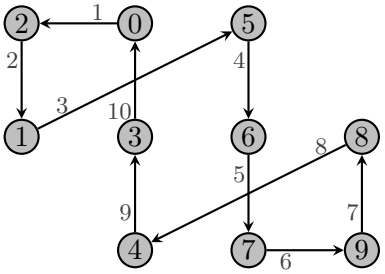
(a) Paso (I) Árbol de expansión mínima.



(b) Paso (II) Creando el multigrafo.



(c) Paso (III) Cadena de Euler:
$(\underline{0}, \underline{2}, 0, \underline{1}, 0, \underline{5}, \underline{6}, \underline{7}, \underline{9}, \underline{8}, 9, 7, \underline{4}, \underline{3}, 4, 7, 6, 5, \underline{0})$.



(d) Paso (III) Circuito hamiltoniano:
$(0, 2, 1, 5, 6, 7, 9, 8, 4, 3, 0)$.

Figura 3.2: Ilustración del algoritmo del árbol.

3.3. Algoritmo de Christofides.

De la misma forma que el algoritmo del árbol, el algoritmo de Christofides es un algoritmo aproximado que puede cometer un error de hasta el 50 %.

Antes de meternos a explicar el algoritmo de Christofides vamos a plantear un problema de gran importancia para el desarrollo del algoritmo.

3.3.1. Problema de emparejamiento

Definimos el problema de emparejamiento como sigue:

Definición 3.4. El *problema del emparejamiento* asociado a un grafo $G = (N, M)$ con un número par de nodos y unos costes c, consiste en emparejar cada nodo i de N con un y sólo un nodo j de N a coste mínimo. Este problema es una generalización del problema de asignación donde los conjuntos de origen y destino no están fijados de antemano.

El problema de emparejamiento admite una formulación, como un problema de programación lineal y entera, muy sencilla. Esta es como sigue:

$$\begin{aligned} \text{minimizar} \quad & \sum_{k \in M} c_k f_k \\ \text{sujeto a} \quad & \sum_{k \in M, i \in k} f_k = 1 \qquad i \in N \\ & f_k \in \{0, 1\} \qquad k \in M \end{aligned}$$

Esto quiere decir que cada arco (i, j) tomará los valores 0 o 1 para indicar si los nodos i y j han sido emparejados. Como estamos en un grafo no orientado, (i, j) y (j, i). La primera restricción nos dice que exactamente un arco será incidente en cada nodo.

Destacar que el problema de emparejamiento es un problema polinomial y los algoritmos más rápidos para resolverlo tienen una velocidad de $O(n^3)$.

3.3.2. Algoritmo de Christofides

El algoritmo de *Christofides* fue definido por *Christofides* en 1976 y en estos momentos esta considerado como el mejor algoritmo aproximado polinomial para el problema del viajante métrico.

Utilizando el siguiente resultado podremos afirmar, mediante una proposición, que el algoritmo de Christofides que vamos a ver ahora, está bien definido.

Lema 3.1. *Todo grafo $G = (N, M)$ tiene un número par de nodos con grado impar.*

Demostración. Claramente, la suma de los grados de todos los nodos, K, ha de ser par, ya que cada arista se suma dos veces. Además, la suma de los grados de todos los nodos de grado par, K_p será par. Entonces, la suma de los grados de todos los nodos de grado impar, K_i, será también par, ya que $K_i = K - K_p$ es la resta de dos números pares. Pero K_i es una suma de números impares y la única forma de que sea par es que haya una cantidad par de sumandos. □

El algoritmo de Christofides es muy semejante al algoritmo del árbol. El único cambio que encontramos con respecto a este último está en el Paso 2, donde se cambia el paso de duplicación del árbol de expansión mínima por la resolución de un problema de emparejamiento. Un *problema de emparejamiento* asociado a un grafo $G = (N, M)$ con un número par de nodos y unos costes c, consiste en emparejar cada nodo i de N con un y sólo un nodo j de N a coste mínimo.

El algoritmo de Christofides es el siguiente:

ALGORITMO DE CHRISTOFIDES

PASO 1
Encontrar un árbol de expansión mínima T.

PASO 2
Buscar los nodos de grado impar en T y encontrar el emparejamiento de mínimo coste entre ellos, E. Definir el multigrafo $\bar{G} = (N, \bar{M})$, donde $\bar{M}$ está formado por las aristas de T más las aristas de E.

PASO 3
Encontrar una cadena euleriana de $\bar{G}$ y un circuito hamiltoniano τ embebido en ella.

El algoritmo de Christofides está bien definido y además es una 1.5-aproximación del problema del viajante. Esto lo demostraremos mediante la siguiente proposición.

Proposición 3.5. *El algoritmo de Christofides es un algoritmo* 1.5*-aproximado del problema del viajante métrico (simétrico).*

Demostración. El Lemma 3.1 nos asegura que el número de nodos con grado impar en T es par, con lo que el problema de emparejamiento está bien definido. El multigrafo $\bar{G}$ es conexo, ya que contiene a un árbol de expansión mínima. Además, todos sus nodos tienen grado par: si un nodo tenía grado par en T, sigue teniendo el mismo grado y si tenía grado impar, ahora hay una arista más incidiendo en él, con lo que tendrá grado par. Por tanto, la Proposición 3.2, nos asegura que $\bar{G}$ es euleriano. Entonces, podremos encontrar una cadena euleriana y un circuito hamiltoniano τ embebido en ella.

