Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales

Multi-TSP para robótica cooperativa

Autor: Jesús Caballero Rodríguez

Tutor: Begoña C. Arrue Ullés

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales

Multi-TSP para robótica cooperativa

Autor:
Jesús Caballero Rodríguez

Tutor:
Begoña C. Arrue Ullés
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Multi-TSP para robótica cooperativa

Autor: Jesús Caballero Rodríguez

Tutor: Begoña C. Arrue Ullés

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

Resumen

El objetivo principal de este trabajo consiste en adoptar un enfoque propio para resolver el problema del viajero, mediante el cual una persona se plantea recorrer una serie de ciudades, desde un origen hasta llegar a un destino, recorriendo la menor distancia posible a lo largo de todo el recorrido. Solo se visitaría una vez cada ciudad.

Además de resolver la versión más simplificada de este problema, se abordará este mismo caso si queremos que varios viajeros se repartan todo el recorrido. Todo ello buscando la mejor solución con los menores requerimientos en cuanto al coste computacional, además de la mayor simplicidad posible en el código que resuelva el problema.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

A^*	Conjugado
c.t.p.	En casi todos los puntos
c.q.d.	Como queríamos demostrar
■	Como queríamos demostrar
e.o.c.	En cualquier otro caso
e	número e
IRe	Parte real
IIm	Parte imaginaria
sen	Función seno
tg	Función tangente
arctg	Función arco tangente
sen	Función seno
$\sin^x y$	Función seno de x elevado a y
$\cos^x y$	Función coseno de x elevado a y
Sa	Función sampling
sgn	Función signo
rect	Función rectángulo
Sinc	Función sinc
∂y ∂x	Derivada parcial de y respecto
$x_\circ$	Notación de grado, x grados.
Pr(A)	Probabilidad del suceso A
SNR	Signal-to-noise ratio
MSE	Minimum square error
:	Tal que
<	Menor o igual
>	Mayor o igual
\	Backslash
⇔	Si y sólo si

1 INTRODUCCIÓN

1.1. Antecedentes

El problema del viajero, abreviado con las siglas "TSP" en inglés (Travelling Salesman Problem), es un problema de optimización mediante el cual se pretende minimizar los costes empleados en sucesivas operaciones.

El ejemplo más claro de optimización, como el propio nombre del problema indica, trata de reducir la distancia recorrida por un viajero a lo largo de su recorrido por varias ciudades. El "coste" a minimizar sería la distancia entre ciudades, mientras que las operaciones mencionadas anteriormente serían las visitas a las ciudades por las que pasa el viajero.

Hay que indicar que se evitará pasar por la misma ciudad dos veces, y que además se pretende tener claramente definidos tanto destino como inicio del recorrido.

Este problema de optimización posee multitud de aplicaciones, en función de lo que decidimos que será tanto el "coste" como las "ciudades" por las que pasa el "viajero".

Algunos ejemplos podrían consistir, por ejemplo, en minimizar el tiempo (coste) que requiere un cierto brazo robótico en moverse a lo largo de posiciones delimitadas en su espacio de trabajo, en las cuales dicho robot debe realizar operaciones determinadas, como agarrar piezas (ciudades). También se puede pretender minimizar el tiempo (coste) que se emplea para almacenar piezas en un determinado almacén en función de los puestos que se encuentran disponibles para su almacenaje, y del tiempo necesario para su correcto posicionamiento en los puestos de almacenaje (ciudades).

1.2. Objetivos

El objetivo fundamental de este trabajo consiste en resolver el problema de optimización propuesto de la manera más eficiente posible, teniendo en cuenta tanto el coste computacional necesario para resolver el problema como lo viable que es la solución obtenida; hay que llegar a un compromiso entre el tiempo de resolución empleado por el software y el grado de validez que tiene la solución obtenida.

La primera parte del trabajo consistirá en resolver el problema del TSP, considerando tan sólo ciudades intermedias por las que el viajero debe pasar y tan sólo un destino y una ciudad de comienzo.

Después, se tratará de complicar el problema. Varios viajeros comenzarán su recorrido partiendo de una única ciudad de origen, para que cada uno siga su propio recorrido hasta llegar a varios destinos que estarán predefinidos. Esta variante del TSP se denomina mTSP ("Multiple Traveller Salesman Problem"), y en ella cada viajero sigue su propio recorrido hasta llegar al destino que le corresponda. Los requisitos fundamentales en este nuevo problema son:

-El recorrido más largo, de entre todos los realizados, debe ser lo más corto posible para que no se tarde demasiado en llegar a la ciudad destino asociada. Téngase en cuenta que el recorrido más largo será el más restrictivo para el caso concreto que se considera dentro del mTSP.

-Los recorridos realizados por los distintos viajeros han de repartirse el recorrido, por lo que no debe haber demasiada varianza entre las distintas distancias obtenidas.

-Cada ciudad destino tendrá asociada, al menos, una ciudad intermedia por la que debe pasar obligatoriamente. El problema del mTSP cobra sentido cuando quiero llegar a un determinado destino habiendo pasado por una o varias ciudades intermedias, aunque posteriormente se abordarán las variantes del MTSP que se han presentado.

1.3. Metodología

Se ha llevado a cabo la programación necesaria para resolver el problema, tanto del TSP como del mTSP, en el entorno software de Matlab. Se explicarán posteriormente las funciones empleadas para resolver el problema de optimización, y también se mostrarán algunas gráficas en las que aparece representado el mejor recorrido calculado para ese problema, teniendo en cuenta el compromiso entre el coste y la validez de la solución, ya explicado anteriormente.

1.4. Estructura de la memoria

En los próximos apartados, se tratará de explicar con detenimiento en qué consiste cada uno de los dos problemas de optimización presentados anteriormente, el TSP y el mTSP.

Comenzando por el TSP, se explicará en qué consiste, de qué manera se puede plantear el problema de la manera más simple y, una vez aclarado el planteamiento general del problema, se irá profundizando en los detalles de las funciones desarrolladas en Matlab. Poco a poco, iremos integrando funciones y scripts en otros de mayor nivel hasta llegar al que nos proporcione la solución que estamos buscando; la más óptima para un coste computacional razonable.

Posteriormente indagaremos en el mTSP: introduciremos brevemente el problema, aclararemos variables introducidas en el código del TSP relacionadas con este nuevo problema, y presentaremos las variantes del mTSP que se han considerado.

Cuando se profundice en cada uno de las variantes, se presentarán los nuevos programas que se fundamentarán en los desarrollados en el TSP.

Una vez explicado cada problema, se mostrarán también programas auxiliares creados exclusivamente para simplificar todo el código principal explicado anteriormente.

Al final de la memoria, se mostrarán ejemplos de cada variante del mTSP y también del TSP, recurriendo a un script principal que pide las coordenadas de las ciudades por pantalla, representa las mismas por pantalla y resuelve el problema. Dicho script se desarrollará para que sirva para cualquiera de los problemas explicados y resueltos con sus respectivas funciones.

Para concluir, se hablará de posibles mejoras en el código y en la metodología empleada para mejorar las soluciones propuestas del problema. También se comparará con otros métodos que pueden resolver el problema, a partir de lo cual será interesante discutir tanto las ventajas como los inconvenientes del método propuesto.

2 DESCRIPCIÓN DEL SISTEMA Y ALGORITMIA

A lo largo de este apartado, se explicará tanto el problema del TSP (Travelling Salesman Problem) como su variante, el mTSP (Multi Travelling Salesman Problem). Explicaremos detenidamente en qué consiste cada problema, se explondrá algún ejemplo para que el lector vaya comprendiendo el plantamiento propuesto, y al final de cada subapartado se mostrarán secciones de código que plasmarán todas las explicaciones consideradas anteriormente.

2.1 TSP

Como se ha dicho anteriormente, se empleará el entorno software de Matlab para programar y representar tanto el problema del TSP como el MTSP.

Se intentará explicar de la forma más clara posible la transición que se ha llevado a cabo, partiendo de la idea básica, hasta llegar al resultado final que se ha conseguido.

2.1.1 Introducción

A pesar de que la idea de recorrer diversas ciudades hasta llegar a un destino puede parecer bastante intuitiva para el ser humano, cuando es una máquina la encargada de dar el resultado más óptimo dentro de sus posibilidades no es algo trivial.

Si tuviésemos, además de un solo destino y un solo comienzo, tan sólo dos ciudades intermedias por las que pasar, no resulta complicado dar la solución óptima del recorrido mínimo; hay muy pocas posibilidades de ruta respetando las premisas necesarias (partir de un origen, recorrer las ciudades intermedias, y llegar a un destino). Incluso con tres ciudades intermedias puede resultar un problema relativamente sencillo a resolver por el ser humano, si se persigue el óptimo en nuestro recorrido.

Sin embrago, ¿qué hacer cuando se nos presenta un número relativamente elevado de ciudades intermedias? ¿Y si tengo 9 o 10 ciudades intermedias? No es en absoluto discutible que una persona proporcione una buena solución ante este problema, aunque dado el enorme número de posibilidades que ahora existen para planificar una ruta, una máquina podría dar una solución más optima que la proporcionada por la intuición humana, haciendo uso de los algoritmos apropiados.

2.1.2 Empleo de nodos

Si lo que nos interesa desde el primer momento es hallar la solución óptima (nos desentendemos del problema del coste computacional de momento), nos interesará estudiar todas las posibilidades de ruta que se nos presenta.

Considerando que tenemos x ciudades intermedias, entre las cuales no se incluye ni el destino ni el origen del recorrido, tendremos x! posibles recorridos que tendrá asociado, cada uno, una distancia. Como hemos señalado en varias ocasiones, nos interesa encontrar de entre los x! recorridos el de menor coste posible.

Para buscar este "menor recorrido", una opción bastante interesante es considerar estados con varios valores asociados, los cuales serán cruciales para saber qué recorrido estamos considerando de cara a la solución final.

Denominaremos "nodos" a dichos estados, y se le asignarán varias variables a cada uno. Algunos de los datos más básicos que manejarán pueden ser:

-Nivel: a medida que vayamos completando el recorrido, iremos creando nuevos nodos con niveles cada vez mayores.

-Distancia recorrida: distancia que hasta el momento se ha recorrido.

-Ciudades por las que he pasado: se almacenará en un vector índices, en el cual cada componente está

relacionada únicamente con una de las ciudades definidas en el espacio.

-Ciudades a las que puedo ir: se trata de las ciudades definidas por las que aún no he pasado. Hasta que no me encuentre en el nivel inmediatamente anterior al más elevado, el índice de la ciudad destino no se encontrará en este vector.

-Punto actual: ciudad en la que me encuentro actualmente.

