Grado en Matemáticas

TRABAJO FIN DE GRADO

Análisis Experimental de heurísticas para el Problema del Viajante de Comercio

José María Verde López

Sevilla, junio de 2018

Dedicado a mi familia y amigos.

Agradecimientos

Para empezar quiero mostrar mi agradecimiento a quien más me ha ayudado en este camino (que no ciclo, porque no acabo donde empecé), mi tutor Álvaro.

Al comienzo del curso pensaba que mi tutor del Trabajo de Fin de Grado sería un profesor al que consultar ocasionalmente alguna duda. Me ha quedado claro que nada más lejos de la realidad, pues él estaba siempre al pie del cañón dispuesto a ayudarme con cualquier problema que tuviera (además de haber demostrado una paciencia infinita con mi ordenador). Profesionales como él hacen que esta universidad crezca día a día.

Quiero agradecer también a mis padres y a mi hermano todo su apoyo en el día a día y su interés por conocer un poco esta ciencia que es uno de los pilares de mi vida, a pesar de que no siempre sea comprensible para ellos.

Igualmente debo mencionar también a mis amigos, quienes están ahí siempre y hacen que los momentos más difíciles queden en simples baches.

Por último he de agradecer a todos mis profesores y, en general, a todo aquel que me ha enseñado algo en la vida, porque gracias a ellos he llegado hasta aquí.

Resumen

El Problema del Viajante de Comercio es un problema de optimización de tipo NP-duro cuyo objetivo es, dado un conjunto de puntos y la distancia entre cada par de ellos, encontrar el camino más corto que pase por todos ellos y empiece y acabe en el mismo punto.

Este problema aparentemente tan sencillo esconde una gran complejidad a la hora de afrontarlo, pues a medida que va aumentando el número de puntos nos damos cuenta de que resolverlo por fuerza bruta es en vano.

Debido a esto y a la importancia de sus aplicaciones se han realizado numerosos estudios acerca del problema, centrados normalmente en el diseño de heurísticas con el fin de reducir el tiempo de cálculo de soluciones.

En este Trabajo de Fin de Grado se ha llevado a cabo la implementación en el lenguaje de programación *Haskell* de varias de esas heurísticas y de un algoritmo de fuerza bruta con el objetivo de hacer al final de la presente memoria un análisis experimental comparativo de la eficiencia de cada uno de estos algoritmos.

Abstract

The Travelling Salesman Problem is an NP-hard optimization problem whose aim is to find the shortest way through all of them that starts and ends at the same point given a set of points and the distance between each of them.

This apparently naive problem hides a huge complexity when being faced since one can easily realize that solving it by a brute-force algorithm does not help once the number of points starts rising.

Due to this and to the importance of its applications, numerous studies about the problem have been carried out, usually centred on the design of heuristics, whose aim is to reduce the computing time of solutions.

We have carried out an implementation of some of those heuristics during this Final Degree Project using the programming language *Haskell*, as well as the implementation of a brute-force algorithm in order to check the hardness of the problem. The aim of all this is to experimentally make a comparative analysis of the eficiency of each of these algorithms.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El Problema del Viajante de Comercio (en inglés *Travelling Salesman Problem*, TSP) es un problema NP-duro que ha sido ampliamente estudiado por ramas de las matemáticas tales como la Investigación Operativa y las Ciencias de la Computación.

Informalmente, el problema consiste en, dada una lista de ciudades y las distancias entre cada par de ellas, encontrar la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen.

1.1. Evolución del TSP a lo largo de la historia

El origen del nombre «Problema del Viajante de Comercio» es un misterio. Aparentemente no hay ningún documento oficial que esclarezca el nombre del autor y no tenemos verdaderos indicios de cuándo se empezó a usar. Uno de los investigadores más influyentes y prolíficos en los primeros avances del TSP fue Merrill Flood, de la Universidad de Princeton y de la RAND Corporation (*Research and Development Corporation*), el cual comentó en una entrevista:

> Mientras trataba de resolver un problema de rutas de autobuses escolares para el Estado de West Virginia me topé con el Problema de los 48 Estados (*48 States Problem*) de Hassler Whitney, el cual fue el primero al que se le adjudicó el nombre de Problema del Viajante de Comercio. Al final, el nombre ha calado bastante hondo y el problema ha resultado ser de gran importancia.

A excepción de pequeñas variaciones en su pronunciación y en la forma de escribirlo, el nombre de TSP se había expandido completamente por la comunidad científica en la década de 1950.

El Manual de 1832 para viajantes de comercio

Los viajantes de comercio estaban muy interesados en el diseño de rutas comerciales que atravesaran los principales puntos de venta. Un importante aporte en este contexto es el manual alemán de 1832 *Der Handlungsreisende – wie er sein soll und was er zu tun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur*, que llegó a manos de la comunidad investigadora del TSP en 1983.

El libro incluye cinco rutas a través de regiones de Alemania y Suiza, cuatro de las cuales vuelven al punto de partida. Pero este no es el único ejemplo de la presencia del TSP en esa época, podemos encontrar también libros como *Commercial Traveller's Guide Book* (L.P. Brockett, 1880) e incluso un juego de mesa de 1890 llamado *Commercial Traveller* que consistía en diseñar rutas a través de una red de ferrocarriles.

Llegada del TSP a la comunidad matemática

La primera aparición del TSP en la literatura matemática corresponde a Karl Menger, quien describió una variante del problema en un coloquio en Viena en 1930. A partir de ese momento empezamos a ver numerosos ejemplos de estudios sobre rutas óptimas en diversos ámbitos, como por ejemplo conexiones entre campos de cultivo en India (Eli S. Marks, 1940), autobuses escolares en EE. UU. (Merrill Flood, 1937) o rutas de vigilancia policiales (Gerald Thompson, principios de los años 60).

Estos son ejemplos de estudios sobre instancias concretas. No obstante, posteriormente fueron desarrollándose los distintos algoritmos que se utilizan hoy en día para la resolución computacional del problema, así como teoremas acerca del TSP que han ido ayudando a refinar dichos algoritmos.

1.2. Formalización del TSP

El problema TSP puede formalizarse matemáticamente como un problema de grafos.

Problema del Viajante de Comercio. *Sea $V = \{v_1, \ldots, v_n\}$ un conjunto de n puntos en un espacio X dotado de una distancia d y sea $p_{i,j} = d(v_i, v_j)$ la distancia entre los puntos v_i y v_j. Sea $G = (V, A)$ el grafo completo ponderado sobre V, con $p_{i,j}$ el peso de la arista $a_{i,j}$.*

El TSP *para G consiste en encontrar el ciclo hamiltoniano de mínimo coste, es decir, una secuencia de vértices $\{v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)}\}$ (donde π es una permutación de los subíndices 1 a n) que minimiza la suma $\sum_{i=1}^{n-1} p_{\pi(i),\pi(i+1)} + p_{\pi(n),\pi(1)}$.*

La complejidad de este problema reside en que la cantidad de ciclos posibles para n ciudades es de $n!$, es decir, crece de forma más que exponencial a medida que aumenta la

cantidad de puntos. Esto implica que resolverlo por un algoritmo de fuerza bruta (evaluando todos los posibles trayectos) es inviable. Por ello, el estudio de formas alternativas de resolución se basa en gran medida en el diseño de heurísticas que proporcionen un cierto equilibrio entre el tiempo de ejecución y la calidad de la solución obtenida.

Antes de continuar, cabe mencionar la existencia de dos principales versiones del TSP, dependiendo de si el grafo es no dirigido (Problema del Viajante de Comercio simétrico, STSP) o es dirigido (Problema del Viajante de Comercio asimétrico, ATSP). En este trabajo nos centraremos únicamente en el estudio del primero, por lo que de ahora en adelante todos los TSPs que tratemos serán simétricos.

1.3. Aplicaciones

Es fácil darse cuenta (y de ahí el interés que suscita) de que los vértices del grafo de una instancia del TSP pueden interpretarse como cualquier tipo de objeto que encaje debidamente en el planteamiento, por lo que la cantidad de problemas que pueden plantearse como un TSP para facilitar su resolución es muy grande.

Algunos ejemplos de aplicaciones son:

- **Diseño de rutas:** se trata de la aplicación más inmediata que se nos puede ocurrir al pensar en el problema, pues es la más presente en la vida diaria. Ejemplos de estas rutas pueden ser desde repartos de correos o recogidas de residuos hasta la ruta seguida por un autobús escolar para recoger a todos sus pasajeros.

- **Orientación de telescopios:** en este caso el recorrido por los puntos no se hace físicamente, sino que el problema consiste en diseñar el recorrido de un telescopio que debe inspeccionar distintos puntos del espacio. La distancia total de este problema es el tiempo que tomaría observar los puntos en ese orden concreto.