Probemos ahora la cota para el error. Denotemos por $\hat{\tau}$ el circuito hamiltoniano de coste mínimo y por $\hat{c}$ el coste asociado. Queremos probar que $c(\tau) \leq 1.5 \cdot \hat{c}$. Claramente, $c(\tau) \leq C(\bar{G}) = c(T) + c(E)$ y $c(T) \leq \hat{c}$. Denotemos por $\{i_1, i_2, \ldots, i_{2l}\}$ el conjunto de nodos de grado impar en T numerados según su orden de aparición en $\hat{\tau}$. Podemos escribir $\hat{\tau} = (\hat{\tau}_0, i_1, \hat{\tau}_1, i_2, \ldots, i_{2l}, \hat{\tau}_{2l})$. Consideremos ahora los siguientes emparejamientos de los nodos de grado impar:

$$E_1 = \{(i_1, i_2), (i_3, i_4), \ldots, (i_{2l-1}, i_{2l})\} \quad \text{y} \quad E_2 = \{(i_2, i_3), (i_4, i_5), \ldots, (i_{2l}, i_2)\}.$$

Si ahora "entrelazamos" estos dos emparejamientos tenemos la cadena cerrada $(i_1, i_2, i_3, \ldots, i_{2l})$

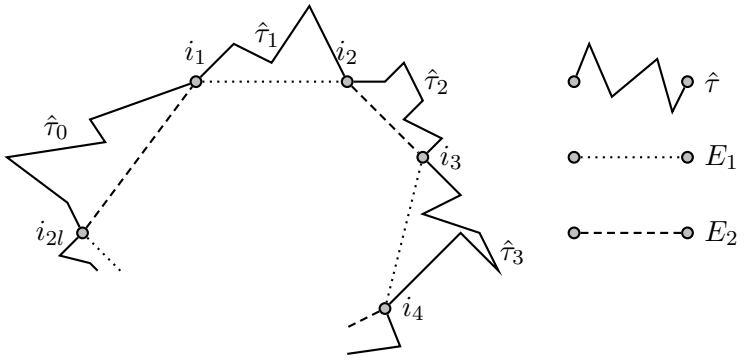


Figura 3.3: Entrelazando los dos emparejamientos de la demostración de la Proposición 3.5.

y la desigualdad triangular nos asegura entonces que $\hat{c} = c(\hat{\tau}) \geq c(E_1) + c(E_2)$ (Figura 3.3). Pero el emparejamiento E construido por el algoritmo es el de mínimo coste, con lo que $\hat{c} \geq c(E_1) + c(E_2) \geq 2c(E)$, es decir, $c(E) \leq \hat{c}/2$. Con lo que tenemos,

$$c(\tau) \leq C(\bar{G}) = c(T) + c(E) \leq \hat{c} + \frac{\hat{c}}{2} = \frac{3}{2}\hat{c}. \qquad \square$$

En la Figura 3.4 presentamos una ilustración del algoritmo de Christofides. Igual que en el caso del algoritmo del árbol no incluimos los costes en el ejemplo.

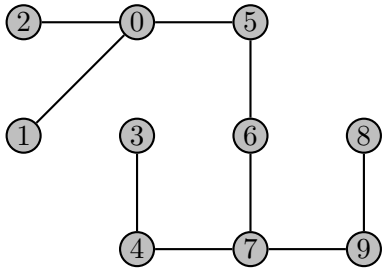
Ahora vamos a presentar dos algoritmos heurísticos. Hay que darse cuenta de que la precisión de estos algoritmos puede variar según están calibradas de una forma u otra las diferentes variables implicadas. Es más, una calibración adecuada a unos problemas puede no ser la ideal para otros con un número de nodos diferente. Debido a esto debemos de ser prudentes a la hora de analizar los valores obtenidos, como puede ser el tiempo, con estos algoritmos.

3.4. Algoritmo de las hormigas.

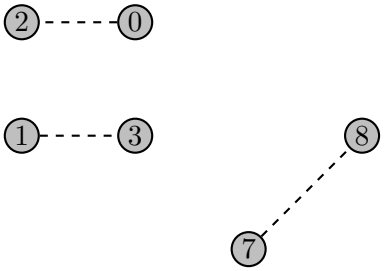
El algoritmo de optimización colonia de hormigas (Ant Colony Optimization, ACO) es una técnica probabilística para solucionar problemas computacionales que pueden reducirse a buscar los mejores caminos o rutas en grafos.

Este algoritmo fue inicialmente propuesto por Marco Dorigo en 1992 en su tesis de doctorado. El primer algoritmo surgió con el objetivo de buscar el camino óptimo en un grafo basado en el comportamiento de las hormigas cuando están buscando un camino entre la colonia y una fuente de alimentos. La idea original se fue desarrollando para resolver una amplia clase de problemas numéricos. Como resultado de esto, han surgido gran cantidad de problemas nuevos basándose en diversos aspectos del comportamiento de las hormigas.

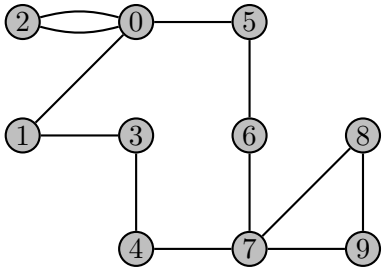
En nuestro mundo natural, las hormigas vagan inicialmente de manera aleatoria al azar hasta encontrar comida. Una vez encontrada regresan a su colonia dejando un rastro de feromonas por el camino. Si otras hormigas encuentran dicho rastro, es probable que estas no sigan caminando aleatoriamente. Es probable que sigan este rastro de feromonas, regresando y reforzándolo si estas encuentran comida.



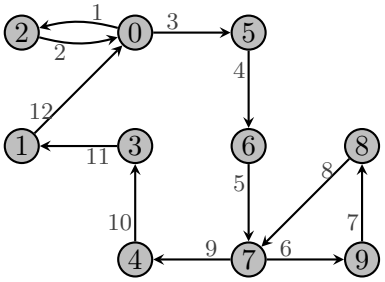
(a) Paso (I) Árbol de expansión mínima.



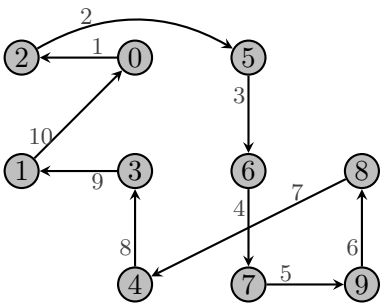
(b) Paso (II) Nodos de grado impar y emparejamiento óptimo.



(c) Paso (II) Creando el multigrafo.



(d) Paso (III) Cadena de Euler: $(\underline{0}, \underline{2}, 0, \underline{5}, \underline{6}, \underline{7}, \underline{9}, \underline{8}, 7, \underline{4}, \underline{3}, \underline{1}, \underline{0})$.



(e) Paso (III) Circuito hamiltoniano: $(0, 2, 5, 6, 7, 9, 8, 4, 3, 1, 0)$.

Figura 3.4: Ilustración del algoritmo de Christofides.