2.1.3 Disposición y estructuración de nodos

Consideraremos cada nodo como una estructura, y a su vez distinguiremos dos tipos de nodo: inmaduro y maduro.

La diferencia fundamental entre ambos tipos de nodo radica en los datos que contiene cada una de las estructuras. Un nodo maduro, por ejemplo, almacena información relacionada con las ciudades a las que el viajero se puede desplazar desde su punto actual, mientras que un nodo inmaduro no sabe aún a qué ciudad podría desplazarse el viajero.

Antes de tratar en profundidad las funciones que nos permitirán manejar nodos (crearlos, madurarlos, etc), presentaremos el esquema de nodos al que recurriremos conforme vayamos avanzando en el problema del TSP.

Si, como ejemplo, tuviésemos tres ciudades intermedias entre destino y origen, el esquema tendría esta forma:

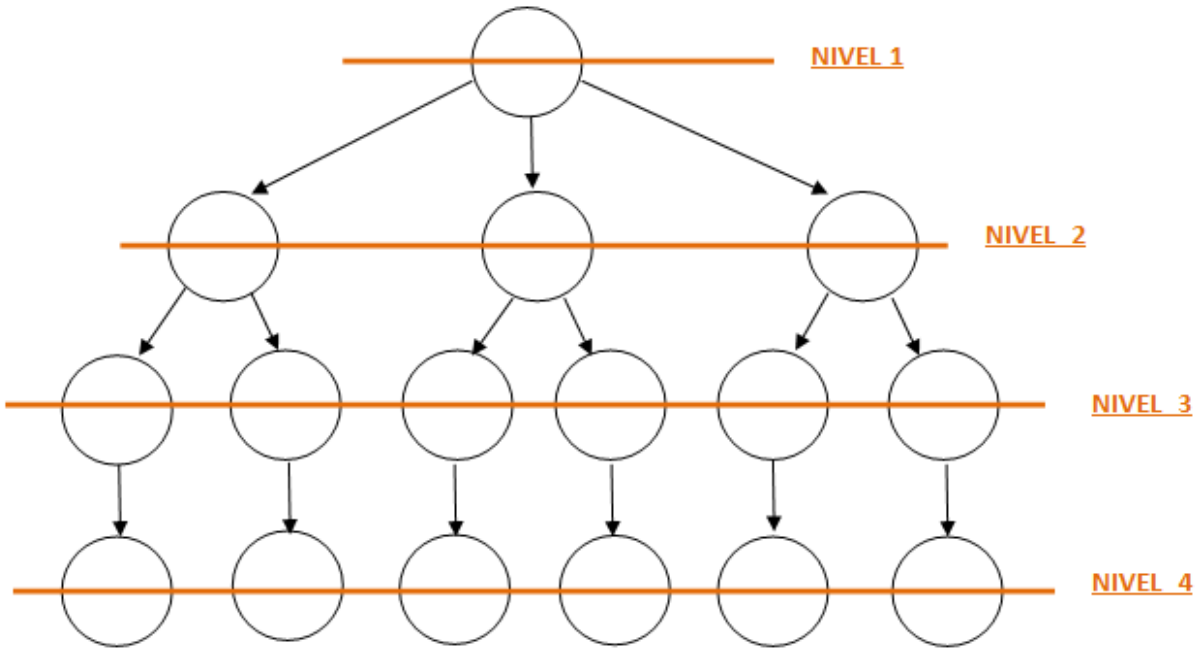

Figura 2-1. Árbol de nodos de tres ciudades intermedias.

Cuando estemos en la ciudad de inicio, nos encontraremos en el único nodo que existe en el nivel 1. Para tres ciudades intermedias, tenemos tres opciones a partir de este estado: ir a cualquiera de las tres ciudades intermedias primeras. Cada una de estas tres elecciones está representada por un nodo del nivel 2. Una vez hecha la elección, quedarán dos ciudades intermedias y, por tanto, dos elecciones, y así sucesivamente hasta llegar al nivel más alto. Este nivel más alto, el cuarto en el esquema anterior, representa el momento en el que el viajero ha llegado a la última ciudad intermedia llevando a cabo un recorrido. Tan sólo quedará alcanzar la ciudad destino y sumar la distancia correspondiente en su recorrido, pero como esto es algo que no implica elegir entre varias alternativas, se descarta su representación en el árbol de nodos de momento. Como señalamos antes, tratamos x! posibles recorridos, donde en este caso x es igual a 3 y, por ello, tenemos 6 recorridos posibles (6 nodos en el último nivel).

A partir de este punto, presentaremos las funciones creadas en Matlab que nos permiten manejar nodos.

2.1.4 Funciones y scripts principales

Consideraremos cada nodo como una estructura, y a su vez distinguiremos dos tipos de nodo: inmaduro y maduro.

La diferencia fundamental entre ambos tipos de nodo radica en los datos que contiene cada una de las estructuras. Un nodo maduro, por ejemplo, almacena información relacionada con las ciudades a las que el viajero se puede desplazar desde su punto actual, mientras que un nodo inmaduro no sabe aún a qué ciudad podría desplazarse el viajero.

2.1.4.1 "Devuelve_hijo"

Esta función tomará como entrada un nodo inmaduro, y devolverá a la salida este mismo nodo maduro además de uno de sus nodos hijos, que será inmaduro.

Diremos que un nodo es hijo de otro nodo cuando el primero represente un estado al que se puede llegar desde el padre, siempre y cuando difieran en solo un nivel.

A continuación se muestra una imagen de la caja negra que modelará esta función, así como una parte del árbol de nodos tanto a la entrada como a la salida de la función:

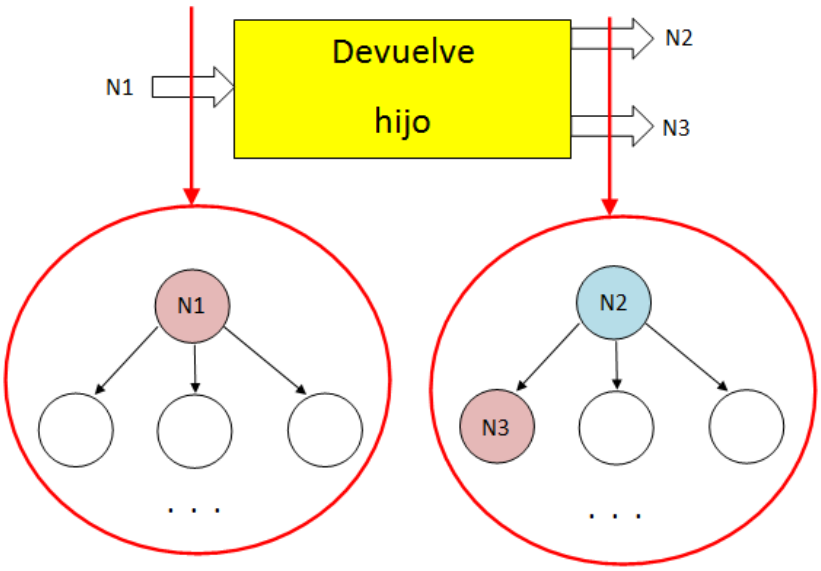

Figura 2-2. Esquema de la función "devuelve_hijo".

Los nodos rellenos de color rojo serán inmaduros, mientras que los azules serán maduros.

Cabe destacar que esta función puede tomar como entrada un nodo que esté maduro, siempre y cuando pueda crear un hijo que no se haya creado anteriormente. Por ejemplo, en el caso de que queramos aplicar esta función en el nodo N2, crearíamos otro hijo inmaduro y, además, obtendríamos a la salida un nodo maduro distinto de N2 pero que representa el mismo recorrido del viajero. ¿Por qué es distinto, si tanto el nodo de entrada como este son maduros? Porque el nodo maduro de salida sabe cuántos nodos hijos ha creado hasta ahora, y también sabe cuántos más puede crear. Esto se traduce en que, aunque tanto los nodo de entrada y salida maduros compartan el mismo tipo de estructura, sus valores van cambiando.

De esta forma se pueden crear varios hijos inmaduros a partir de un nodo inmaduro que se convierte en maduro, algo de lo que se ocupará la función "desglosa_nodo" que trataremos más adelante.

Código de "devuelve_hijo"

```
function [nodo,nodo_,counter]=devuelve_hijo(nodo,dots,dis,counter)

% nodo: nodo a la entrada de la función.
% A la salida se transforma en un nodo maduro.
% nodo_: nodo hijo que se crea.


% A cada hijo se le asigna un identificador, para diferenciarlos del
resto de nodos creados.
nodo_.indicador=counter; counter=counter+1;

% Se les asigna nivel.
nodo_.nivel=1+nodo.nivel;
nodo_.definido=0;



%% BUSCO EL MENOR TRAYECTO DESDE NODO: DEFINO DICHO NODO

if (nodo.definido==1)
    % Si el nodo es maduro, no hago nada en este "if".
else
    % en el "for" voy guardando lo que dista el
    % punto actual de cada una de
    % las ciudades intermedias inexploradas
for i=1:length(nodo.punto_sig)
c=[(nodo.punto_act) (nodo.punto_sig(i))];
distancia=norm(dots(c(1),:)-dots(c(2),:));
nodo.camino(i)=distancia;
nodo.camino_activo(i)=1;
end

nodo.vivo=1; % Como tratamos un nodo inmaduro (definido=0), se
considera vivo.
nodo=ordena_nodo(nodo); %Ordena de menor a mayor distancia cada
vector de la estructura (de izda. a dcha.)

end

aux=ones(length(nodo.camino_activo));
```

```
  nodo.escoge=aux(1,:); % Siempre estara lleno de unos al principio

  %% ESCOJO UNO DE LOS CAMINOS DEL NODO

  % Le asigno al nodo hijo la menor distancia posible de las
consideradas
  % antes, siempre y cuando no sea uu hijo que hayamos creado antes.
  flag=0;
  for i=1:length(nodo.camino_activo)
      if (nodo.camino_activo(i)==1 && flag==0)
          flag=1;
          nodo.camino_activo(i)=0;
          nodo.escoge(i)=0;
          dist=nodo.camino(i);
          nodo_.dist_recorrida=dist+ dis;
          nodo_.punto_act=nodo.punto_sig(i);
      end
  end

  % Si el nodo padre no puede crear más hijos, no está vivo.
  if (devuelve_activo(nodo.camino_activo)==0)
      nodo.vivo=0;
  end

  nodo.definido=1;
  %% DEFINO LOS CAMINOS POSIBLES DEL NODO_, LAS CIUDADES A LAS QUE
PODRÁ IR

  nodo_.punto_sig=refresca_auxiliar_f((nodo.punto_sig).*(nodo.escoge));

  %% GUARDA EL RECORRIDO
   nodo_.recorrido=[nodo.recorrido,nodo_.punto_act];

  end
```

2.1.4.2 “Desglosa_nodo”

Esta función tomará como entrada un nodo inmaduro, y devolverá a la salida todos sus nodos hijos, además del nodo padre maduro, que estará muerto.