- **Cristalografía de Rayos X:** el TSP aparece en este contexto al tratar de minimizar el tiempo empleado en cambiar un cristal por otro y un equipo de rayos X por el siguiente (debido a que cada cristal requiere de un equipo distinto).

- **Secuenciación del genoma:** al realizar estudios sobre secuenciación genómica en el ADN se obtienen distintos fragmentos (series de nucleótidos) que han de ser ensamblados entre sí como si de un puzzle se tratara, lo que implica la resolución de instancias del TSP.

- **Producción de chips:** cada chip cuenta con una serie de puntos cuyo circuito integrado debe conectar entre sí, de manera que la calidad del chip depende en gran medida de la distancia total del circuito. Aunque pueda parecer que las distancias dentro del chip son muy cortas, la inmensa cantidad de veces que debe recorrerse el

circuito hace que sea de vital importancia optimizar el diseño del chip lo máximo posible.

- **Perforado de circuitos impresos:** para montar distintas capas de los chips mencionados en el apartado anterior se necesitan unas placas especiales en las que se realizan multitud de agujeros para ensamblar los chips. Esto lleva a un problema de tipo TSP a la hora de automatizar el orden de realización de dichos agujeros en la placa.

1.4. Variantes del TSP

Existen multitud de problemas de grafos similares al TSP que pueden ser fácilmente traducibles a un TSP propiamente dicho:

- **MaxTSP:** este problema consiste en encontrar el ciclo hamiltoniano de máxima distancia. Podemos convertirlo en un TSP cambiando el signo de los pesos de las aristas (y añadiendo una constante grande a cada uno en caso de que la forma de resolverlo pida explícitamente que estos sean positivos).

- **TSP con visitas múltiples:** en esta variante se debe visitar cada vértice al menos (en lugar de exactamente) una vez.

- **BottleneckTSP:** este problema consiste en encontrar un ciclo hamiltoniano que minimice el valor de la arista con mayor peso del ciclo. Puede ser reinterpretado como un TSP con aristas con un peso exponencialmente grande.

- **ClusterTSP:** en este caso los puntos se agrupan en conjuntos disjuntos y se añade la restricción de que los puntos de un mismo conjunto deben ser visitados de forma consecutiva. La reinterpretación en forma de TSP se hace aumentando considerablemente el peso de aristas entre puntos de distintos conjuntos.

- **Problema de los m-Viajeros de Comercio:** dado un vértice v de $G = (V, A)$ se trata de encontrar una partición $X_1, \ldots, X_m$ del conjunto $V \setminus \{v\}$ que minimice la suma de las distancias de las soluciones del TSP para cada $X_i \cup \{v\}$, siendo siempre v la ciudad de partida y de llegada.

- **Problema del Cartero Chino:** problema análogo al TSP, pero cuyo objetivo es buscar el ciclo euleriano de mínimo coste dentro del grafo.

Capítulo 2

Algoritmos para la resolución del TSP

Vamos a proceder ahora a describir el funcionamiento de cada uno de los algoritmos implementados durante la realización de este Trabajo de Fin de Grado, así como mostrar una parte del código utilizado para ello y los resultados de eficiencia obtenidos para cada algoritmo, con el fin de realizar un estudio comparativo de todos ellos.

2.1. Algoritmo de Fuerza Bruta

Los algoritmos de fuerza bruta son una primera aproximación trivial al proceso de resolución computacional de un problema. Destacan por su simpleza, ya que consisten en generar todas las soluciones candidatas para luego evaluar cada una de ellas, comprobando si se ajustan o no a los requisitos que debe cumplir una solución correcta del problema.

El inconveniente que tienen es el enorme coste computacional que conllevan, pues en muchos casos la cantidad de posibles soluciones a generar crece de forma exponencial o factorial al aumentar el tamaño del problema (en el caso del TSP, como ya hemos mencionado anteriormente, el número de soluciones candidatas es del orden $O(n!)$).

2.1.1. Planteamiento

Definición 1. Dado un conjunto $V = \{v_1, \ldots, v_n\}$, definimos la familia de conjuntos $V_{S_n} = \big\{\{v_{\pi(1)}, \ldots, v_{\pi(n)}\} \mid \pi \in S_n\big\}$, donde S_n es el conjunto de todas las permutaciones de $\{1, \ldots, n\}$.

Dada una instancia $G = (V, A)$ del TSP, el algoritmo de fuerza bruta consiste en generar el conjunto V_{S_n} de todas las posibles permutaciones de V, para después evaluar la distancia total de cada permutación.

La distancia asociada a cada permutación π será la longitud total de la ruta que recorre los puntos en el orden indicado y volviendo finalmente al punto inicial, es decir,

la suma $\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)})$ (siendo d una distancia dada).

La solución final del problema mediante este algoritmo será la permutación que minimice la suma anterior.

2.1.2. Implementación

Se ha implementado el algoritmo de fuerza bruta de dos maneras ligeramente diferentes. La primera de ellas utiliza la función `permutations`, ya definida en el preludio de *Haskell*, para generar todas las posibles permutaciones de las ciudades.

Las instancias del TSP consideradas asumen ciudades numeradas de 1 a n, siendo la ciudad 1 la de partida y llegada.

Los datos de entrada en ambas implementaciones deben ser una lista de aristas ponderadas y el número de ciudades. Cada arista debe indicarse en formato `((ciudad de origen, ciudad`

Se proporciona como resultado un par de la forma `(ciclo óptimo de ciudades, distancia to` en la primera implementación y de la forma `(ciclo óptimo de ciudades, distancia total del` en la segunda.

Fuerza bruta (con `permutations`)

```
-- | Calcula el ciclo de menor longitud a partir de la lista
-- de aristas g. El número de ciudades n se da para no tener
-- que recorrer g para conocerlo.
menorCamino :: [((Int,Int),Float)] -> Int -> (Float,[Int])
menorCamino g n =
    minimum (zip (map (fn g) w)
                 w)
  where v = permutations [2..n]
        w = anhade1 v

-- | Añade el vértice 1 a cada una de las rutas.
anhade1 :: [[Int]] -> [[Int]]
anhade1 []     = []
anhade1 (x:xs) = (1:x) : (anhade1 xs)

fn :: [((Int,Int),Float)] -> [Int] -> Float
fn g (x:xs) = pesoDeCamino (x:xs) 0 g

-- | Calcula la distancia total de un ciclo.
pesoDeCamino :: [Int] -> Float -> [((Int,Int),Float)] -> Float
pesoDeCamino (x:y:xs) p g
    |null xs   = p + (pesoArista (x,y) g)
                   + (pesoArista (y,1) g)
```

```
    |otherwise = pesoDeCamino (y:xs)
                               (p + (pesoArista (x,y) g))
                               g

-- | Devuelve el peso de la arista (x,y).
pesoArista :: (Int,Int) -> [((Int,Int),Float)] -> Float
pesoArista (x,y) (g:gs)
    |((x,y) == fst g) || ((y,x) == fst g) = snd g
    |otherwise                            = pesoArista (x,y) gs
```

La optimización de la segunda implementación consiste en que, en lugar de calcular todos los ciclos desde el principio y luego evaluarlos, la distancia del ciclo se va actualizando a medida que se añaden vértices nuevos (como podemos ver en la función `rutasPosibles`), por lo que no es necesario recorrer las listas nuevamente.

Fuerza bruta (propia)

```
-- | Calcula el ciclo de menor longitud a partir de la lista
-- de aristas g. El número de ciudades n se da para no tener
-- que recorrer g para conocerlo.
menorCamino :: [((Int,Int),Float)] -> Int -> ([Int],Float)
menorCamino g n = search (tail sig) (head sig)
  where sig = rutasPosibles (([1],[2..n]),0) g

-- | Devuelve el ciclo de menor longitud de la lista (r:rs).
search ::  [([Int],Float)] -> ([Int],Float) -> ([Int],Float)
search []     k    = k
search (r:rs) k
    |snd r < snd k = search rs r
    |otherwise     = search rs k

rutasPosibles :: (([Int],[Int]),Float) ->
    [((Int,Int),Float)] -> [([Int],Float)]
rutasPosibles ((xs,[]),p) g     =
    [(xs,p + peso (head xs) (last xs) g)]
rutasPosibles (((x:xs),ys),p) g =
    concat [rutasPosibles
            (((y : x : xs), elimina y ys),p + (peso x y g))
               g
               | y <- ys]

-- | Encuentra la arista (x,y) en la lista de aristas y
```

```haskell
-- devuelve su peso.
peso :: Int -> Int -> [((Int,Int),Float)] -> Float
peso x y (g:gs)
    |((x,y) == fst g) || ((y,x) == fst g) = snd g
    |otherwise                            = peso x y gs

-- | Devuelve la lista (y:ys) sin el elemento x.
elimina :: Int -> [Int] -> [Int]
elimina x []   = []
elimina x (y:ys)
    |x == y    = ys
    |otherwise = y : (elimina x ys)
```

2.2. Heurística Strip

La heurística Strip se aplica a instancias geométricas del TSP, es decir, aquellas en las que los vértices del grafo son puntos del plano y la distancia entre ellos es la distancia euclídea. El objetivo de esta heurística es obtener una solución rápidamente, evidentemente a costa de la calidad de la misma.