Figura 3.5: El trabajo de las hormigas.

Sin embargo, con el paso del tiempo el rastro de feromonas comienza a evaporarse, reduciéndose así su fuerza de atracción. Cuanto más tiempo le lleve a una hormiga recorrer el camino, más tiempo tienen las feromonas para evaporarse. Un camino corto en comparación, es recorrido más frecuentemente y por tanto la densidad de feromonas se hace más grande en los caminos cortos que en los largos.

Por tanto, cuando una hormiga encuentra un buen camino entre la colonia y la fuente de comida, hay más posibilidades de que otras hormigas de la colonia tomen el mismo camino. Con una retroalimentación positiva se conduce finalmente a todas las hormigas a un solo camino. La idea del algoritmo de las hormigas es imitar este comportamiento con "hormigas simuladas" caminando a través de un grafo que representa el problema en cuestión.

En el algoritmo de las hormigas, una hormiga es simplemente un agente computacional. Este algoritmo construye iterativamente una solución para el problema en cuestión donde en cada iteración cada hormiga se mueve de un estado x a un estado y (en el caso del TSP, cada uno de estos estados son cada una de las ciudades que hay que visitar). Para una hormiga k, la probabilidad P_{xy}^k de moverse de un estado x a un estado y depende de la combinación de dos valores: el atractivo del movimiento, η_{xy} computado por alguna variable dada de antemano que nos da el de alguna forma forma la conveniencia de ir de un lado a otro, y el nivel de rastro del movimiento, τ_{xy} que nos dice de alguna forma la frecuencia con que se pasó por ese arco.

El nivel de rastro representa a posteriori una indicación de la conveniencia de ese movimiento. Los rastros son actualizados después de cada iteración después de que cada hormiga hiciera su recorrido, aumentando o disminuyendo los niveles según la recorrido fuera bueno o malo.

En general la k-ésima hormiga se mueve del estado x al estado y con la siguiente probabilidad:

$$P_{xy}^k = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum (\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}$$

donde τ_{xy} es la cantidad de feromonas que serán depositadas en la transición del estado x

al estado y, α es un parámetro para controlar la influencia de τ_{xy}. η_{xy} es la conveniencia del estado de transición xy (un conocimiento a priori, tipicamente $1/d_{xy}$, donde d es la distancia), y β es un parámetro para controlar la influencia de η_{xy}.

Cuando todas las hormigas han completado una solución, los rastros son actualizados por $\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \Delta\tau_{xy}^k$, donde ρ es el coeficiente de evaporación de feromonas y $\Delta\tau_{xy}^k$ es la canatidad de feromonas depositadas por la k-ésima hormiga. Tipicamente dado para el problema del viajante viene dado por:

$$\Delta\tau_{xy}^k = \begin{cases} Q/L_k & \text{si la } k\text{-ésima hormiga usa la arista } xy \text{ en ese circuito} \\ 0 & \text{en otro caso} \end{cases}$$

donde L_k es el costo de la ruta de la k-ésima hormiga (en la mayoría de los casos es la longitud) y Q es una constante.

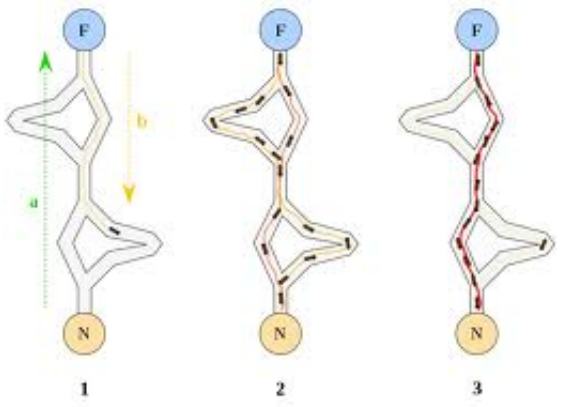


Figura 3.6: Funcionamiento.

Los algoritmos de optimización de colonias de hormigas son aplicados en muchos algoritmos de optimización combinatorios. Han sido usados para producir soluciones bastante cercanas a las soluciones óptimas del problema del viajante de comercio. Este algoritmo es de interes en los campos de enrutamiento de redes y en los sistemas de transporte urbanos entre otros.

El primer algoritmo de colonia de hormigas, llamado *Ant System*, tenía el objetivo de resolver el problema del viajante, que consiste en encontrar el circuito hamiltoniano más corto en un grafo completo.

En este trabajo vamos a programar el código de este algoritmo utilizando el programa *R*, que lo presentaremos en uno de los anexos.

3.5. Algoritmo Lin-Kernighan.

Aunque la heurística de *Lin-Kernighan*, Lin y Kernighan (1973), para el problema del viajante de comercio (TSP) data del año 1973, es considerada todavía la mejor heurística disponible para este problema.

Como dijimos, el algoritmo de Lin-Kernighan es un algoritmo heurístico que se conoce también con el nombre de *algritmo de búsqueda variable*, ya que permite ejecutar varias transformaciones consecutivas a la hora de mejorar la solución actual.

Antes de meternos a explicar este algoritmo daremos un par de conceptos que nos ayudarán a la hora de entenderlo mejor.

Definición 3.5. Supongamos que tenemos un circuito hamiltoniano con dos aristas $\{i, j\}$ y $\{k, l\}$ incidentes en cuatro nodos distintos (es decir, $|\{i, j, k, l\}| = 4$) y tales que el tour visita estos nodos en el orden i, j, k, l. Entonces, el 2-*intercambio* asociado con las aristas $\{i, j\}$ y $\{k, l\}$ consiste en reemplazarlas por las aristas $\{i, k\}$ y $\{j, l\}$.

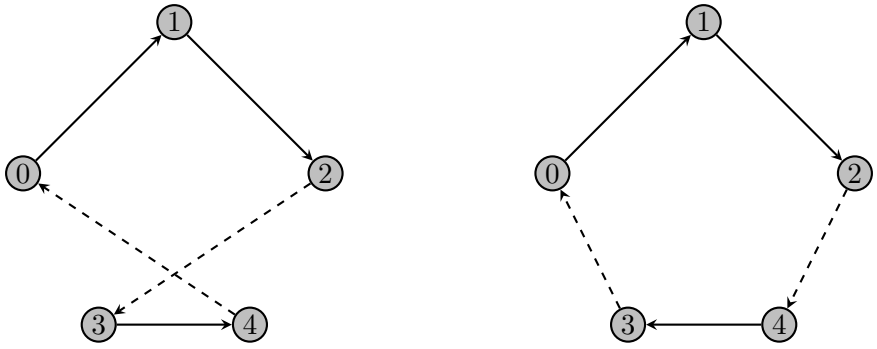


Figura 3.7: Ejemplo de un 2-intercambio en un circuito hamiltoniano.