Como aparecía en las líneas de código de la función “devuelve_hijo”, un nodo se encuentra vivo siempre y cuando no se haya desglosado. Cuando esto ocurra y no haya más hijos que crear a partir de un nodo padre, el nodo padre estará muerto.

En el apartado dedicado a esta función se mostrarán con detenimiento los valores que conforma cada una de las estructuras de nodo (maduro e inmaduro). Lo haremos en este punto de la memoria para comprender con más facilidad el significado de cada uno de los valores, ya que iremos desglosando un nodo poco a poco.

2.1.4.2.1 Valores almacenados por la estructura “nodo_inmaduro”

-Nivel.

-Punto actual.

-Distancia recorrida.

-Punto siguiente (punto_sig).

-Recorrido.

-Indicador: se trata de un índice que, como se explicará más adelante, nos ayudará a crear nuevos nodos hasta llegar a buenos resultados de recorridos.

-Definido: vale 0 si el nodo es inmaduro, y 1 si es maduro.

2.1.4.2.2 Valores adicionales almacenados por la estructura "nodo_maduro"

Esta estructura almacenará los valores que también están contenidos en la estructura "nodo inmaduro". Sin embargo, además de dichos valores, almacenará una serie de vectores cuyo significado se intentará explicar con un ejemplo.

Supongamos que el viajero tiene que pasar por tres ciudades intermedias, cada una identificada con un solo índice entero. Si los índices son 3, 4 y 2, y estamos en el nodo inicial ya desarrollado, el vector "punto_sig" valdrá [2 3 4], ya que por simplicidad se ordenan los índices de menor a mayor. Nos encontramos en la ciudad de inicio, a la que siempre le corresponderá el índice 1.

¿A qué ciudad nos movemos? O, mejor dicho, ¿cómo exploramos las posibles soluciones?

Supondremos que las distancias entre el punto actual y el vector "punto_sig" son:

Tabla 2–1. Distancias de la primera iteración (unidades)

Distancia (1-3)	Distancia (1-4)	Distancia (1-2)
4.6	3	9

Se define el vector "**camino**" como los valores ordenados de menor a mayor de las distancias que se pueden sumar si nos movemos a un índice/ciudad determinado.

En este caso, camino vale [3 4.6 9].

Nos aseguraremos de que los recorridos más cortos estén situados lo más a la izquierda posible del esquema de nodos, de manera que dichos recorridos serán los primeros en conocerse. Por esto mismo, los nodos hijos estarán ordenados de izquierda a derecha según los índices del vector camino; primero se crea el nodo mediante el cual me muevo a la ciudad 4 y, por último, creo el nodo según el cual me he movido a 9. La ciudad 4 es la más cercana y la 9 es la más lejana a la ciudad actual, por lo que parece viable este ordenamiento.

Como tenemos que asegurarnos de que no se haya creado ningún nodo dos veces, hemos de asegurarnos de que cada nodo tenga memoria de alguna manera, para que "sepa" las opciones que se han explorado anteriormente.

Si inicialmente uso la función "desglosa_nodo" para desglosar el nodo inicial, crearé tres nodos que guardarán la ciudad correspondiente a la que me puedo desplazar, así como la distancia total que acumularía en mi viaje. Me debería desplazar a la ciudad 4 porque el viajero sumará la mínima distancia posible, teniendo como referencia el vector "**camino**". Sin embargo, si en un futuro quiero decidir a qué ciudad me muevo desde el nodo inicial y ya he explorado la ciudad de índice 4, en esta ocasión he de ir a la ciudad de índice 3, porque la de índice 4 ya ha sido explorada.

¿Cómo consigo que la función "devuelve_hijo", que estará dentro de "desglosa_nodo", sepa cuándo una ciudad ha sido o no explorada anteriormente? Definiré un vector de la misma longitud que "camino", inicialmente con todas sus componentes a 1. Llamo a dicho vector "**camino activo**", y cada índice del vector estará asociado al índice del vector "camino" en el mismo orden. De esta manera, la primera vez que use la función "devuelve_hijo" tomando como entrada el nodo inicial, podré a cero la primera componente de "camino_activo" porque me desplazaré a la ciudad 4, y el nodo actual pasará a ser aquel en el que me he movido a dicha ciudad.

Si volviese a usar la función con el nodo de inicio, cambiaría el vector "camino_activo" de [0 1 1] a [0 0 1], es decir, me muevo a la segunda distancia más corta que podría sumar y pongo el índice correspondiente a 0.

Con el vector "**escoge**", que se define también dentro de la función "devuelve_hijo", elegiremos el valor de "camino_activo" que pondremos a 0 para almacenar su correspondiente índice y, de la misma manera, la ciudad a la que nos desplazaremos.

Cuando el vector "camino_activo" sólo contenga ceros en un nodo, significara que éste no se puede desglosar más y consideraremos que no está vivo. Precisamente, definiremos un valor entero llamado "vivo" que a partir de ese momento valdrá 0 y, anteriormente, habrá valido 1.

2.1.4.2.3 Ilustración.

A continuación se muestra una imagen de la caja negra que modelará esta función, así como una parte del árbol de nodos en varios tramos de la función, a los que se han llamado A, B, C y D. El desglose del nodo inicial del árbol se ha hecho siguiendo el ejemplo del apartado en el que se explicaba la estructura "nodo maduro":

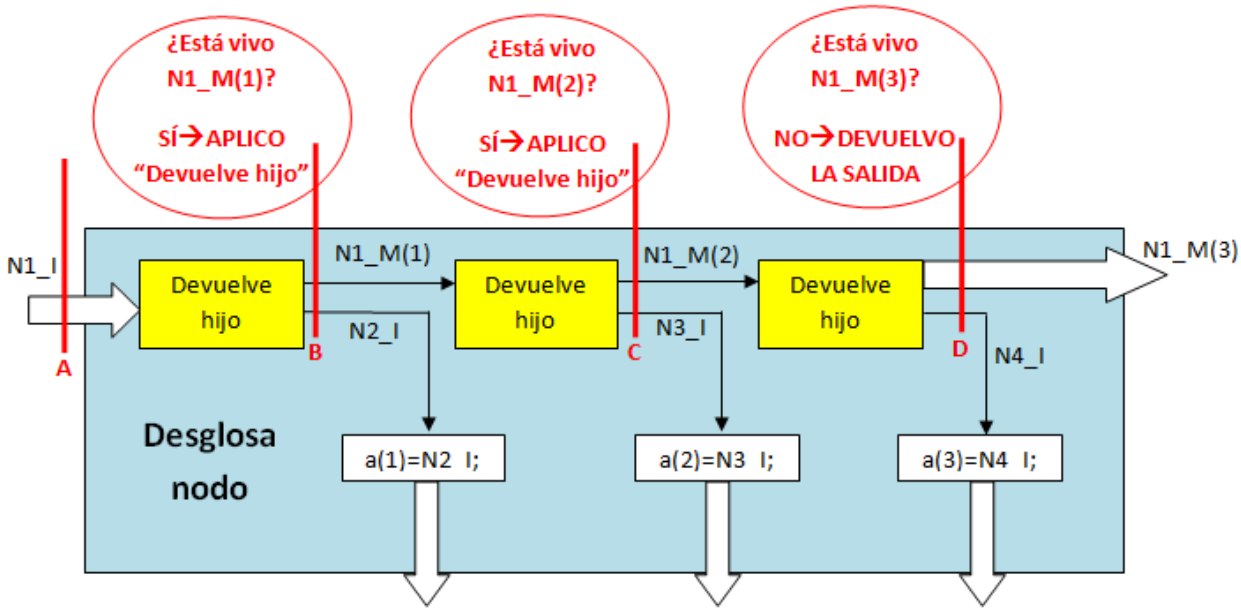

Figura 2-3. Esquema de la función "desglosa_nodo".

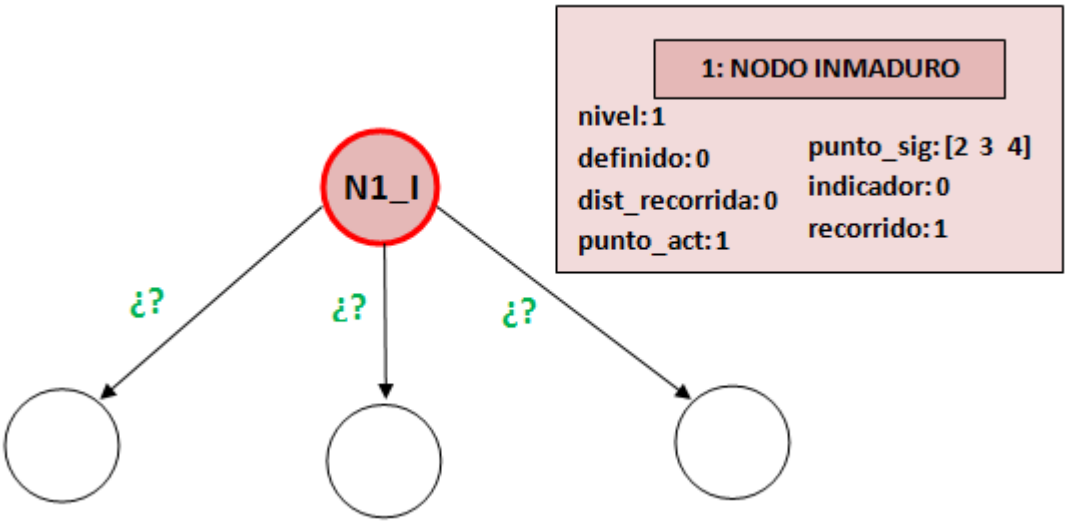

Figura 2-4. Sección A.