2.2.1. Planteamiento

Definición 2. Dado un conjunto de n puntos $A = \{(a_i, b_i) : 1 \leq i \leq n\} \subset \mathbb{R}^2$, definimos el mínimo rectángulo recubridor de A como $M = \{(x, y) : \text{mín}_{1\leq i\leq n}\, a_i \leq x \leq \text{máx}_{1\leq i\leq n}\, a_i, \text{mín}_{1\leq i\leq n}\, b_i \leq y \leq \text{máx}_{1\leq i\leq n}\, b_i\}$.

Dado el conjunto $A = \{(a_i, b_i) : 1 \leq i \leq n\}$, el algoritmo Strip comienza dividiendo M en franjas verticales de anchura $\sqrt{\frac{n}{3}}$. Llamamos M_j, $1 \leq j \leq k$, a la j-ésima franja formada de esta manera, siendo M_1 la situada más a la izquierda y M_k la de más a la derecha.

El siguiente paso es, para cada M_j, formar una ruta que recorra sus puntos en orden creciente o decreciente de la variable y, según sea j impar o par respectivamente. Uniendo el último vértice de la ruta de M_j con el primero de la ruta de M_{j+1}, $1 \leq j \leq k-1$, y el último de M_k con el primero de M_1, obtenemos un ciclo hamiltoniano. Se puede demostrar que las rutas construidas por la heurística Strip tienen en el caso peor una longitud a lo sumo $\Omega(\sqrt{n})$ veces la longitud de una ruta óptima.

Las figuras 2.1, 2.2 y 2.3 muestran un ejemplo de funcionamiento de la heurística.

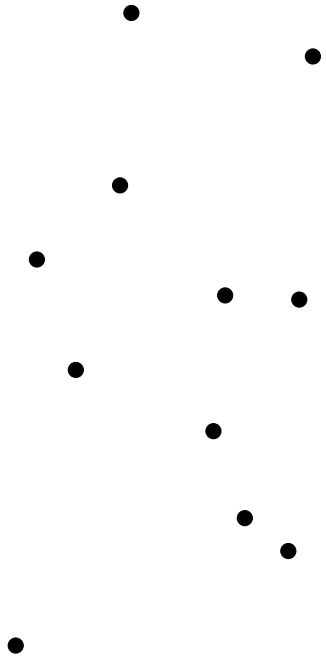

Figura 2.1: 11 puntos generados aleatoriamente

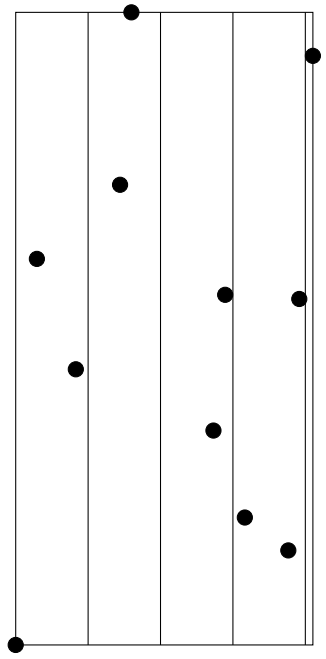

Figura 2.2: Mínimo rectángulo recubridor y su división en franjas

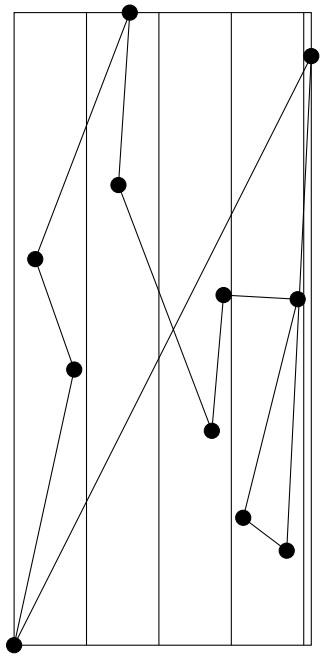

Figura 2.3: Ruta generada por la heurística STRIP

2.2.2. Implementación

El código de la implementación de la heurística STRIP viene en el CD adjunto. Cabe destacar varias cosas:

- La implementación sigue paso por paso el algoritmo y para usarla hay que aplicar la función `heuristicaStrip`.
- La entrada basta que sea la lista de puntos del plano, no es necesario proporcionar el peso de las aristas, ya que se calculará como la distancia euclídea entre los puntos que conecta.
- La salida es un par formado por la ruta construida por la heurística y su longitud.

2.3. Heurística del vecino más cercano

La heurística del vecino más cercano (*Nearest Neighbor*, NN) construye un ciclo hamiltoniano de la siguiente forma:

1. Se elige una ciudad inicial c_1.

2. Suponiendo que $c_1, \ldots, c_i$ es la ruta parcial actual, se elige c_{i+1} como la ciudad más cercana a c_i de entre todas las no incluidas todavía en la ruta.

3. Se completa la ruta conectando la ciudad c_n con la ciudad c_1.

Para este algoritmo utilizaremos una estructura de datos de particionado del espacio llamado Árbol KD (abreviatura de árbol k-dimensional o *k-d tree* en inglés) que nos permitirá realizar la búsqueda del vecino más cercano de un punto de una forma más eficiente que una búsqueda exhaustiva. Nosotros vamos a limitarnos al caso $k = 2$, pues las instancias con las que trabajamos están sobre el plano euclídeo $\mathbb{R}^2$.

La heurística del vecino más cercano tiene en general un coste en tiempo del orden $\Theta(n^2)$, pero para instancias geométricas el uso de árboles kd permite reducirlo a $O(n \log n)$ en la práctica.

2.3.1. Planteamiento

Definimos primero varios conceptos que nos servirán de ayuda:

Definición 3. Dado un conjunto de puntos $S = \{(a_i, b_i) : 1 \leq i \leq n\} \subset \mathbb{R}^2$ definimos $med_{S,x}$, la mediana de S con respecto a la variable x, como el (a_j, b_j) tal que a_j es la mediana del conjunto $\{a_i : 1 \leq i \leq n\}$ si n es impar. En caso de ser par y, por tanto, que dos valores a_{j_1}, a_{j_2} representen la mediana, tomaremos el mayor de los dos. Al valor a_j tal que $med_{S,x} = (a_j, b_j)$ se le llama valor de particionado de S para x, $VP_{S,x}$.

Análogamente definimos también la mediana de S con respecto a la variable y, $med_{S,y}$, cambiando a por b donde corresponda en la definición anterior. Al valor b_j tal que $med_{S,y} = (a_j, b_j)$ se le llama valor de particionado de S para y, $VP_{S,y}$.

En las dos definiciones anteriores, en caso de haber más de un punto cumpliendo esos requisitos, tomamos el que tenga un mayor valor en la otra variable.

La construcción de un Árbol KD tomando el conjunto de puntos $S = \{(a_i, b_i) : 1 \leq i \leq n\}$ es un proceso recursivo que cumple:

- La raíz (el nodo superior del árbol) contendrá el punto $med_{S,x}$ y un valor 0 (denominado valor de orientación o VO, que puede ser 0 o 1), indicando que en ese punto se ha realizado una partición del conjunto S atendiendo a la variable x (en caso de ser la variable y se almacenaría un 1).

  De la raíz saldrán dos subárboles, uno a la izquierda que contendrá a los puntos $\{(a_i, b_i) : a_i \leq VP_{S,x}, (a_i, b_i) \neq med_{S,x}, 1 \leq i \leq n\}$ y otro a la derecha con los puntos $\{(a_i, b_i) : a_i > VP_{S,x}, 1 \leq i \leq n\}$. El VO de ambos subárboles será 1.

- Cada vez que el árbol se divide en dos subárboles, uno a la izquierda con los puntos S_- y otro a la derecha con los puntos S_+, se almacena en cada uno un VO opuesto al de su nodo superior.

El punto almacenado en el subárbol izquierdo será $med_{S_-,f(VO)}$ y el del derecho, $med_{S_+,f(VO)}$, donde $f(VO) = x$ si $VO = 0$ e y si $VO = 1$

Del subárbol izquierdo saldrán otros dos, uno con los puntos de S_- que tengan la coordenada $f(VO)$ menor o igual que $VP_{S_-,f(VO)}$ (y sean distintos de $med_{S_-,f(VO)}$), y otro con el resto (menos $med_{S_-,f(VO)}$).

Del subárbol derecho saldrán otros dos, uno con los puntos de S_+ que tengan la coordenada $f(VO)$ menor o igual que $VP_{S_+,f(VO)}$ (y sean distintos de $med_{S_+,f(VO)}$), y otro con el resto (menos $med_{S_+,f(VO)}$).