Tener en cuenta que un 2-intercambio en un circuito hamiltoniano siempre dará lugar a un nuevo circuito hamiltoniano (Figura 3.7). La diferencia de coste entre un circuito original y el nuevo circuito se corresponderá con la diferencia de costes entre las aristas intercambiadas.

La definición anterior de 2-intercambio se puede modificar convenientemente para definir para definir k-intercambios. Los algoritmos (heurísticos) de búsqueda local basados en k-intecambios se conocen como k-*opt*. En la práctica casi siempre se trabaja con 2-intercambios y 3-intercambios ya que está demostrado que para valores más altos de k el tiempo extra de computación no compensa en relación con la mejora de los resultados.

La idea del algoritmo de Lin-Kernighan es muy sencilla, es posible que un algoritmo como 2-opt se pare en un máximo local τ porque cualquier 2-intercambio incrementa el coste del circuito. Sin embargo, es posible que si realizásemos el menos malo de los 2-intercambios para pasar de τ a $\tau^{'}$ y después buscásemos el mejor 2-intercambio desde $\tau^{'}$, llegásemos a un circuito $\tau^{''}$ de coste más bajo que τ. Es decir, podríamos tener:

$$c(\tau^{''}) < c(\tau) < c(\tau^{'})$$

El algoritmo de Lin-Kernighan no es más que una modificación de los algoritmos 2-opt y 3-opt que permita también una pequeña búsqueda en vecindarios de mayor profundidad (la aplicación sucesiva de 2-intercambios será equivalente a algún k-intercambio para k suficientemente grande). Este algoritmo nos permite hacer cambios inicialmente costosos pero que a medio plazo resultan beneficiosos.

El funcionamiento del algoritmo de Lin-Kernighan basado en 2-intercambios para un problema del viajante con n nodos sería el siguiente:

ALGORITMO DE LIN-KERNIGHAN

PASO 1

Elegimos un circuito hamiltoniano τ.

PASO 2

A continuación:

a) Realizamos el mejor 2-intercambio desde τ (aunque esto incremente la longitud del circuito) obteniendo el circuito τ^1.

b) Realizamos el mejor 2-intercambio desde τ^1 que no involucre a ninguna de las aristas añadidas en este Paso 2 obteniendo el circuito τ^2.

... Terminamos el Paso 2 cuando no haya más 2-intecambios posibles porque ya no queda ninguna arista del circuito original τ. Definimos como $\hat{\tau}$ el circuito de coste más bajo de los circuitos τ^1, τ^2,...

PASO 3

Si $c(\hat{\tau}) < c(\tau)$, reemplazamos τ por $\hat{\tau}$ y repetimos el Paso 2. Si $c(\hat{\tau}) \geq c(\tau)$, el algoritmo termina ya que no hemos conseguido mejorar.

Fijarse en el Paso 2 del algoritmo que exige que una arista que fue añadida en algún momento de este paso pueda ser cambiada en futuros pasos de esta iteración, esto debido a que una de las motivaciones de este algoritmo es intentar evitar que este termine en óptimos locales. Ya que si no, se podría dar el caso de que el mejor 2-intercambio en el primer apartado del Paso 2 sea también el mejor pero a la inversa en el Paso 2 b). Con lo cual estaríamos estancados en un mismo circuito.

El algoritmo de Lin-Kernigan es uno de los mejores algoritmos para la resolución del problema del viajante y es ampliamente usado en la práctica. Aunque tiene el problema que, por mucho que nos esforcemos en no caiga en un máximo local no tenemos la seguridad completa de que no lo haga.

Capítulo 4
Comparativa de algoritmos

Contenidos

En este capítulo hemos creado de forma aleatoria una serie de problemas del viajante de comercio con diferente cantidad de nodos, que resolveremos mediante la utilización de diferentes algoritmos. Utilizaremos algoritmos heurísticos, que no tienen porque calcular la solución la solución exacta del problema, pero que nos dan una solución que puede ser óptima, y algoritmos que sí que nos calculan una solución exacta. Algunos de estos algoritmos fueron programados por nosotros, otros están implementados en la red y también tenemos otros cedidos por Julio González y Laura Calaza para la realización de este trabajo. Todos los valores obtenidos los recopilaremos en una tabla que podemos ver más adelante. Por último, analizaremos los valores obtenidos en la siguiente tabla, intentando comparar el funcionamiento de estos algoritmos.

Los valores que vamos a obtener de cada algoritmo serán el coste del mejor circuito y el tiempo de ejecución del programa para calcularlo, en cada uno de los ejemplos. Por otro lado, en la tabla tendremos en cuenta otros valores: en una columna tendremos el tiempo medio de ejecución de cada programa para la obtención de la solución para cada uno de los ejemplos teniendo en cuenta la cantidad de nodos de los mismos. En la otra columna tenemos el porcentaje de error cometido en el cálculo de la solución comparándolo con el óptimo. Tomaremos como valores óptimos los valores obtenidos por concorde que sabemos que los son.

Antes de nada queremos hacer unas consideraciones previas a analizar los valores obtenidos, ya que podemos llegar a conclusiones erróneas de no tenerlas en cuenta, como por ejemplo, el programa en el que están implementados los algoritmos puede hacer que se ejecuten en menor o mayor tiempo.

4.1. Consideraciones previas.

Las metodologias que utilizaremos son: el algoritmo de las hormigas, empleado con tres calibraciones del mismo e implementado por nosotros, presentamos el problema también como un problema de programación lineal y entera para luego resolverlo con la librería *lpSolveAPI*; el algoritmo del árbol y el algoritmo de Christofides que fueron cedidos por Julio González y Laura Calaza para la realización de este trabajo; y el algoritmo de Lin-Kernighan y Concorde que aplicamos de la red en la página de Neos: http://www.neos-server.org/neos/solvers/co:concorde/TSP.html.