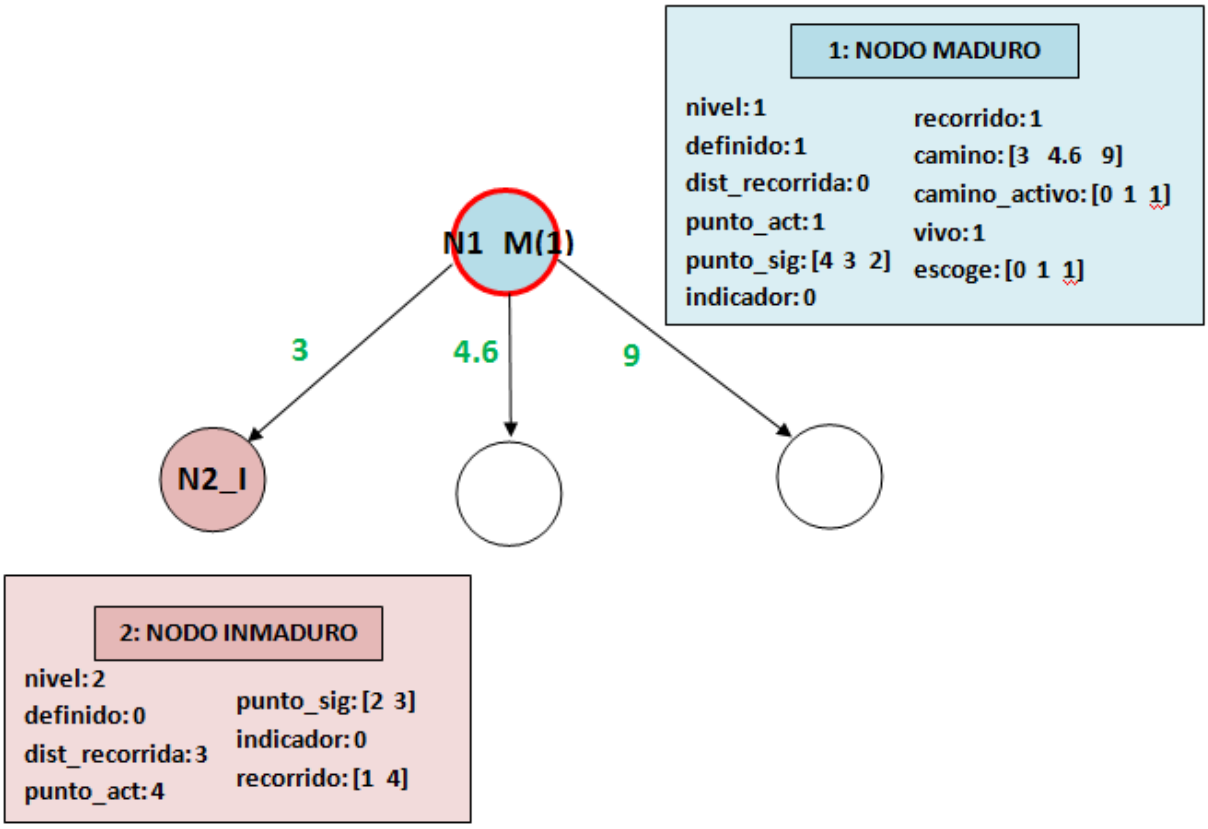

Figura 2-5. Sección B.

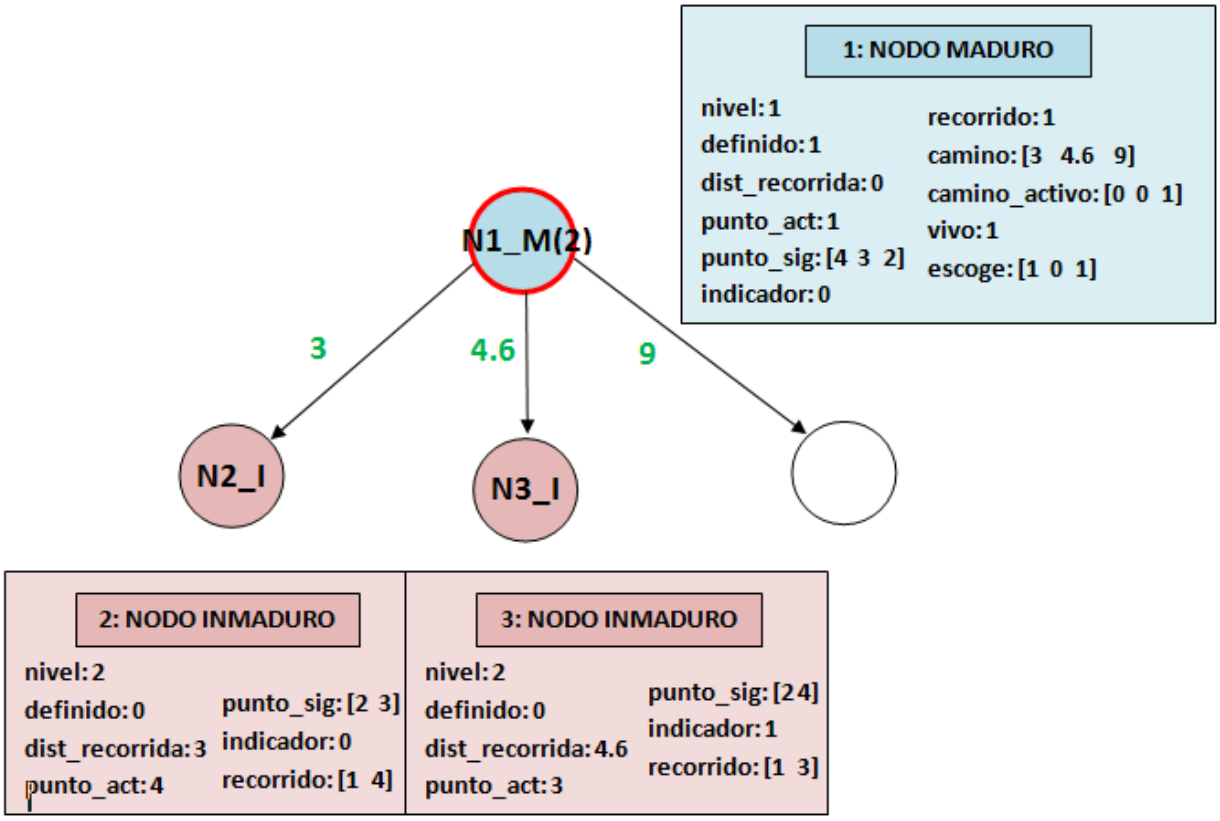

Figura 2-6. Sección C.

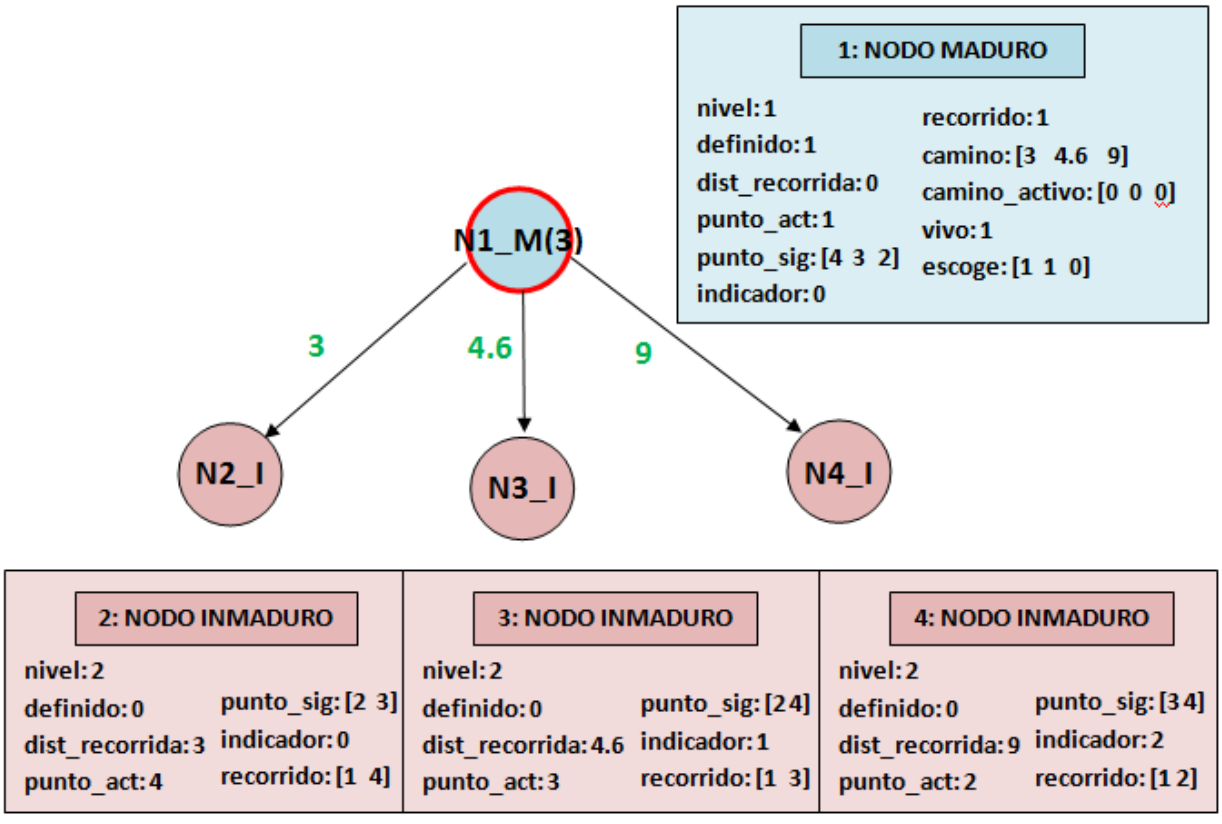

Figura 2-7. Sección D.

Una vez aclarado el propósito de la función, se muestra el código de la función empleada en Matlab:

Código de "desglosa_nodo"

```
 function
[a,nodo_1,counter]=desglosa_nodo(nodo_1,dots,distancia,counter)

  salir=0; contador=1;
 while (salir==0)
[nodo_1n,nodo_2,counter]=devuelve_hijo(nodo_1,dots,distancia,counter);
 a(contador)=nodo_2; contador=contador+1;
 if (nodo_1n.vivo==0)
    salir=1;
 end
 nodo_1=nodo_1n;
 end


  end
```

2.1.4.3 "Repetición"

2.1.4.3.1 Explicación

Ahora que tenemos claro lo que es "desglosar un nodo", tan solo queda preguntarse de qué manera recorrer el árbol de nodos desglosando cada uno de ellos. ¿Cuál sería la manera más eficiente de hacerlo?

Existen dos maneras de recorrer los nuevos nodos que se vayan creando, siguiendo una estructura LIFO o una estructura FIFO[1].