- Cuando el conjunto de puntos almacenados en un árbol es vacío el proceso se detiene.

Las figuras 2.4 y 2.5 muestran un ejemplo de construcción de Árbol KD.

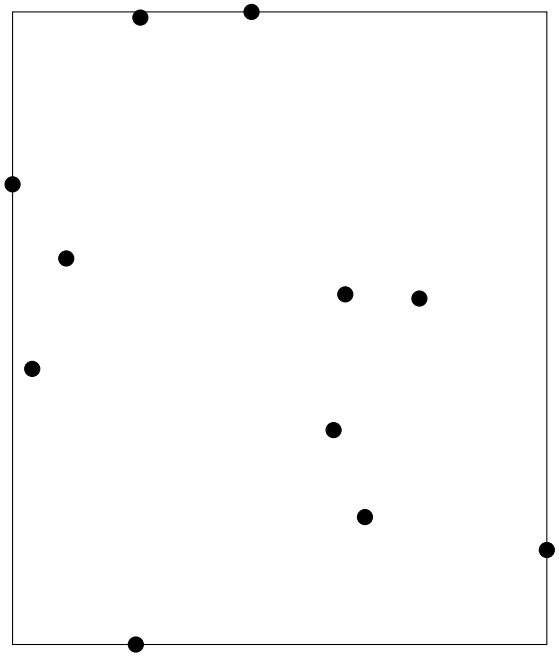

Figura 2.4: Puntos para la construcción de un Árbol KD

Para buscar el vecino más cercano de un punto dado (a, b) en un Árbol KD procedemos de la siguiente manera:

1. Localizamos el cuadrante en el que se encuentra (a, b). Para ello descendemos por el árbol de forma que:
   - Si encontramos un nodo donde se realiza una partición del espacio en sentido vertical en un valor x_0 tomamos el subárbol de la izquierda, si $a \leq x_0$, o el subárbol de la derecha en caso contrario.

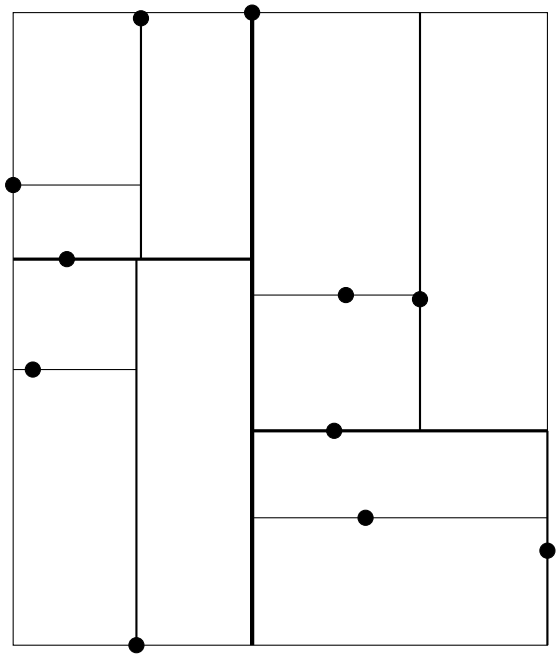

Figura 2.5: Partición generada por el ÁRBOL KD a partir de los puntos de la figura 2.4

- Si encontramos un nodo donde se realiza una partición del espacio en sentido horizontal en un valor y_0 tomamos el subárbol de la izquierda, si $b \leq y_0$, o el subárbol de la derecha en caso contrario.
- En caso de llegar a un nodo hoja (donde no se realicen más particiones), tomamos como punto p más cercano por el momento el almacenado en el nodo superior a la hoja, y como distancia mínima r la distancia euclídea entre p y (a, b).

2. Volvemos a descender por el árbol para analizar los puntos de este de la siguiente forma:

   - Cuando encontramos un nodo (con un punto q almacenado) donde se realiza una partición del espacio en sentido vertical en un valor x_0 calculamos la distancia entre q y (a, b), r'. Aquí pueden darse dos casos:
     - $r \leq r'$**:** analizamos el subárbol de la izquierda, si $a \leq x_0$, o por el subárbol de la derecha en caso contrario. En caso de que la diferencia $|x_0 - a| < r$ se analizan ambos.
     - $r > r'$**:** el punto q reemplaza a p como punto más cercano de esa rama y r' reemplaza a r como nueva distancia mínima. Se analiza el subárbol de la izquierda, si $a \leq x_0$, o el subárbol de la derecha en caso contrario.

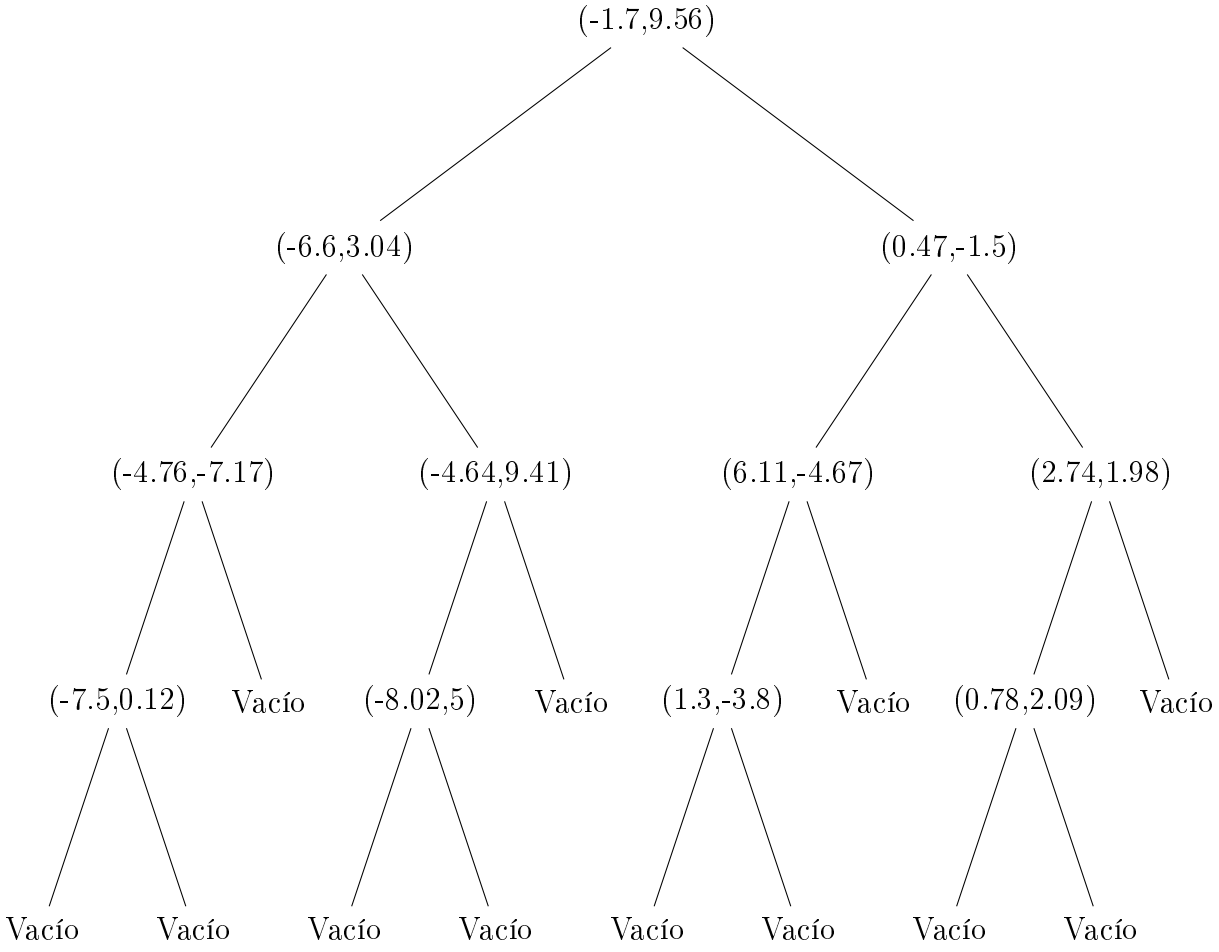

Figura 2.6: Estructura del Árbol KD

- Cuando encontramos un nodo (con un punto q almacenado) donde se realiza una partición del espacio en sentido horizontal en un valor y_0 calculamos la distancia entre q y (a, b), r'. Aquí pueden darse dos casos:
  - $r \leq r'$: analizamos el subárbol de la izquierda, si $b \leq y_0$, o por el subárbol de la derecha en caso contrario. En caso de que la diferencia $|y_0 - b| < r$ se analizan ambos.
  - $r > r'$: el punto q reemplaza a p como punto más cercano de esa rama y r' reemplaza a r como nueva distancia mínima. Se analiza el subárbol de la izquierda, si $b \leq y_0$, o el subárbol de la derecha en caso contrario.
- Cuando la búsqueda en una rama llega a una hoja del árbol, esta se detiene y se guardan el punto más cercano a (a, b) de esa rama y la distancia a él.