Lo primero que queremos destacar es que tanto, el algoritmo de las hormigas, como el del árbol y el algoritmo de Christofides, están implementados en R, que es un lenguaje de programación interpretado y no compilado. Esto quiere decir que es más lento en la ejecución que otros compilados como por ejemplo C++. Por otro lado, tanto Concorde como Lin-Kernighan suponemos que son programas que están muy bien calibrados, que son muy eficientes, tener en cuenta que Concorde ahora mismo es el mejor programa que calcula una solución exacta para hallar una solución para este tipo de problemas, y Lin-Kernighan es el mejor heurístico para lo mismo. Considerando estos datos, tenemos que tener cuidado a la hora de comparar los tiempos de ejecución, ya que, por ejemplo si el algoritmo de las hormigas lo tuviésemos implementado en otro lenguaje de programación, podría variar el tiempo de ejecución. Lo que pretendemos decir con esto es que las conclusiones observadas, serán valoradas teniendo esto en cuenta.

Por otra parte, centrándonos en algoritmo de Christofides, debemos destacar que al resolver el problema de emparejamiento para nodos de grado impar, este en el programa que nosotros

utilizamos se plantea como un problema de programación lineal entera, que no es la forma más eficiente de hacerlo, lo que conlleva un coste de tiempo más alto en la ejecución del algoritmo.

El algoritmo de programación lineal y entera nos da una solución exacta del problema siempre que alcance una solución. Pero tiene un problema, ya que el coste de tiempo crece exponencialmente cuando le vamos añadiendo nodos al problema. En la realización de este trabajo sólo pudimos aplicar este algoritmo a la resolución de los ejemplos con 13 nodos, ya que en cuanto empezábamos a añadir nodos, el tiempo de resolución crecía de forma exponencial y en algunos casos el ordenador se bloqueaba. Esto mismo nos pasó con el algoritmo de Christofides sólo que para ejemplos con más de 40. Hasta ejemplos de este tamaño funciona relativamente bien, pero al aumentar su tamaño el tiempo de computación aumenta de tal manera que con los medios de los que disponemos nos es imposible obtener sus soluciones.

Decir también que las tres versiones del algoritmo de las hormigas utilizadas y que llamaremos Hormigas 1, Hormigas 2 y Hormigas 3 se diferencian en las siguientes variables, b que es la variable que da más peso a la carga de feromonas, *evap* una variable que nos dice la cantidad de feromona que se evapora después de cada iteración y *Qinit* que es valor inicial de la variable Q. Los valores los encontramos en el artículo titulado *On Optimal Parameters for Ant Colony Optimization algorithms* [3] y son los siguientes: Hormigas 1 con $b = 6$, $evap = 0,4$ y $Qinit = 0,2$, Hormigas 2 con $b = 10$, $evap = 0,4$ y $Qinit = 0,3$, y Hormigas 3 con $b = 12$, $evap = 0,4$ y $Qinit = 0,2$.

4.2. Tabla de precisión de soluciones y tiempos medios.

	13 nodos		40 nodos		80 nodos	
	Error	T. M.	Error	T. M.	Error	T. M.
Concorde	0	0,0053	0	0,0630	0	0,1898
Prog. Entera	0	8,003333	-	-	-	-
Árbol	0,1566	0,4640	0,3064	15,2856	0,3939	129,3264
Christofides	0,0685	0,4740	0,1356	32,5228	-	-
Hormigas 1	0,0129	23,8847	0,1244	549,3780	0,2137	3653,6328
Hormigas 2	0,0178	20,6127	0,0854	560,5410	0,1299	4021,8772
Hormigas 3	0,0179	20,5447	0,0792	518,7772	0,1180	4083,6882
Lin-Kernighan	0	0	0	0,0104	4.9094e-05	0,0342

Los huecos de la tabla que están cubiertos con guiones quiere decir que no fueron ejecutados para esos ejemplos, ya que el coste de tiempo era muy excesivo, incluso en muchos casos el ordenador se bloqueaba.

4.3. Análisis de los valores.

Una vez vistos los valores de la tabla, haremos un breve análisis de los mismos. Lo primero en lo que nos fijamos es en el tiempo medio de ejecución en el Concorde y Lin-Kernighan, que como avanzamos antes, el sus tiempo de ejecución son muy pequeños. Darse cuenta que como estamos tomando como valores óptimos los costes obtenidos con el concorde, sus columnas de

error lógicamente es 0, ya que la proporción de error cometido es 0. Lin-Kernighan, que no tendría porque dar la solución óptima, llega a ella casi siempre, sólo empieza a dar algún error pero muy muy pequeño en los ejemplos con 80 nodos, sólo falla y de manera muy pequeña en 4 de los 50 ejemplos tomados para 80 nodos.

Como vemos, utilizando programación lineal y entera llegamos a la solución óptima siempre, y para los ejemplos con 13 nodos con un tiempo relativamente bueno, aproximadamente 8 segundos. Si esto lo comparamos con el tiempo empleado para los mismos ejemplos por las tres versiones del algoritmo de las hormigas está muy bien ya que estos andan sobre los 20 segundos y además sin alcanzar la solución exacta, ya que comete un error aproximadamente del $1,8\,\%$ con respecto al óptimo.

Si ahora observamos los tres conjuntos de valores obtenidos por las tres variantes del algoritmo de las hormigas, vemos que funcionan de forma semejante. El tiempo de ejecución es el muy parecido y el porcentaje de error cometido con respecto al óptimo es también semejante. Quizás, donde encontramos una mayor diferencia es en los ejemplos con 80 nodos, donde *Hormigas* 1 es más rápido que los demás en aproximadamente 400 segundos de media, lo que pasa que esa rapidez le lleva a perder precisión en cuanto a la solución en comparación con *Hormigas* 2 y *Hormigas* 3, obteniendo aproximadamente un $10\,\%$ menos de error con respecto al óptimo.