Si optásemos por la estructura "FIFO", recorreríamos los nodos según estos se van creando, en el orden que se muestra en el siguiente esquema:

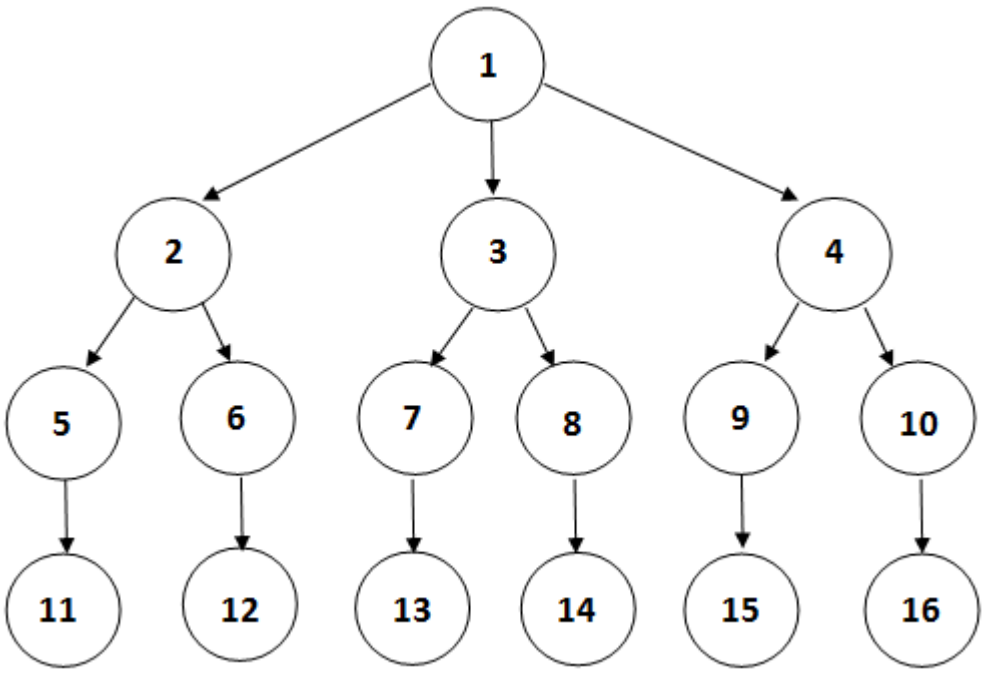

Figura 2-8. Estructura FIFO.

Si se optase por emplear esta estrategia, habría que definir todos los nodos que se encuentran en el árbol de nodos, ya que se van creando todos lo que existen nivel a nivel. Para un número de ciudades intermedias mayor que tres o cuatro, esto puede aumentar considerablemente el coste computacional.

Si por el contrario decidiésemos recorrerlas siguiendo una estructura "LIFO", iríamos recorriendo los nodos desglosando siempre el primer nodo creado del nivel inmediatamente superior. De esta manera, se generan recorridos completos con más rapidez que en el caso anterior, ya que antes prácticamente se tendrían que definir y recorrer todos los nodos que pudiésemos dado un número de ciudades intermedios y, sólo al final, sabríamos los x! recorridos.

Con esta nueva estrategia, vamos descubriendo nuevos caminos paulatinamente y, como veremos, nos ahorramos una gran cantidad de nodos a explorar:

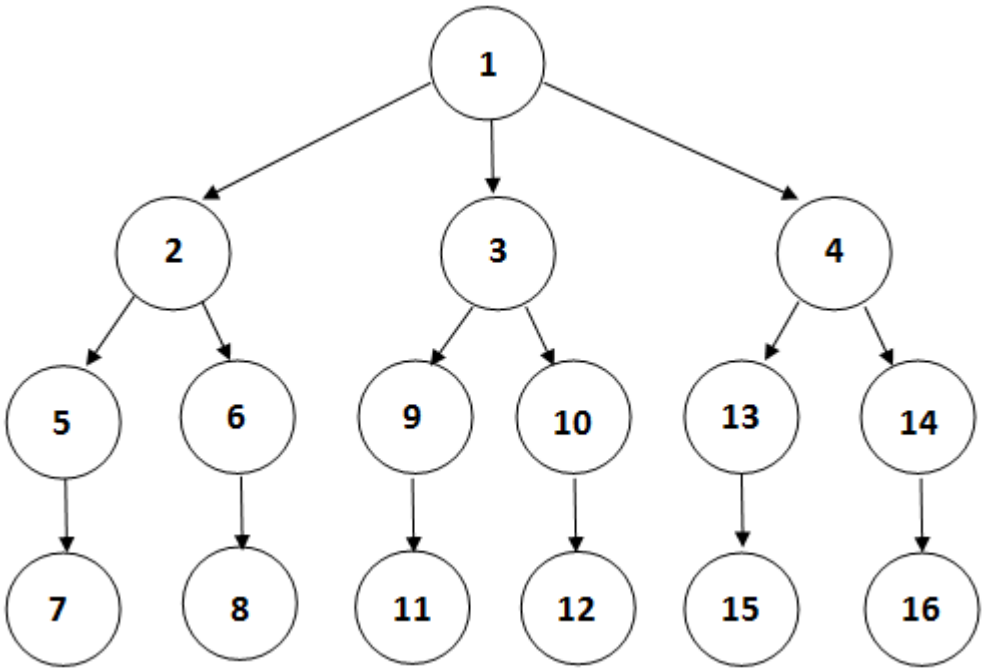

Figura 2-9. Estructura LIFO.

Comparándola con la estrategia "FIFO", en la "LIFO" podemos conocer recorridos completos realizados por el viajero antes de explorar todos los nodos situados en niveles inferiores. Para crear los nodos 7 y 8, por ejemplo, aún no hemos explorado los nodos 9, 10, 13 y 14 del nivel anterior, lo que nos reduce el coste de cálculo.

Además, se ha de tener en cuenta que se van conociendo antes los recorridos situados más a la izquierda del árbol de nodos.

Si recordamos las figuras expuestas cuando explicamos la función "desglosa_nodo", los nodos hijos situados más a la izquierda eran los que menor distancia tenían entre el punto actual y el punto siguiente, por lo que el empleo de una estructura "LIFO" parece ser muy conveniente en nuestro caso.

Sin embargo, combinaremos ambas estrategias para crear una "estrategia híbrida", de manera que se recorran las ramas principales del árbol y, a su vez, se recorran las subrramas de izquierda a derecha. Así, se obtendrán antes las mejores soluciones garantizando al mismo tiempo que se considerarán aquellas que se encuentran en cada una de las subrramas.

La "estrategia híbrida" tendría esta forma:

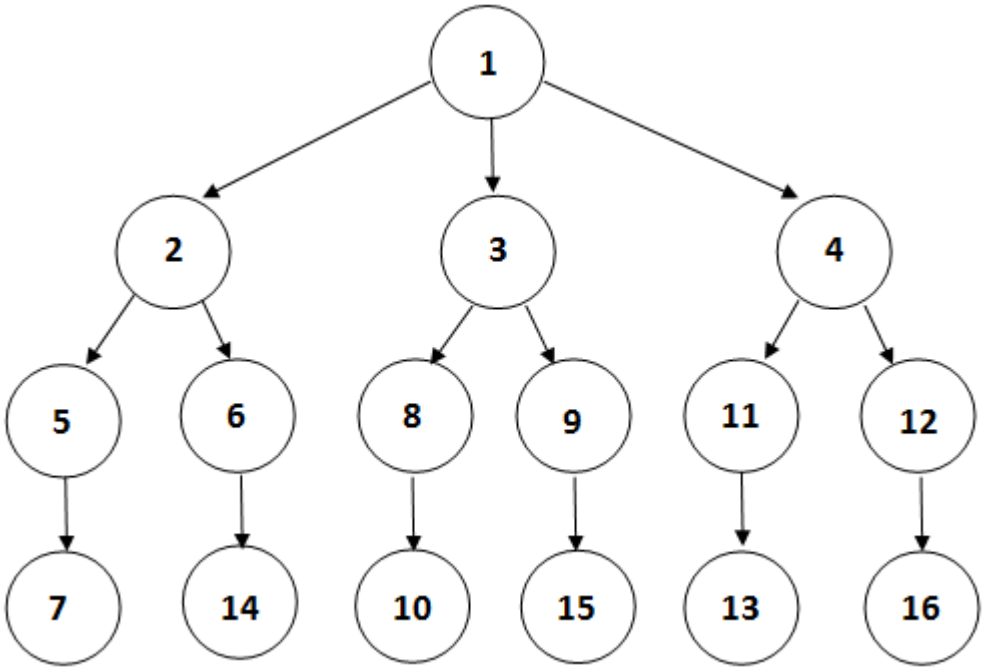

Figura 2-10. Estrategia híbrida.

La razón fundamental por la cual se considerará esta estrategia es, aparte de lo señalado anteriormente, que se programa de una manera relativamente sencilla. Cada nodo tendrá asignado un índice, y estos se irán asignando de menores a mayores valores conforme se van creando. De esta manera, tan solo hay que conseguir que se desglosen los nodos inmaduros en orden de asignación de índices creciente.

El script "repetición" se tomará un nodo inmaduro a la entrada, el de menor índice. A partir de dicho nodo, se irá desglosando su hijo con la distancia más corta entre punto actual y siguiente, luego el nieto con la distancia más corta… y así hasta llegar a un nodo situado en el último nivel.

Básicamente, el script se encargará de partir de un nodo inmaduro hasta completar el recorrido con distancias más favorables entre ciudad y ciudad.

La idea es llevar a cabo varias iteraciones del script "repetición" variando el índice del nodo inmaduro que toma a la entrada. Esto se explicará con más detalle en el código.

2.1.4.3.2 Ilustración.

A continuación se muestra, paso a paso, cómo queda un árbol de nodos de tres ciudades intermedias aplicando la estrategia híbrida explicada anteriormente.

En rojo están sombreados los nodos inmaduros, mientras que en azul se representan los nodos maduros. En verde se representan los nodos creados en el último nivel, que almacenan la distancia total en el recorrido escogido sin cosiderar la ciudad destino.