3. Cuando todas las búsquedas se han detenido tomamos el vecino más próximo a (a, b) de todos los que hemos obtenido como resultado en cada una de las hojas.

2.3.2. Implementación

El código de la implementación de la heurística NN viene en el CD adjunto.
Cabe destacar varias cosas:

- La implementación sigue paso por paso el algoritmo y para usarla hay que aplicar la función `heuristicaNN`.
- De nuevo no es necesario proporcionar el peso de las aristas como entrada, ya que se calculará como la distancia euclídea entre los puntos que conectan.
- La salida es un par formado por la ruta construida por la heurística y su longitud.
- Se hace uso de la biblioteca Data.Trees.KdTree que implementa la estructura de Árbol KD, así como diversas operaciones, en particular la búsqueda del vecino más cercano y la eliminación de elementos, que son requeridas por la heurística NN. Esta biblioteca trabaja en general con una clase abstracta `Point` y, en concreto, con una instancia `Point2d` de la misma para representar los puntos del plano. Esto ha hecho necesario implementar funciones de «traducción» entre ese tipo `Point2d` y nuestra representación de los puntos del plano como pares de números reales.

2.4. Algoritmo de Christofides

Las heurísticas Strip y NN proporcionan soluciones aproximadas al TSP, sacrificando la calidad de las mismas a cambio de obtenerlas rápidamente. El siguiente algoritmo que vamos a considerar, publicado por Nicos Christofides en 1976, inclina la balanza

hacia el otro lado. Asumiendo únicamente que la distancia considerada satisface la desigualdad triangular (como es el caso de la distancia euclídea), garantiza que la longitud de la solución proporcionada será a lo sumo una vez y media la longitud óptima. Este es a día de hoy el mejor resultado de aproximación conocido para el TSP en espacios métricos generales.

2.4.1. Planteamiento

Sea un conjunto de puntos $V = \{v_1, \ldots, v_n\} \subset \mathbb{R}^2$, construimos el grafo completo inducido por V, $G = (V, A)$, asignando como peso a cada arista $a_{i,j} \in A$ la distancia euclídea entre v_i y v_j.

Los distintos pasos del algoritmo sobre el grafo G son:

1. Obtener el árbol de expansión mínimo T de G.

2. Sea O el conjunto de vértices de grado impar en T, hallar un emparejamiento perfecto M de mínimo peso en el grafo completo sobre los vértices de O.

3. Combinar las aristas de M y T para crear el multigrafo H.

4. Obtener un ciclo euleriano en H.

5. Obtener un ciclo hamiltoniano a partir del ciclo euleriano anterior, descartando los vértices ya visitados. Este ciclo hamiltoniano será el argumento de salida del algoritmo.

En las figuras 2.8, 2.9 y 2.10 puede verse un ejemplo del algoritmo de Christofides sobre el grafo de la figura 2.7.

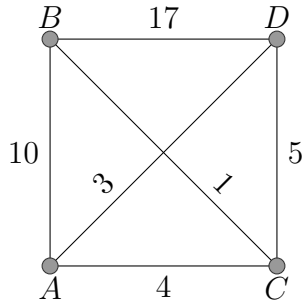

Figura 2.7: Grafo G

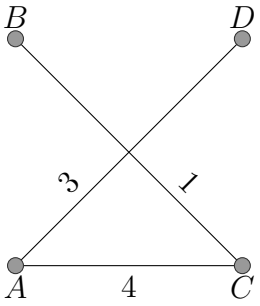

Figura 2.8: Árbol de expansión mínimo T sobre el grafo G

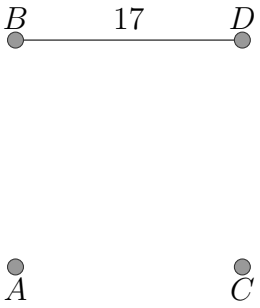

Figura 2.9: Vértices de grado impar en T y emparejamiento perfecto de mínimo coste sobre ellos

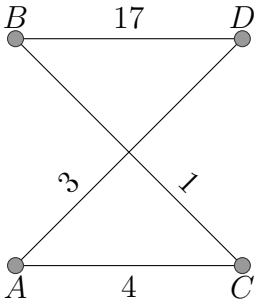

Figura 2.10: Multigrafo H obtenido combinando las aristas de M y T. Coincide con los ciclos euleriano y hamiltoniano construidos sobre el propio H.

2.4.2. Implementación

El código de la implementación de la heurística NN viene en el CD adjunto. Puntos a destacar:

- El tipo con el que escribiremos los grafos será:

  Tipo de dato de los grafos

  ```
  type Node = String
  -- La cadena es el nombre del vértice en cuestión.

  type Edge = ([Node], Float)
  -- El primer elemento es la lista de los vértices que une
  -- la arista y el segundo el peso de esta. Esta lista
  -- tendrá siempre dos elementos únicamente.

  type Graph = ([Node], [Edge])
  -- El primer elemento es la lista de vértices y el segundo
  -- la de aristas.
  ```

- El cálculo del árbol de expansión mínimo se realiza mediante el algoritmo de Kruskal, cuya implementación se ha obtenido de https://github.com/ddrake/haskell-kruskal/blob/master/Kruskal.hs.

- El cálculo del emparejamiento perfecto de mínimo peso es la parte más complicada de implementar. El algoritmo más eficiente a día de hoy se llama *Blossom V* y es una versión mejorada del llamado algoritmo *Blossom*, desarrollado por el matemático Jack Edmonds.

  Debido a la complejidad de esta implementación se ha recurrido a una heurística que devuelve otro emparejamiento perfecto (no de mínimo coste necesariamente), ya que se ha comprobado que en la práctica basta con obtener una buena aproximación.

  Esta heurística es conocida como voraz y funciona de la siguiente manera:

  - Tenemos un grafo ponderado $G = (V, A)$ y una lista M de las aristas que forman el emparejamiento perfecto, inicialmente vacía.
  - En cada iteración se busca la arista $a = ((v_i, v_j), p) \in A$ de mínimo peso y se añade a M. Se borran los vértices v_i y v_j de V y todas las aristas incidentes en ellos (incluyendo a) de A. Si V es vacío termina el algoritmo y se obtiene el grafo (V, M) como argumento de salida. En caso contrario se repite este paso.

- Para el cálculo del ciclo euleriano en H se utiliza una técnica conocida como el «Hilo de Ariadna» para llevar la cuenta de en qué vértices ha habido una encrucijada de caminos (necesidad de elegir una de entre varias aristas para continuar construyendo el ciclo). Esto lo hacemos actualizando a medida que avanzamos una lista con ternas de la forma (vértice v visitado en este paso, 1 o 0 dependiendo de si ha habido una encrucijada en v o no, aristas incidentes en v que ya hemos probado y cuyas posibilidades dan todas a callejones sin salida). Estas ternas nos son útiles porque, aparte de llevar la cuenta del ciclo, nos dan información sobre qué hacer en cada paso dependiendo de lo que encontremos.

- El argumento de entrada del algoritmo es un grafo proporcionado en la forma descrita arriba (cada vértice representa a un punto y los pesos de las aristas son las distancias entre los distintos puntos), y el de salida un par de la forma `(Float,[Node])`, siendo el primer elemento la longitud de la ruta construida por la heurística y el segundo la ruta propiamente dicha.

- La función que ejecuta el algoritmo es `christofides`.

2.5. Algoritmo genético

Los algoritmos genéticos son una de las técnicas más usadas por la inteligencia artificial. Están inspirados en la evolución biológica y su base genético-molecular y se utilizan en problemas de optimización.

Los algoritmos genéticos trabajan con una población de individuos, cada uno de los cuáles posee unas características determinadas que vienen codificadas en sus «genes». Esta población va evolucionando a lo largo del tiempo igual que una población real de seres vivos, de forma que aquellos individuos cuyas características los hacen estar «mejor adaptados» tendrán más probabilidades de sobrevivir.

Además del factor de supervivencia existe otro muy importante en este tipo de algoritmos, que es la capacidad de los individuos de «reproducirse» entre sí para dar lugar a otros cuyos genes son una combinación de los de sus progenitores. Estos genes pueden, además, sufrir mutaciones en el proceso (modificaciones aleatorias de la secuencia genómica) para dar lugar a individuos completamente distintos de los ya existentes en la población.

El rendimiento del algoritmo depende en gran medida de la capacidad para expresar las características a optimizar en los individuos, es decir la capacidad de modelizar bien el problema. Esta tarea puede ser sencilla en algunos casos (como el del problema del TSP) o muy compleja (en casos de optimización de diseños industriales, por ejemplo).