Si observamos la fila perteneciente a los valores obtenidos aplicando el algoritmo de Christofides y los comparamos con los otros resultados obtenidos con el algoritmo del árbol y las tres variantes de las hormigas nos damos cuenta que realmente es el que obtiene unos resultados más compensados en relación al error cometido y al tiempo medio empleado. El error cometido por Christofides es menor que el cometido por el algoritmo del árbol y algo mayor al cometido por las variantes de las hormigas, sin embargo el tiempo de ejecución es significativamente menor que el empleado por las hormigas y algo mayor al empleado por el algoritmo del árbol.

Otra deducción que se aprecia al observar los datos es que al aumentar el número de nodos en los ejemplos, aumenta también el error cometido con respeto al óptimo.

Por último, destacar la comparación de resultados entre el algoritmo del árbol y las variantes del algoritmo de las hormigas. El tiempo de ejecución del algoritmo del árbol es incluso 20 veces más rápido que el de las hormigas pero por el contrario comete un error mayor a la hora de calcular la solución óptima, aproximadamente un $16\,\%$ más que las variantes de las hormigas. Ahora tendríamos que valorar, que nos interesa más un programa rápido que cometa un error aceptable pero más alto que un programa mucho más lento pero que su solución dista relativamente poco de la solución óptima.

Bibliografía

[1] Donald Davendra. "TRAVELING SALESMAN PROBLEM, THEORY AND APPLICATIONS". InTech (2010).

[2] Julio González Díaz. "Técnicas De Optimización Da Xestión (Curso 2011/2012)".

[3] Dorian Gaertner y Keith Clark. "On Optimal Paarameters for Ant Colony Optimization algorithms". Dept of Computing, Imperial College London.

[4] Christos H. Papadimitriou y Kenneth Steiglitz. "Combinatorial Optimization: Algorithms and Complexity". General Publishing Company (1998).

[5] Silvia A. Ramos. El problema del viajante; conceptos, variaciones y soluciones alternativas. Modelos y Optimización.

[6] A. Turing. On the complexity of algorithms, with an application to the entscheidungsproblem. En "Proceedings of the London Mathematical Society" (1936).

Apéndice A

Código R

Código del algoritmo de las hormigas:

```
#############################################################################################
##############################ALGORITMO DE LAS HORMIGAS#######################################
#############################################################################################

#Función mediante la que llamo al algoritmo
alg_hormigas <- function(D, b=12, margenmejora=100, evap=0.4, maxit=1000){


#######################################################
###############Definimos las variables################
#######################################################

a <- 1
numnodos <- nrow(D) #Guardo el número de nodos que tiene nuestro problema.
sumacoste <- 0 #Guardo el coste del recorrido de cada hormiga en cada iteración.
menorcoste <- 0 #Guardo el menor coste de los circuitos después de cada iteración.
nomejoro <- 0 #Guardo el número de it consecutivas en las que no mejoro mi resultado.
it <- 1 #En esta variable guardo el número de iteraciones que se van ejecutando.
Qinit<-0.2 #Variable de precisión para el cálculo de Q.
power<- 1 #Variable de precisión para controlar el peso de las feromonas.
tolmin <- 0.01 #Controla que la matriz C no tenga infinitos.
C <- 1/(D+tolmin)
mejorcoste <- sum(D)+5 #Variable que guarda el mejor coste que voy consiguiendo.
Q<-Qinit*min(D)*numnodos #Variable que controla el peso que le damos a las feromonas.

hor <- numnodos

#En esta matriz guardamos el peso de las feromonas.
R <- matrix(1,nrow=numnodos,ncol=numnodos)

#Matriz en la que guardo los circuitos en cada iteración.
circuitos <- matrix(nrow=numnodos,ncol=hor)

#Vector que guarda el coste del circuito en cada iteración.
costehor <- rep(0,hor)
```

```
mejorcircuito <- numeric(numnodos)

#####################################################
###############Programamos el código################
#####################################################

nodos<-1:numnodos

while((it < (maxit+1)) && nomejoro < margenmejora){

for(k in 1:hor){
visitados<-numeric()
visitados<-c(visitados,1)
      pendientes<-setdiff(nodos,visitados)
      sumacoste <- 0

while(length(pendientes)>0){
       actual<-visitados[length(visitados)]

       if(length(pendientes)>1){
       P<-C[actual,pendientes]^b*R[actual,pendientes]^a
       P<-P/sum(P)
       nuevo<-sample(pendientes,1,prob=P)
       }
       else nuevo<-pendientes[1]
       visitados<-c(visitados,nuevo)
       pendientes<-setdiff(nodos,visitados)

        sumacoste<-sumacoste+D[actual,nuevo]
       }

      circuitos[,k]<-visitados
costehor[k]<-sumacoste+D[nuevo,1]
      }

      menorcoste <- min(costehor[k])
      if (menorcoste < mejorcoste) {
      mejorcoste <- menorcoste
      nomejoro <- 0
      mejorcircuito <- circuitos[,which.min(costehor)]
}
    else nomejoro <- nomejoro+1
it <- it+1


   circuitosext<-rbind(circuitos,rep(1,hor))
   R<-(1-evap)*R
   for(i in 1:k){
      for(j in 1:numnodos){
      orig<-circuitosext[j,i]
      dest<-circuitosext[j+1,i]
      R[orig,dest]<-R[orig,dest]+Q/costehor[i]^power
```

```
      }
   }

}

#################################################################
#########################SOLUCIÓN################################
#################################################################

return(list(circuito=solucion,coste=mejorcoste))
}
```