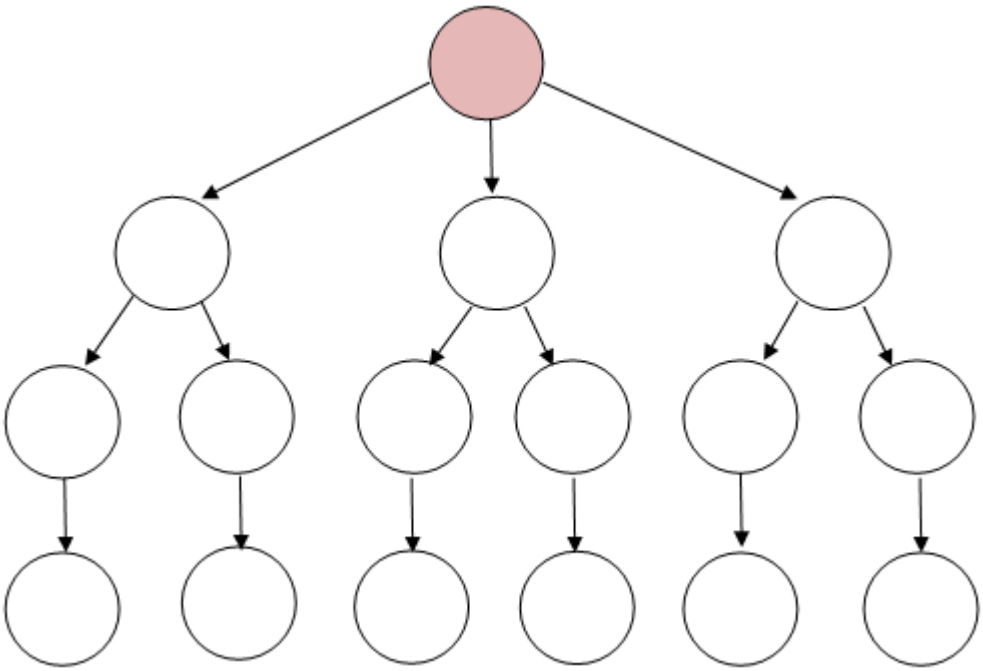

Figura 2-11. Iteración 1 de “repeticion”: Partimos del nodo inicial inmaduro.

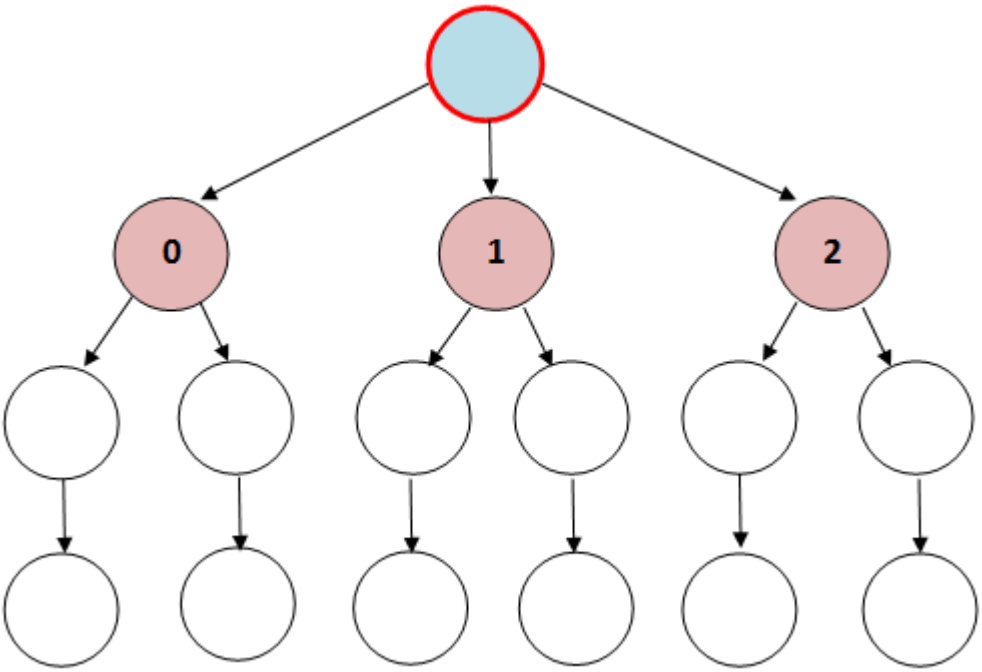

Figura 2-12. Iteración 1 de “repeticion”: Desglosamos el nodo inicial.

Desglosaré a continuación el nodo inmaduro 0, que me dará la menor distancia entre ciudad y ciudad (es el hijo de más a la izquierda).

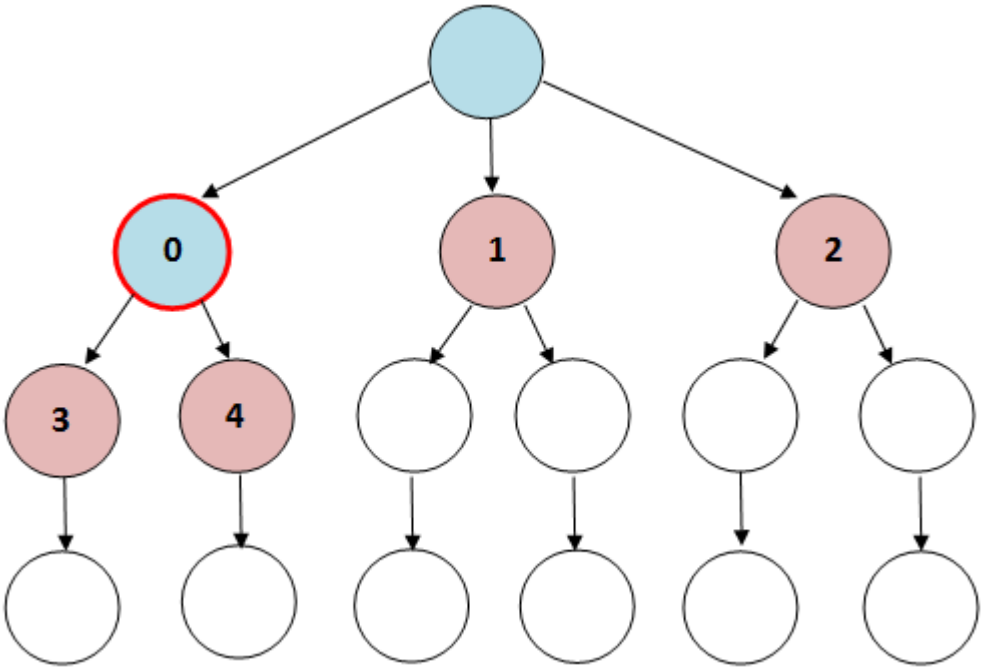

Figura 2-13. Iteración 1 de “repeticion”: Desglosamos el nodo 0.

Desglosaré a continuación el nodo inmaduro 3, que me dará la menor distancia entre ciudad y ciudad (es el hijo de más a la izquierda).

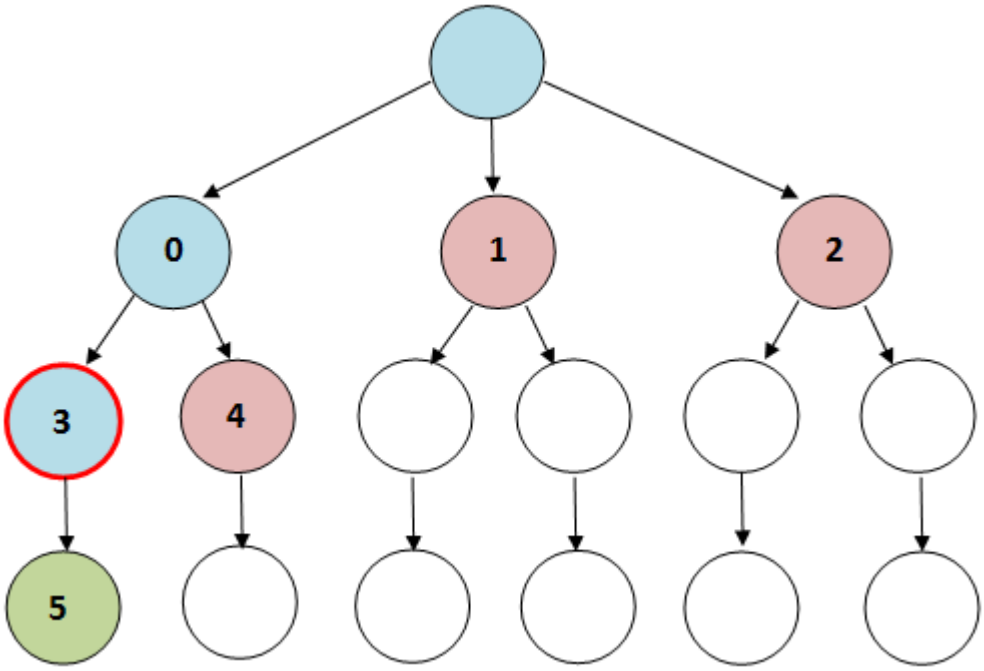

Figura 2-14. Iteración 1 de “repeticion”: Desglosamos el nodo 3.

Se acaba la primera iteración, ya que el viajero del nodo hijo no tiene punto siguiente al que desplazarse.

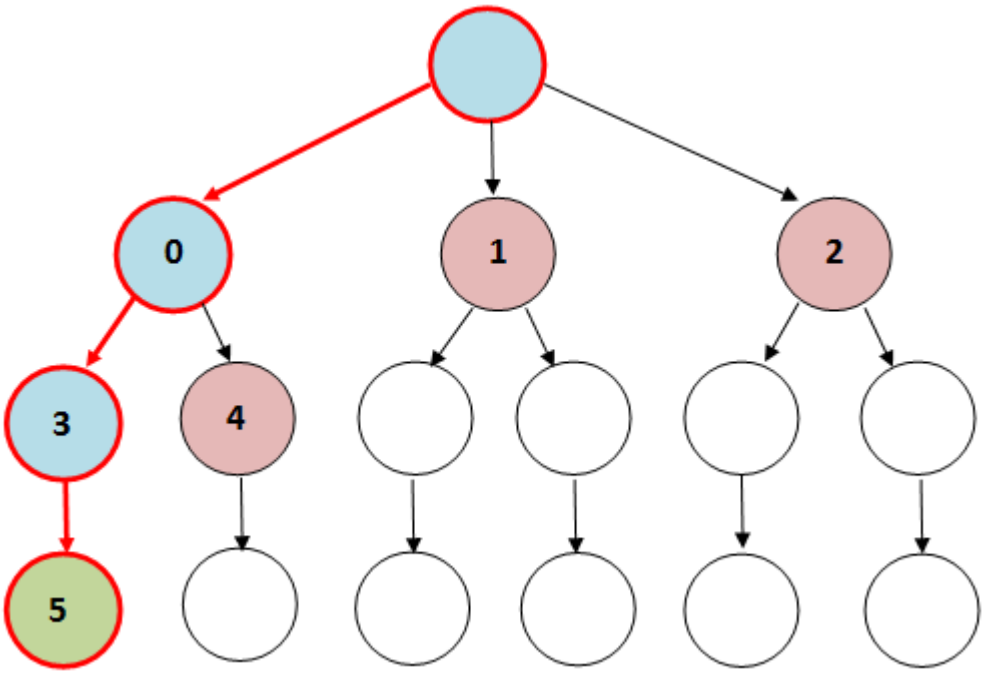

Figura 2-15. Resumen de la iteración 1 de “repeticion”.