2.5.1. Planteamiento

Definición 4. Sea P un problema de optimización con unas restricciones dadas, x es una solución de P ($sol(P, x)$) si satisface sus restricciones. Definimos también el conjunto $X_P = \{x : sol(P, x)\}$ de las soluciones del problema P.

En el caso del problema del TSP, las soluciones son todas las permutaciones posibles de las ciudades dadas como argumento de entrada del problema, por lo que una no-solución del problema sería, por ejemplo, una lista con ciudades repetidas o donde falta alguna de ellas.

Definición 5. Dado un problema de optimización P, definimos la función $f_P : X_P \to \mathbb{R}$ de aptitud de P, que mide lo buena que es una solución del problema P.

Para el TSP la función de aptitud es la que asigna a cada solución su distancia total.

Definición 6. Sea P un problema y f_P su función de aptitud, $\hat{x} \in X_P$ es la solución óptima de P si $f_P(\hat{(x)}) \geq f_P(x)$ (resp., $f_P(\hat{(x)}) \leq f_P(x)$) $\forall x \in X_P$.

La medida de la aptitud de las soluciones por parte de f es relativa, pues es posible que aunque para un problema P se tenga $k = f_P(\hat{x})$, para otro problema P' las soluciones del conjunto $f^{-1}(k) \subset X_{P'}$ estén muy alejadas en calidad de la óptima para P'.

Así, en una instancia del TSP con 2.000 ciudades, la longitud de la solución óptima podría ser mucho mayor que la distancia de cualquiera de las soluciones de otra instancia con 10 ciudades.

Un algoritmo genético para un problema de optimización P consta de los siguientes pasos:

1. Generamos aleatoriamente una población inicial de individuos llamada POB. POB es un conjunto de soluciones del problema (pudiendo haber soluciones repetidas), de forma que $POB \subset X_P$.

2. Aplicamos la función de aptitud f_P a cada uno de los individuos de la población anterior.

3. Como en muchos casos desconocemos cuánto vale $f(\hat{x})$ para P, se utilizan otros métodos de parada. Los dos principales son:

   - Detenerse después de un número prefijado de generaciones
   - Detenerse cuando, al producir una nueva generación, la mejor solución de esta es muy parecida a la de la generación anterior (o directamente no cambia).

   Estas condiciones de parada se pueden utilizar alternativa o simultáneamente. En caso de no cumplirse la condición (o las condiciones) de parada se realizan las siguientes operaciones sobre el conjunto POB:

a) **Selección de los mejores:** Tomamos un subconjunto $SEL \subset POB$ de los individuos con mayor aptitud. Esta selección puede realizarse de forma completamente determinista o semialeatoria. En nuestro caso lo haremos utilizando un sistema llamado «de torneo». Consiste en tomar dos individuos de POB, x e y, de forma aleatoria y calcular el valor absoluto de la diferencia $f(x) - f(y)$. Si esta es mayor que un parámetro, que llamaremos α, se selecciona el individuo de mayor aptitud, en caso contrario se hace aleatoriamente. Este proceso se realiza repetidas veces hasta que el conjunto SEL es lo suficientemente grande (el tamaño considerado suficiente dependerá de cómo enfoquemos el problema).

b) **Cruzamiento:** Representa la reproducción sexual biológica y consiste en la combinación de los genes de dos individuos del conjunto SEL para dar un nuevo individuo $x \in X_P$. Para completar la población de la siguiente generación repetiremos este proceso de cruzamiento de individuos hasta que sea necesario, obteniendo un conjunto de individuos descendientes del conjunto SEL. Este nuevo conjunto se denominará DES.

Para aplicar el cruzamiento al TSP tomamos dos individuos, x e y, que serán ciclos de ciudades. Escogemos dos puntos aleatorios del ciclo de x y nos quedamos con las ciudades situadas entre estos dos puntos, obteniendo así una subsecuencia x' donde algunos huecos han quedado vacíos. Dichos huecos serán rellenados con las ciudades que no estén en x' colocadas en el mismo orden en el que están en y.

Este proceso queda ilustrado en la figura 2.11:

c) **Mutación:** Este proceso se realiza sobre el conjunto DES. En cada individuo de DES se lleva a cabo una ligera modificación con una cierta probabilidad. El conjunto formado por los individuos de DES tras pasar la etapa de la mutación se denomina MUT.

Aplicado al TSP, este proceso elige en cada individuo dos ciudades de su ciclo e intercambia sus posiciones (también con una cierta probabilidad). Podemos ver un ejemplo en la figura 2.12

Tras aplicar estos operadores genéticos formamos una nueva generación de individuos juntando los conjuntos SEL y MUT (que representen a los progenitores que han sobrevivido y su descendencia ya mutada). Esta nueva generación vuelve al paso 2 del algoritmo en el lugar de POB.

Puede verse un esquema del funcionamiento del algoritmo en la figura 2.13.

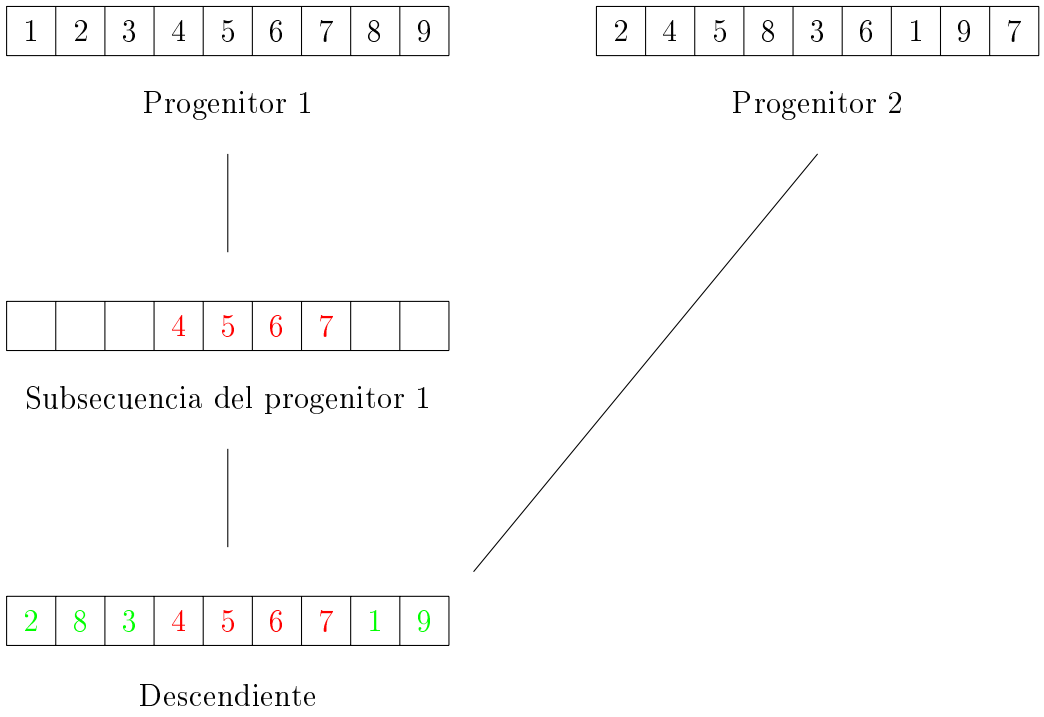

Figura 2.11: Cruzamiento de dos individuos en una instancia del TSP con 9 ciudades.

Figura 2.12: Mutación de un individuo en una instancia del TSP con 9 ciudades.

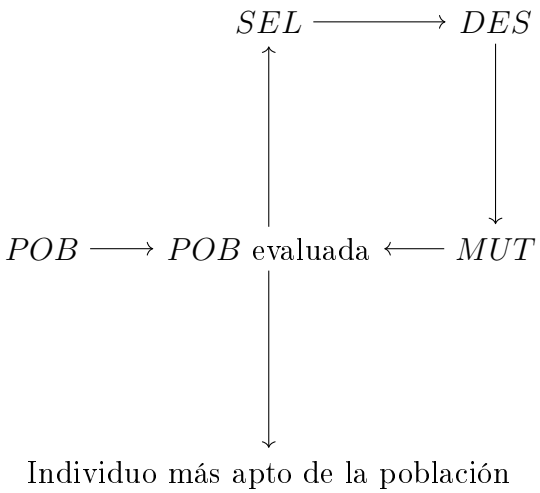

Figura 2.13: Esquema del algoritmo genético

2.5.2. Implementación

El código de la implementación del algoritmo genético aplicado al TSP viene en el CD adjunto.