Código del algoritmo de programación lineal y entera:

```
#############################################################################################
#################################ALGORITMO DE PLE############################################
#############################################################################################

#Función mediante la que llamo al algoritmo
alg_ple<-function(D,relax=0){

#####################################################
###############Programamos el código################
#####################################################

n<-nrow(D)

c<-c(as.vector(t(D)),rep(0,n))

b<-c(rep(1,n),rep(1,n),rep(n-2,(n-1)*(n-1)-(n-1)))

A1<-matrix(0,nrow=n,ncol=n*n+n)

aux<-c(1,rep(0,n-1))
aux<-rep(aux,n)
for(j in 1:n){
   A1[j,1:(n*n)]<-c(rep(0,j-1),aux[1:(length(aux)-j+1)])
}

A2<-matrix(0,nrow=n,ncol=n*n+n)

for(i in 1:n){
   A2[i,(((i-1)*n+1):(i*n))]<-1
}

A3<-matrix(0,nrow=(n-1)*(n-1)-(n-1),ncol=n*n+n)

fila<-0
for (i in 2:n){
   for(j in 2:n){
      if(i != j){
         fila<-fila+1
         uinit<-n*n
         A3[fila,uinit+i]<-1
         A3[fila,uinit+j]<--1
         A3[fila,(i-1)*n+j]<-n-1
      }
   }
}

A<-rbind(A1,A2,A3)

library(lpSolveAPI)

tipores<-c(rep("=",n),rep("=",n),rep("<=",nrow(A)-2*n))
```

```
viajante<-make.lp(nrow(A),ncol(A))

for (i in 1:nrow(A)) set.row(viajante,i,A[i,])
set.objfn(viajante,c)
set.rhs(viajante,b)
set.constr.type(viajante,tipores)

diag<-seq(1,n*n,by=n+1)
cotassup<-c(rep(1,ncol(A)-n),rep(Inf,n))
cotassup[diag]<-0


if (relax==0){
set.type(viajante, 1:(n^2), "binary")
}

set.bounds(viajante, upper = cotassup)

solve(viajante) #Resolvemos el problema

coste<-get.objective(viajante)
solucion<-t(matrix(get.variables(viajante)[1:(n^2)],nrow=n,ncol=n))

################################################################
#########################SOLUCIÓN###############################
################################################################

return(list(circuito=solucion,coste=coste))
}
```

Código con el que se generan los ejemplos:

```
#############################################################################
##############GENERAMOS LAS MATRICES Y LAS GUARDAMOS EN UNA LISTA#############
####################TAMBIÉN LAS GUARDAMOS EN UNA CARPETA#####################
#############################################################################

dimvec <- c(40,80,13)                #numero de nodos
numsim <- 50                         #numero de simulaciones con cada numero de nodos

MATRICES5<-list()
MATRICES10<-list()

for(j in 1:length(dimvec)){
tam<-dimvec[j]
for(i in 1:numsim){
    A<-matrix(as.integer(tam*10*runif(tam^2)),nrow=tam,ncol=tam)
    diag(A)<-0
    A<-A+t(A)
    A<-verif_triang(A,imp=0)
if(j==1){
    MATRICES5[[i]]<-A
write.table(MATRICES5[[i]],paste("matrices5/matriz5-",i,".txt",sep=""),col.names=FALSE,row.names=FALSE)
}
else {
MATRICES10[[i]]<-A
  write.table(MATRICES10[[i]],paste("matrices10/matriz10-",i,".txt",sep=""),col.names=FALSE,row.names=FALSE)
}
}
}


MATRICES13<-list()
tam<-13
for(i in 1:15){
    A<-matrix(as.integer(tam*10*runif(tam^2)),nrow=tam,ncol=tam)
    diag(A)<-0
    A<-A+t(A)
   A<-verif_triang(A,imp=0)
MATRICES13[[i]]<-A
write.table(MATRICES13[[i]],paste("matrices13/matriz13-",i,".txt",sep=""),col.names=FALSE,row.names=FALSE)
}
```

Código con el que ejecutamos los ejemplos:

```
################################################################################
############################LLAMAMOS A LOS PROGRAMAS############################
################################################################################

MATRICES5<-list()
MATRICES10<-list()
MATRICES13<-list()

for (i in 1:numsim){
   archivo<-paste("matrices5/matriz5-",i,".txt",sep="")
   MATRICES5[[i]]<-read.table(archivo)
}

for (i in 1:numsim){
   archivo<-paste("matrices10/matriz10-",i,".txt",sep="")
   MATRICES10[[i]]<-read.table(archivo)
}

for (i in 1:15){
   archivo<-paste("matrices13/matriz13-",i,".txt",sep="")
   MATRICES13[[i]]<-read.table(archivo)
}

#####PRIMERO, LLAMAMOS AL ALGORITMO DE LAS HORMIGAS

numsim <- 50
costemediohor <- c(0,0,0)
tiempomediohor <- c(0,0,0)
costehor2 <- matrix(0,nrow=numsim,ncol=length(dimvec))


for(j in 1:length(dimvec)){
if(j==1){
numsim<-50
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_hormigas(MATRICES5[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costehor2[i,j] <- res[[2]]
costemediohor[j] <- costemediohor[j]+res[[2]]/numsim
tiempomediohor[j] <- tiempomediohor[j]+tiempo/numsim
cat(numsim,costehor2[i,j],tiempo)
print(j)
}
}
else{
if(j==2){
numsim<-50
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_hormigas(MATRICES10[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costehor2[i,j] <- res[[2]]
costemediohor[j] <- costemediohor[j]+res[[2]]/numsim
```

```
tiempomediohor[j] <- tiempomediohor[j]+tiempo/numsim
cat(numsim,costehor2[i,j],tiempo)
print(j)
}
}
else{
numsim <- 15
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_hormigas(MATRICES13[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costehor2[i,j] <- res[[2]]
costemediohor[j] <- costemediohor[j]+res[[2]]/numsim
tiempomediohor[j] <- tiempomediohor[j]+tiempo/numsim
cat(numsim,costehor2[i,j],tiempo)
print(j)
}
}
}
}


write.table(costehor2,paste("costes/costehor2.txt",sep=""),col.names=FALSE,row.names=FALSE)
tiempomediohor
costemediohor

#####SEGUNDO, LLAMAMOS AL ALGORITMO DEL ÁRBOL

numsim <- 50
costemedioar <- c(0,0,0)
tiempomedioar <- c(0,0,0)
costear <- matrix(0,nrow=numsim,ncol=length(dimvec))

for(j in 1:length(dimvec)){
if(j==1){
numsim<-50
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_arbol(MATRICES5[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costear[i,j] <- res[[2]]
costemedioar[j] <- costemedioar[j]+res[[2]]/numsim
tiempomedioar[j] <- tiempomedioar[j]+tiempo/numsim
cat(numsim,costear[i,j],tiempo)
print(j)
}
}
else{
if(j==2){
numsim<-50
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_arbol(MATRICES10[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costear[i,j] <- res[[2]]
costemedioar[j] <- costemedioar[j]+res[[2]]/numsim
```

```
tiempomedioar[j] <- tiempomedioar[j]+tiempo/numsim
cat(numsim,costear[i,j],tiempo)
print(j)
}
}
else{
numsim <- 15
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_arbol(MATRICES13[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costear[i,j] <- res[[2]]
costemedioar[j] <- costemedioar[j]+res[[2]]/numsim
tiempomedioar[j] <- tiempomedioar[j]+tiempo/numsim
cat(numsim,costear[i,j],tiempo)
print(j)
}
}
}
}


write.table(costear,paste("costes/costear.txt",sep=""),col.names=FALSE,row.names=FALSE)
tiempomedioar
costemedioar

#####TERCERO, LLAMAMOS AL ALGORITMO DE CHRISTOFIDES

numsim <- 50
tiempomediochris <- c(0,0,0)
costemediochris <- c(0,0,0)
costechris <- matrix(nrow=numsim,ncol=length(dimvec))

for(j in 1:length(dimvec)){
if(j==1){
numsim<-50
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_christofides(MATRICES5[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costechris[i,j] <- res[[2]]
costemediochris[j] <- costemediochris[j]+res[[2]]/numsim
tiempomediochris[j] <- tiempomediochris[j]+tiempo/numsim
cat(numsim,costechris[i,j],tiempo)
print(j)
}
}
else{
if(j==3){
numsim<-15
for(i in 1:numsim){
          tiempo<-system.time(res<-alg_christofides(MATRICES13[[i]]),gcFirst=TRUE)
          tiempo<-tiempo[[1]]
costechris[i,j] <- res[[2]]
costemediochris[j] <- costemediochris[j]+res[[2]]/numsim
```

```
tiempomedioar[j] <- tiempomediochris[j]+tiempo/numsim
cat(numsim,costechris[i,j],tiempo)
print(j)
}
}
}
}

write.table(costechris,paste("costes/costechris.txt",sep=""),col.names=FALSE,row.names=FALSE)
tiempomediochris
costemediochris

#####CUARTO, LLAMAMOS AL ALGORITMO DE PLE

numsim <- 15
tiempomediople <- 0
costemediople <- 0
costeple <- matrix(0,nrow=numsim,ncol=1)

for(i in 1:numsim){
numsim <- 15
tiempo<-system.time(res<-alg_ple(MATRICES13[[i]]),gcFirst=TRUE)
tiempo<-tiempo[[1]]
costeple[i] <- res[[2]]
costemediople <- costemediople+res[[2]]/numsim
tiempomediople <- tiempomediople+tiempo/numsim
}

write.table(costeple,paste("costes/costeple.txt",sep=""),col.names=FALSE,row.names=FALSE)
tiempomediople
costemediople
```