La segunda iteración partirá del nodo inmaduro con menor índice, el 1.

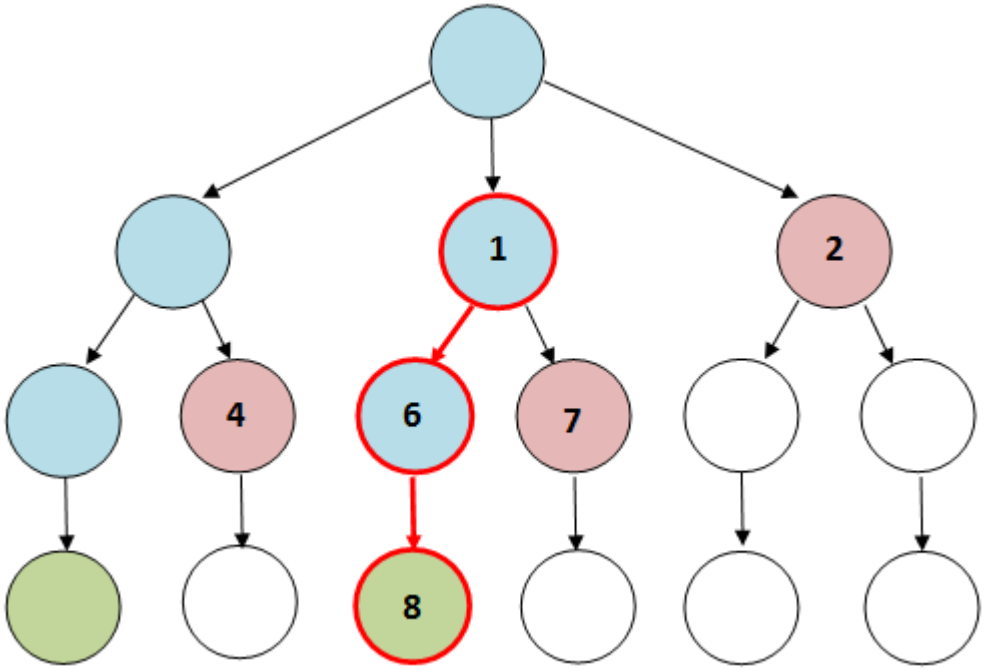

Figura 2-16. Resumen de la iteración 2 de “repeticion”.

La tercera iteración partirá del nodo inmaduro con menor índice, el 2.

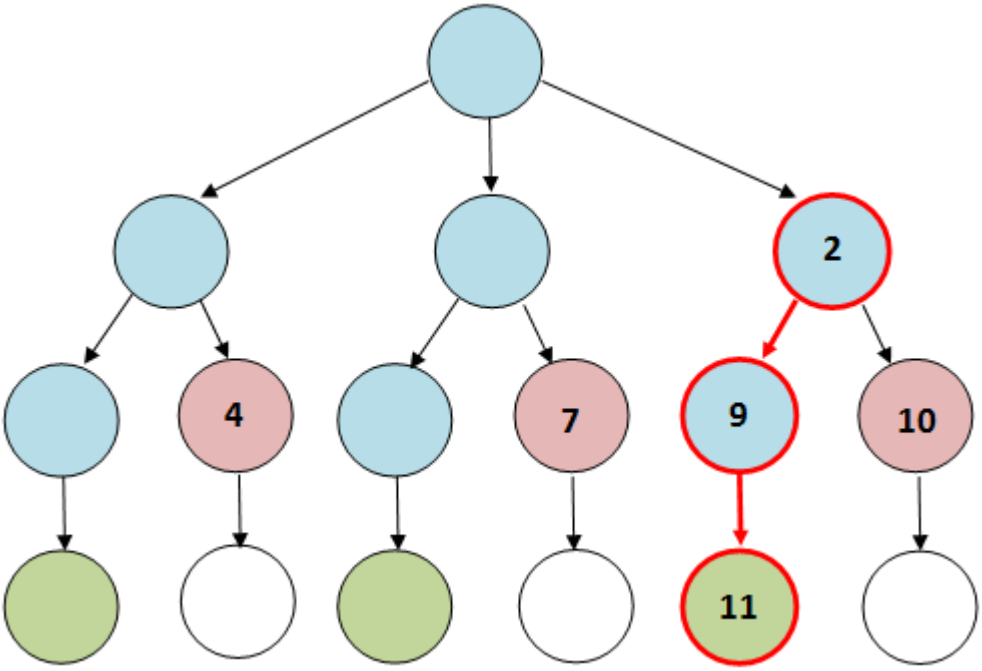

Figura 2-17. Resumen de la iteración 3 de “repeticion”.

La cuarta iteración partirá del nodo inmaduro con menor índice, el 4.

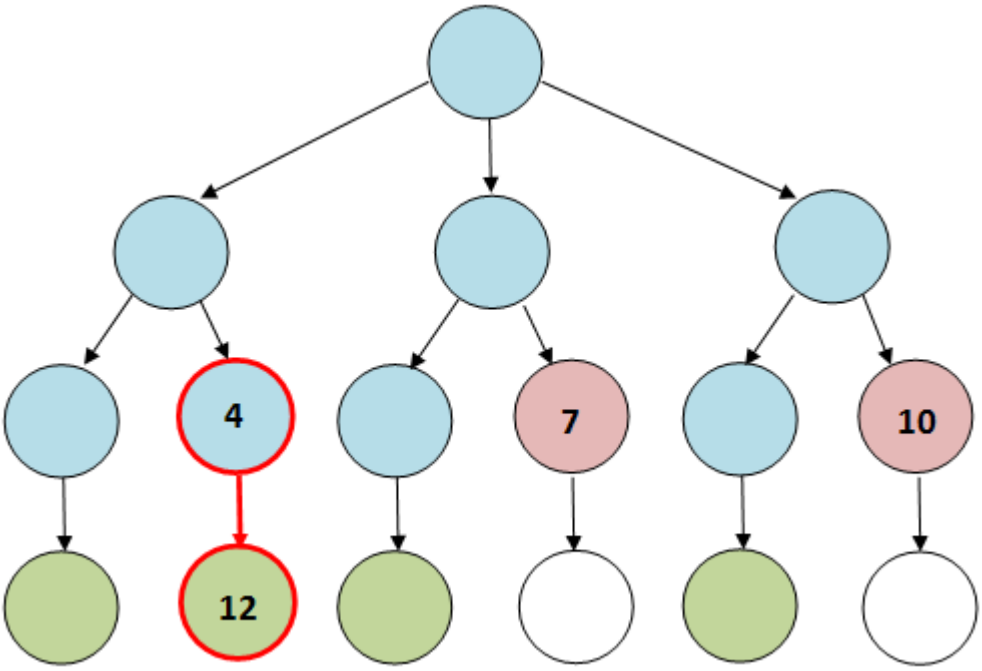

Figura 2-18. Resumen de la iteración 4 de “repeticion”.

La quinta iteración partirá del nodo inmaduro con menor índice, el 7.

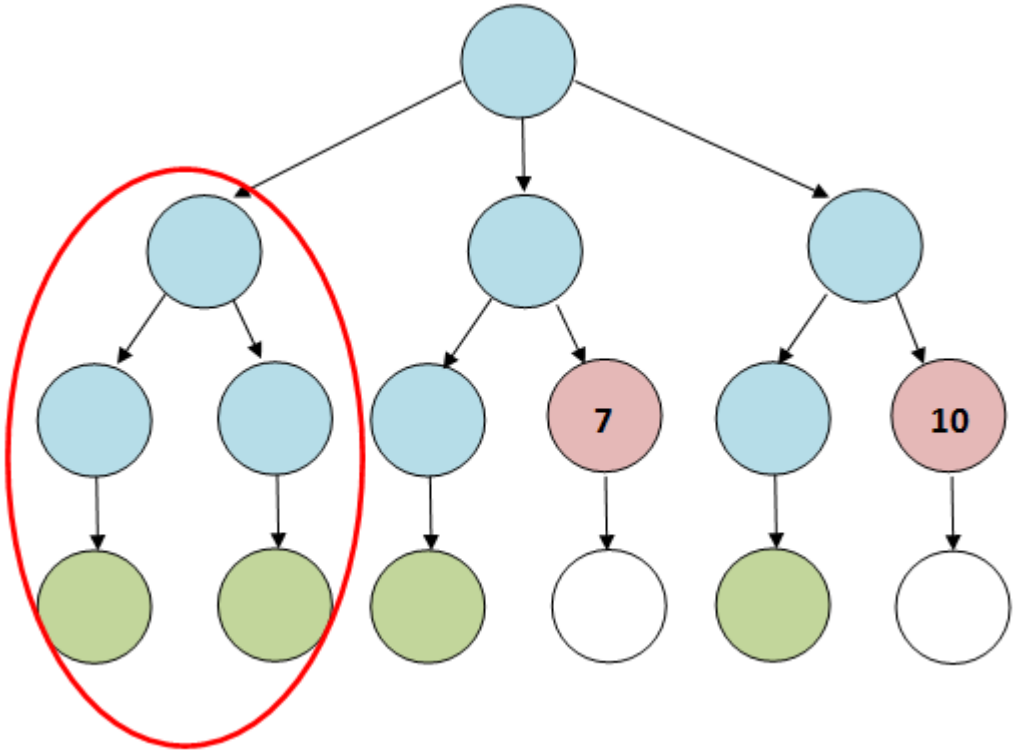

Figura 2-19. Resumen de "repetición" para la primera subrama.

Como se indicó anteriormente, se completan antes las ramas de la izquierda, que son las que aportarán las mejores soluciones.