Cabe destacar varias cosas:

- La función que ejecuta el algoritmo es `algoritmoGenetico`.
- El argumento de entrada del algoritmo es la lista de puntos en formato `(Float,Float)`. No es necesario proporcionar el peso de las aristas porque se calculará como la distancia euclídea entre los puntos que conectan.
- La salida es un par formado por la ruta construida por el algoritmo y su longitud.
- El algoritmo tiene varios parámetros que se pueden modificar para obtener distintos resultados, según nos queramos centrar más en la calidad de la ruta obtenida o en acortar el tiempo de ejecución.Dichos parámetros son:
  - **Probabilidad de mutación:** es la probabilidad de que un individuo mute.
  - **Tamaño de la población:** es el número de individuos que tiene la población. Se mantiene constante al pasar de una generación a la siguiente.
  - **Índice de superioridad:** influye a la hora de seleccionar los mejores individuos de la población. Cuando tomamos dos individuos al azar para compararlos, si la diferencia relativa entre sus aptitudes es menor que el índice superioridad entonces se elige uno de ellos al azar. En caso contrario se elige al más apto.

- **Número de generaciones:** es el número de generaciones que calcularemos antes de detener el algoritmo.

Estos parámetros también influyen a la hora de obtener soluciones que sean óptimos locales o el óptimo global. Por ejemplo, cuanto mayor sea la probabilidad de mutación mayor diversidad habrá en la población.

Capítulo 3

Batería de pruebas

La batería de pruebas utilizada para probar la eficiencia de los algoritmos estudiados en el trabajo consta de 20 instancias del TSP obtenidas de la página http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html, la cual es una biblioteca de instancias reales para el TSP y otros problemas relacionados.

Todas las instancias que hemos seleccionado son conjuntos de puntos del plano (por lo que se corresponden con problemas STSP) de distintos tamaños. El hecho de que estén contenidos en el plano facilita la implementación de los algoritmos, ya que la distancia entre estos puntos es la euclídea.

Estos puntos se han guardado en ficheros de texto separados para poder leerlos fácilmente con un fichero de *Haskell* llamado «intficheros». Junto con estos puntos también se ha guardado la longitud del ciclo óptimo de cada instancia para poder comprobar a posteriori la calidad de la solución obtenida por cada algoritmo.

Para testear los algoritmos se ha escrito otro fichero de *Haskell* llamado «testeador» que toma los puntos de cada una de esas 20 instancias y escribe un fichero de texto con la longitud del ciclo construido por el algoritmo y el tiempo empleado para ello. En caso de que se sobrepase un límite de tiempo previamente establecido se pasa a la siguiente instancia automáticamente (para ello se utiliza la función `timeout` definida en la biblioteca System.Timeout).

En la tabla 3 se pueden ver los tiempos de ejecución del algoritmo Strip para cada una de de las instancias de la batería de pruebas, así como la longitud de la ruta obtenida y la longitud de la solución óptima.

Para la heurística del vecino más cercano se ha usado la batería de pruebas con un tiempo máximo de 10 minutos por instancia, obteniéndose unos resultados sorprendentemente buenos. Los resultados pueden verse en la tabla 3 (para las instancias que no se encuentran en la tabla la heurística tardó más tiempo del estipulado).

Para el algoritmo genético hemos establecido un límite de tiempo de 35 minutos para cada instancia. Solamente tenemos datos de las 12 primeras (aunque la 1 y la 11 superaron el tiempo límite) porque a partir de ella se agotó la memoria del ordenador.

Instancia	Tiempo (en segundos)	Longiud de la ruta de la heurística	Longitud de la solución real
1	0.0156001	2818.6216	2579
2	0.0156	10299.8955	3916
3	0.0156	276532.34	126643
4	0	373943.3	29368
5	0.0156	22205.621	7542
6	0	393998.28	118282
7	0.0156001	273236.47	26130
8	0.0156001	47800.773	6110
9	0.0156001	52812.145	6528
10	0	327452.4	29437
11	0	5.576335e8	18660188
12	0.0156	1313.4684	426
13	0.0156001	1974.714	538
14	0	26295.639	2378
15	0.0156	191393.75	21282
16	0.0156	157184.67	22141
17	0.0156001	287850.34	26524
18	0.0156	36478.17	14379
19	0.0156	119866.92	42029
20	0	62756.973	44303

Cuadro 3.1: Análisis de la heurística STRIP

Los resultados pueden verse en la tabla 3.

Para el algoritmo de Christofides no se han obtenido resultados en tiempos inferiores a 35 minutos (tiempo máximo establecido por instancia).

La conclusiones que obtenemos de este estudio son:

- La calidad de las soluciones obtenidas por el algoritmo STRIP es muy baja aunque se encuentren soluciones de instancias grandes en muy poco tiempo.
- La heurística del vecino más cercano obtiene resultados realmente buenos en comparación con su tiempo de ejecución.
- El algoritmo genético obtiene en tiempos de ejecución relativamente grandes resultados que distan bastante de las soluciones óptimas de las instancias, luego es preferible la heurística del vecino más cercano.
- El algoritmo de Christofides presenta grandes tiempos de ejecución en la práctica debido a la calidad de sus soluciones (las soluciones de este algoritmo son, a lo sumo, tres medios de la solución óptima).

Instancia	Longiud de la ruta de la heurística	Longitud de la solución real
4	35798.402	29368
8	7575.2876	6110
9	8194.613	6528
11	24630962	18660188
15	26856.387	21282
16	29155.04	22141
17	33609.863	26524

Cuadro 3.2: Análisis de la heurística del vecino más cercano

Instancia	Tiempo (en segundos)	Longiud de la ruta del algoritmo	Longitud de la solución real
2	923.056635	16098.492	3916
3	1198.425305	637724.1	126643
4	1859.3916367	127488.375	29368
5	154.877072	10609.441	7542
6	608.9938697	243789.95	118282
7	642.9551292	78301.37	26130
8	417.8311339	15336.689	6110
9	509.0600941	22170.756	6528
10	753.2785231	126987.02	29437
12	537.6101316	486.0849	426

Cuadro 3.3: Análisis del algoritmo genético

Apéndice A

Definiciones de Teoría de la Complejidad

En este apéndice vamos a definir algunos conceptos de la Teoría de la Complejidad utilizados a lo largo de la memoria:

En primer lugar es necesario definir el concepto de máquina de Turing, aunque sea de manera informal, pues es la base de toda la Teoría de la Complejidad.

Definición 7. Una **Máquina de Turing** es un dispositivo teórico que modela matemáticamente a una máquina que opera mecánicamente sobre una cinta. Dicha cinta es (teóricamente) infinita en ambas direcciones y sobre ella hay escritos símbolos que la máquina de Turing es capaz de reconocer, aparte de escribir otros.

La particularidad de las máquinas de Turing es que, a pesar de su sencillez, son capaces de describir cualquier operación o algoritmo que pueda implementarse en una computadora moderna, por lo que es una de la bases de las Ciencias de la Computación.

Ahora veremos algunas ideas cuyo estudio se basan en las máquinas de Turing.

Definición 8. Sea M una máquina de Turing determinista que para sobre cualquier dato de entrada. El tiempo de computación de M es una función $f : \mathbb{N} \to \mathbb{N}$ tal que $f(n)$ es el máximo número de pasos que M usa en cualquier dato de entrada de longitud n. Si $f(n)$ es el tiempo de computación de M, decimos que M computa en tiempo $f(n)$ y que M es una máquina de Turing de tiempo $f(n)$.

Debido a que el tiempo de computación exacto es a menudo una expresión compleja se suele usar una estimación de este. Una de las formas usuales de hacerlo es estudiando el tiempo de computación en instancias de entrada de gran tamaño, en lo que se llama el **Análisis Asintótico**. Este análisis asintótico considera únicamente el término de mayor orden de la expresión del tiempo computacional, despreciando todos los demás.

Por ejemplo, en la expresión $f(n) = 6n^3+2n^2+20n+45$ el término a considerar es $6n^3$, mientras que todos los demás son despreciados. Si también despreciamos el coeficiente 6 decimos que f es asintóticamente como mucho n^3.

La **Notación asintótica** o **Notación de la O-grande** para describir esta relación es $f(n) = O(n^3)$.

Definición 9. Sean f y g funciones $f, g : \mathbb{N} \to \mathbb{R}^+$. Decimos que $f(n) = O(g(n))$ si existen enteros positivos c y n_0 tales que para todo entero $n \geq n_0$, $f(n) \leq cg(n)$.

Cuando $f(n) = O(g(n))$ decimos que $g(n)$ es una **cota superior** de $f(n)$, o más concretamente, $g(n)$ es una **cota superior asintótica** de $f(n)$.

De forma parecida se puede definir la **notación de la o-pequeña**:

Definición 10. Sean f y g funciones $f, g : \mathbb{N} \to \mathbb{R}^+$. Decimos que $f(n) = o(g(n))$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

En otras palabras, $f(n) = o(g(n))$ significa que para cualquier número real $c > 0$, existe un número n_0 tal que $f(n) < cg(n)$ para todo $n \geq n_0$.