Código con el que cálculo el error cometido, con respeto a la solución óptima, de los valores obtenidos por cada uno de los programas:

```
costehor1<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costehor.txt",sep="")
costehor1<-read.table(coste)

costehor2<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costehor1.txt",sep="")
costehor2<-read.table(coste)

costehor3<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costehor2.txt",sep="")
costehor3<-read.table(coste)

costepl<-matrix(0,nrow=15,ncol=1)
coste<-paste("costes/costeple.txt",sep="")
costepl<-read.table(coste)

costearbol<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costear.txt",sep="")
costearbol<-read.table(coste)

costelk<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costelinker.txt",sep="")
costelk<-read.table(coste)

costeconcorde<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costeconcorde.txt",sep="")
costeconcorde<-read.table(coste)

costechris<-matrix(0,nrow=50,ncol=3)
coste<-paste("costes/costechris.txt",sep="")
costechris<-read.table(coste)


for(j in 1:3){
if(j==1){
coste40hor1<-0
coste40hor2<-0
coste40hor3<-0
coste40arbol<-0
coste40lk<-0
for(i in 1:50){
coste40hor1<-coste40hor1+(costehor1[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste40hor2<-coste40hor2+(costehor2[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste40hor3<-coste40hor3+(costehor3[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste40arbol<-coste40arbol+(costearbol[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste40lk<-coste40lk+(costelk[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
}
coste40hor1<-coste40hor1/50
coste40hor2<-coste40hor2/50
```

```
coste40hor3<-coste40hor3/50
coste40arbol<-coste40arbol/50
coste40lk<-coste40lk/50
}
else{
if(j==2){
coste80hor1<-0
coste80hor2<-0
coste80hor3<-0
coste80arbol<-0
coste80lk<-0
for(i in 1:50){
coste80hor1<-coste80hor1+(costehor1[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste80hor2<-coste80hor2+(costehor2[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste80hor3<-coste80hor3+(costehor3[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste80arbol<-coste80arbol+(costearbol[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste80lk<-coste80lk+(costelk[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
}
coste80hor1<-coste80hor1/50
coste80hor2<-coste80hor2/50
coste80hor3<-coste80hor3/50
coste80arbol<-coste80arbol/50
coste80lk<-coste80lk/50
}
else{
coste13hor1<-0
coste13hor2<-0
coste13hor3<-0
coste13arbol<-0
coste13lk<-0
for(i in 1:15){
coste13hor1<-coste13hor1+(costehor1[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste13hor2<-coste13hor2+(costehor2[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste13hor3<-coste13hor3+(costehor3[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste13arbol<-coste13arbol+(costearbol[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
coste13lk<-coste13lk+(costelk[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
}
coste13hor1<-coste13hor1/15
coste13hor2<-coste13hor2/15
coste13hor3<-coste13hor3/15
coste13arbol<-coste13arbol/15
coste13lk<-coste13lk/15
}
}
}

###Programación lineal y entera
coste13pl<-0
for(i in 1:15){
coste13pl<-coste13pl+(costepl[i,1]-costeconcorde[i,3])/costeconcorde[i,3]
}
coste13pl<-coste13pl/15
```

```
###Christofides

for(j in 1:3){
if(j==1){
coste40chris<-0
for(i in 1:50){
coste40chris<-coste40chris+(costechris[i,j]-costeconcorde[i,j])/costeconcorde[i,j]
}
coste40chris<-coste40chris/50
}
else{
coste13chris<-0
for(i in 1:15){
coste13chris<-coste13chris+(costechris[i,j]-costeconcorde[i,j])/costeconcorde[i,j]

}
coste13chris<-coste13chris/15
}
}
```