Definición 11. Sea un lenguaje L y un dato de entrada m de tamaño n, decimos que L es **Decidible** con tiempo de computación $O(f(n))$ si el tiempo de computación que toma comprobar si L para o no sobre m es $f(n)$.

Definición 12. Sea $t : \mathbb{N} \to \mathbb{R}^+$ una función, se define su clase de tiempo de computación, $TIME(t(n))$, como el conjunto de todos los lenguajes decidibles por una máquina de Turing con tiempo de computación $O(t(n))$.

Definición 13. Definimos P como la clase de todos los lenguajes que son decidibles en tiempo polinomial en una máquina de Turing determinista de cinta única.

Es decir, $\mathsf{P} = \bigcup_k TIME(n^k)$.

Definición 14. Un **Verificador** de un lenguaje A es un algoritmo V, donde $A = \{w : V$ acepta (w, c) para alguna cadena $c\}$.

El tiempo de un verificador se mide únicamente en términos de w, por lo que un **Verificador de Tiempo Polinomial** funciona en tiempo polinomial en la longitud de w. Un lenguaje A es **Polinomialmente Verificable** si tiene un verificador de tiempo polinomial.

Definición 15. NP es la clase de los lenguajes que tienen verificadores de tiempo polinomial.

Tras esta introducción previa estamos preparados para entender el problema principal de la Teoría de la Complejidad:

Definición 16. Definimos el problema $\mathsf{P} = \mathsf{NP}$ como el problema de determinar si todos los lenguajes pueden ser verificados en tiempo polinomial.

Existen también ciertos problemas especialmente interesantes en la clase NP (y muy relacionados con la hipótesis P = NP) cuya complejidad individual está relacionada con la de toda su clase. Si un algoritmo en tiempo polinomial existiera para cualquiera de esos problemas, todos los problemas de la clase NP serían resolubles en tiempo polinomial. Estos problemas son los conocidos como **NP-completo**.

Definición 17. Un problema Q es **NP-duro** si todos los problemas en NP se pueden reducir en un tiempo polinomial a Q. Ello no implica que el propio Q deba ser NP.

Apéndice B

Definiciones de Teoría de Grafos

A continuación vamos a definir una serie de conceptos de Teoría de Grafos que hemos utilizado a lo largo de la memoria:

Definición 18. Un **Grafo Dirigido** G consiste de dos conjuntos:

- $V(G)$: un conjunto no vacío cuyos elementos son llamados **Vértices** de G.
- $E(G)$: un conjunto de pares ordenados de vértices llamados **Aristas** de G.

Nótese que por la definición se tiene que la arista (u, v) no es equivalente a la arista (v, u). Es decir, el orden en que son listados los vértices indica la **Dirección** de la arista.

Definición 19. Un grafo G para el cual se tiene que las aristas (u, v) y (v, u) son equivalentes es llamado **Grafo No Dirigido**. Ello implica que en estos grafos es válido moverse del vértice u al vértice v así como del vértice v al u pasando por la arista (u, v).

Definición 20. Un grafo G es llamado **Multigrafo** si posee aristas repetidas (también conocidas como **Aristas Múltiples** o **Aristas Paralelas**).

Definición 21. Sea G un grafo. Los vértices u, $v \in V(G)$ se denominan **Adyacentes** si existe una arista $e \in E(G)$ tal que $e = (u, v)$. En tal caso, u y v son llamados **Puntos Finales** o **Extremos** de e, y además se dice que e conecta u y v. Además, la arista e se dice que es **Incidente** a cada uno de sus puntos finales u y v.

Definición 22. El **Grado de un Vértice** v en un grafo G, denotado como $deg(v)$, es igual al número de aristas en G que contienen a v, es decir, aquellas que son incidentes a v.

Definición 23. Se dice que un vértice es **Par** o **Impar** de acuerdo a si su grado es un número par o impar, respectivamente.

Definición 24. Considérese un grafo G con sus conjuntos $V(G)$ y $E(G)$. Un grafo H con sus conjuntos $V(H)$ y $E(H)$ es llamado un **Subgrafo** de G si los vértices y aristas de H están contenidas en los conjuntos de vértices y aristas de G.

Definición 25. Sea U un conjunto de vértices tal que $U \subset V(G)$. $G\{U\}$ denota el subgrafo de G cuyas aristas son aristas de G y sus vértices están en U. A $G\{U\}$ se le llama el **Subgrafo Inducido por el Conjunto de Vértices U**.

Definición 26. Si v es un vértice en G, entonces $G \setminus v$ es el subgrafo de G obtenido al eliminar a v de G y al eliminar a todas las aristas en G que contienen a v.

Definición 27. Si e es una arista en G, entonces $G \setminus e$ es el subgrafo obtenido al eliminar la arista e de G.

Definición 28. Una **Ruta** en un grafo G consiste de una secuencia alternante de vértices y aristas de la forma $(v_0, e_1, v_1, e_2, v_2, \ldots, e_{n-1}, v_{n-1}, e_n, v_n)$. En donde cada arista e_i contiene los vértices v_{i-1} y v_i (los cuales aparecen antes y después de e_i en la secuencia. Al número n de aristas se le denomina la **Longitud de la Ruta**. Por simplicidad, se denota a una ruta solo por la secuencia de sus vértices, asumiendo que efectivamente existen las aristas que conectan a dos vértices consecutivos dentro de la misma.

Definición 29. Se dice que una ruta es **Cerrada** si $v_0 = v_n$. Una **Ruta Simple** es aquella en la cual todos sus vértices son distintos. Una ruta en la cual todas las aristas son distintas es llamada una **Vía**. Un **Circuito** es una ruta cerrada de longitud mayor o igual que 3 en la cual todos los vértices son distintos excepto $v_0 = v_n$.

Definición 30. Se dice que un grafo G es **Completo** si para cualquier par de vértices u, v en $V(G)$, $u \neq v$, existe en $E(G)$ la arista (u, v) que los conecta.

Definición 31. Un grafo G es llamado **Grafo Pesado** o **Grafo Ponderado** si a cada arista e de G le es asignado un número no negativo $w(e)$ llamado el **Peso** de e.

Definición 32. El **Peso de una Ruta** en un grafo pesado de G es definido como la suma de los pesos de las aristas en la ruta.

Definición 33. Un grafo G es **Conectado** si existe una ruta entre cualesquiera dos de sus vértices.

Definición 34. Una **Ruta Euleriana** en un grafo conectado G es una ruta que pasa por cada arista de G exactamente una vez, aunque puede visitar a un vértice en más de una ocasión. Si un grafo tiene una ruta Euleriana y además ésta es una ruta cerrada entonces se tiene que tal ruta de hecho es un **Circuito Euleriano** y el grafo es caracterizado como **Grafo Euleriano**.

Teorema 35. *Un grafo conectado es Euleriano si y solo si cada uno de sus vértices tiene grado par.*

Definición 36. Un **Circuito Hamiltoniano** en un grafo conectado G es un circuito que contiene a todos los vértices de G. Si G admite un circuito Hamiltoniano entonces G es llamado **Grafo Hamiltoniano**.

Definición 37. A un grafo T se le llama **Árbol** si es conectado y además T no tiene circuitos.

Definición 38. Un subgrafo T de un grafo conectado G es llamado un **Árbol de Expansión** de G si T es un árbol y además T incluye a todos los vértices de G, es decir, $V(T) = V(G)$. En el caso de que G sea ponderado, un árbol de expansión T de G es el **Árbol de Expansión Mínimo** de G si la suma de los pesos de sus aristas es el mínimo de todos los árboles de expansión de G.

Definición 39. Un **Emparejamiento Perfecto** de G es un subconjunto M de las aristas de G tal que cada vértice de G es incidente exactamente con una de las aristas de M. En el caso de que G sea ponderado, un emparejamiento perfecto M de G es el **Emparejamiento Perfecto de Mínimo Coste** si el peso total de M, es decir, la suma de los pesos de las aristas de M es la menor de todos los emparejamientos perfectos de G.

Bibliografía

[1] Aguila, R.P. *Una introduccion a las matematicas discretas y teoria de grafos.* El Cid Editor, 2013.

[2] Applegate, D.L., Bixby, R.E., Cook, W.J., and Chvatal, V. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2007.

[3] Avis, D. A survey of heuristics for the weighted matching problem. *Networks* (1983).

[4] Bentley, J.L. Multidimensional binary search trees used for associative searching. *Communications of the ACM* (1975).

[5] Bentley, J.L. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing* (1990).

[6] Friedman, J.H., Bentley, J.L., and Finkel, R.A. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software* (1977).

[7] Gutin, G., and Punnen, A. *The Traveling Salesman Problem and its Variations.* Kluwer Academic Publishers, 2002.

[8] Kao, M.Y. *Encyclopedia of Algorithms.* Springer, 2008.

[9] Sipser, M. *Introduction to the Theory of Computation.* PWS Publishing, 1997.