Se muestra a continuación el código del script "repetición", con las indicaciones pertinentes escritas en verde:

Código de "repetición"

```
%% REPETICIÓN

% Almaceno en "nodo_1" el nodo a partir del cual empezará a desglosar
el script.

salir=0;
nodo_1=nodo_desglosable(INDICE);
distancia=nodo_1.dist_recorrida;

%% CAMBIO AQUI POR EL MTSP (se explica más adelante)
if (es_ultimo(nodo_desglosable(INDICE),dots,fin)>0)
    % no desgloso el nodo si está en el último nivel
    % que quiero
else
%%
while (salir==0)
   % desgloso los nodos del nivel inferior al que considero, y lo
defino completamente
[a,nodo_1n,counter]=desglosa_nodo(nodo_1,dots,distancia,counter);
```

```
  nodo=[nodo,nodo_1n];  % vuelco en "nodo" el nodo completamente
definido

  nodo_1=a(1); % nodo con distancia recorrida más corta en el tramo
  distancia=a(1).dist_recorrida;

  nodo_desglosable=[nodo_desglosable,a]; % vuelco los nodos inmaduros
aún no definidos pero creados a partir de "nodo_1".


  % Cuando el nivel del nodo llegue a un "tope", almaceno en
"nodo_final" la distancia del viaje y en "nodo_pre" el nodo inmaduro
que es padre del nodo final.

  %Usare "nodo_pre" para sacar la sucesión de ciudades en el orden en
que se recorrieron a partir de la distancia total del viaje.
  if (a(1).nivel>=(tope+1))

     for ii=1:length(a)
      nodo_pre=[nodo_pre,a(ii).dist_recorrida];

nodo_final=[nodo_final,(a(ii).dist_recorrida+norm(dots(a(ii).punto_act
,:)-dots(fin,:)))];
     end

      salir=1;
  end

  end

  end

  %% SECCIÓN DE CÓDIGO ENCARGADA DE REFRESCAR "INDICE" ENTRE
ITERACIONES.

  % El vector v almacenará los índices que no me interesa considerar
para elegir "INDICE".
  v=[];

  % recorremos los nodos desarrollados, "nodo".
  for i=1:length(nodo)
     if     (nodo(i).indicador==0)     ||     (nodo(i).indicador-nodo(i-
1).indicador)<0 || componente_rep(v,nodo(i).indicador)
     else
      % cuando el índice sea positivo, sea mayor que el del nodo
madurado justo antes o el índice no pertenezca a v, lo incluyo en el
vector:
      v=[v,nodo(i).indicador];
     end
  end

  flag=0;
  % Recorro los nodos inmaduros, "nodo_desglosable"
  for i=1:length(nodo_desglosable)
```

```
      % Cuando encuentre un nodo inmaduro con índice positivo, no
comparta indicador con los almacenados en "v" y tampoco sea un nodo
del último nivel (porque no se puede desglosar más)...
      if              (nodo_desglosable(i).indicador>0)              &&
(componente_rep(v,nodo_desglosable(i).indicador)==0)  &&  flag==0    &&
((isempty(nodo_desglosable(i).punto_sig))==0)
      %Almaceno el lugar que ocupa el nodo, y pongo su indicador a 0
para no considerarlo posteriormente en sucesivas iteraciones.
          INDICE=i;
          nodo_desglosable(i).indicador=0; flag=1;
      end
  end
```

2.1.4.4 "Busca_recorrido"

2.1.4.4.1 Explicación

El script "busca_recorrido" será el encargado de devolver el recorrido que se corresponda con la menor distancia obtenida de entre todas las iteraciones de "repetición" consideradas. A continuación se enseñará el código de dicho script, comentado en verde para que se entienda cómo se ha conseguido sacar el índice de cada ciudad y el orden en el que se recorren.

2.1.4.4.2 Código

```
  %% busca_recorrido

  % recorro "nodo_final", que almacena las diitancias de los distintos
  % recorridos considerados. Almaceno en "vital" el índice del de
distancia mínima.
  for i=1:length(nodo_final)
      if (min(nodo_final)==nodo_final(i))
          vital=i;
      end
  end
  % guardo en "MINIMO" la distancia menor considerada.
  MINIMO(C)=min(nodo_final); C=C+1;



  % Recorro todos los nodos inmaduros que se han creado anteriormente
(en "nodo_desglosable")

  % Impongo varias condiciones en el bucle "for": que el nodo esté en
el último nivel, y que la distancia total del recorrido sea la
distancia de ese nodo más lo que dista de la ciudad final (con un
márgen de error).
  aux=[]; error=10^-4;
  for i=1:length(nodo_desglosable)
     distancia=norm(dots(fin,:)-dots(nodo_desglosable(i).punto_act,:));

     c11=(isempty(nodo_desglosable(i).punto_sig));
     c1=nodo_desglosable(i).nivel==(tope+1);
     c2=(distancia-(nodo_final(vital)-nodo_pre(vital)))<error; % lo del
error funciona mejor
     c3=(nodo_desglosable(i).dist_recorrida==nodo_pre(vital));
```

```
  if (c1 && c3 && c2)
          aux=[aux,nodo_desglosable(i)];

  end

  end

  %% En el caso de que haya habido varias coincidencias, que quedo solo
con una (aux(1))
  if length(aux)>1
  aux=aux(1);
  end

  %% Represento gráficamente el camino

  for i=1:length(aux.recorrido)

      if i==length(aux.recorrido)
         plot([dots(aux.recorrido(i),1)
dots((fin),1)],[dots(aux.recorrido(i),2) dots((fin),2)])
      else
         plot([dots(aux.recorrido(i),1)
dots((aux.recorrido(i+1)),1)],[dots(aux.recorrido(i),2)
dots((aux.recorrido(i+1)),2)])
      end
  end


  %% Pongo a 0 los índices de las ciudades por las que he pasado.
  % Esto cobra sentido en el MTSP con ciudades auxiliares.

  for j=1:length(indices)

     if(componente_rep(aux.recorrido,indices(j)))
         indices(j)=0;
     end

  end
```

2.2 MTSP

2.2.1 Introducción

Una vaz resuelto el problema de TSP, trataremos de complicarlo un poco considerando, en esta ocasión, varios viajeros que parten de una misma ciudad de inicio, se reparten las ciudades intermedias definidas en el espacio de una manera u otra (en función de los casos que se consideren, explicados más adelante) y, además, se indagará en las secciones de código definidas hasta ahora para aclarar lñas variables que se emplearán en este nuevo problema, el mTSP.

2.2.2 Variable “tope”.

Las funciones y los scripts que se han mostrado hasta ahora también consideran el caso del mTSP, ya que aparece una variable “tope” de la que no se ha hablado.

Hasta ahora “tope” es el valor del nivel en el que se completa el recorrido de las ciudades intermedias. ¿Pero qué pasa si disminuimos el valor de “tope”? Entonces podemos recorrer el árbol de nodos de manera distinta: guardamos la distancia total del recorrido cuando consideramos un número menor de ciudades de las que se nos proporciona.

Esquemáticamente, se entendería de la siguiente manera:

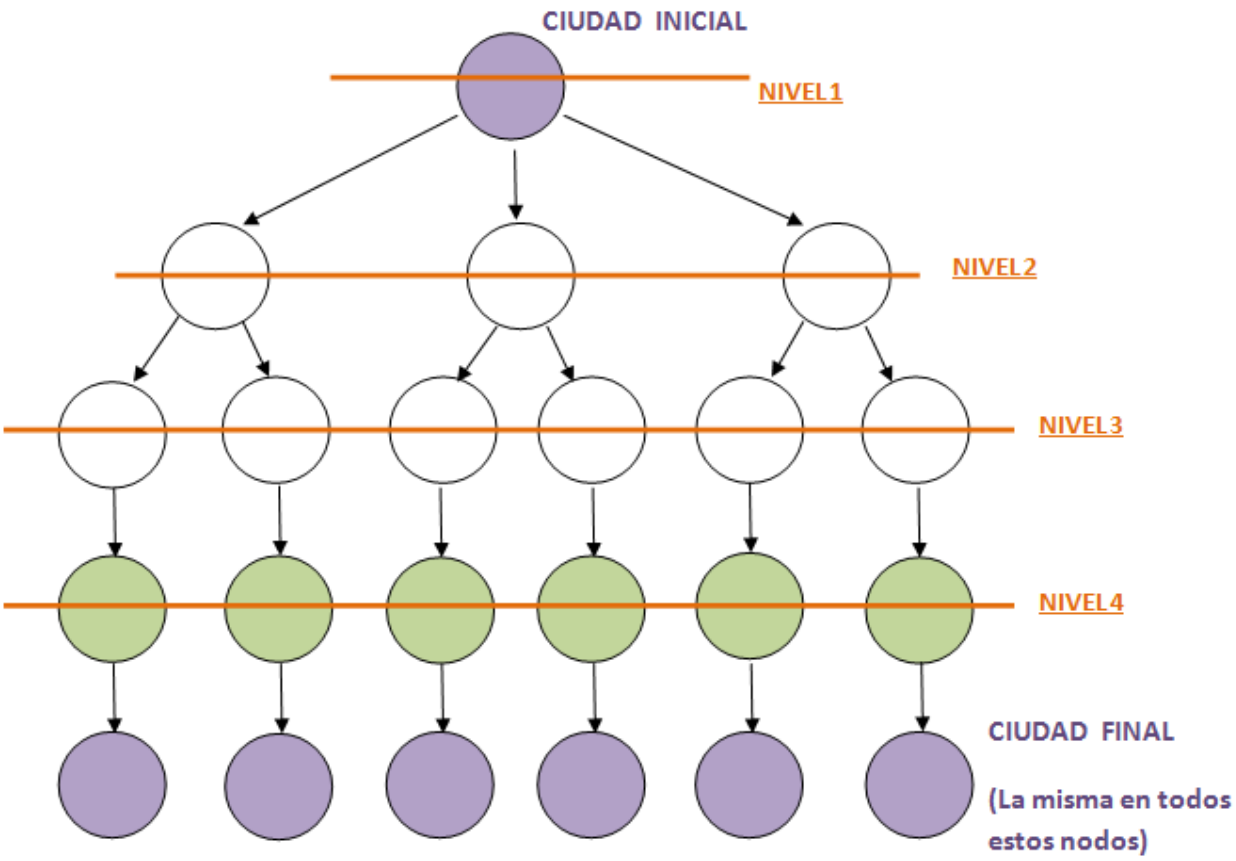

Figura 2-20. Árbol de nodos: “tope” toma el valor del nivel máximo.

Aquí mostramos el esquema de nodos que se ha empleado hasta ahora, añadiendo los nodos en el que se ha sacado la distancia total del recorrido (los morados).

Hay que decir que, aunque antes se haya indicado que los nodos del nivel 4 guardan la distancia total del recorrido, se guarda sin tener en cuenta la ciudad de destino, que se considera en el script “busca_recorrido” que hemos presentado anteriormente. Como se mostraba en dicho código, se unía la última ciudad intermedia por la que el viajero había pasado con la ciudad final. Por eso no le asignamos nivel a los nodos que guardan el recorrido completo, ya que con la distancia almacenada en los nodos del último nivel tenemos una buena referencia para sacar el recorrido entero de todas las ciudades.

En el esquema anterior se considera que el tope está en el nivel 4, es decir, se consideran todas las ciudades intermedias en la síntesis del recorrido. Pero, ¿y si decimos que el tope se sitúe en el nivel 3? El árbol de nodos tendría esta forma:

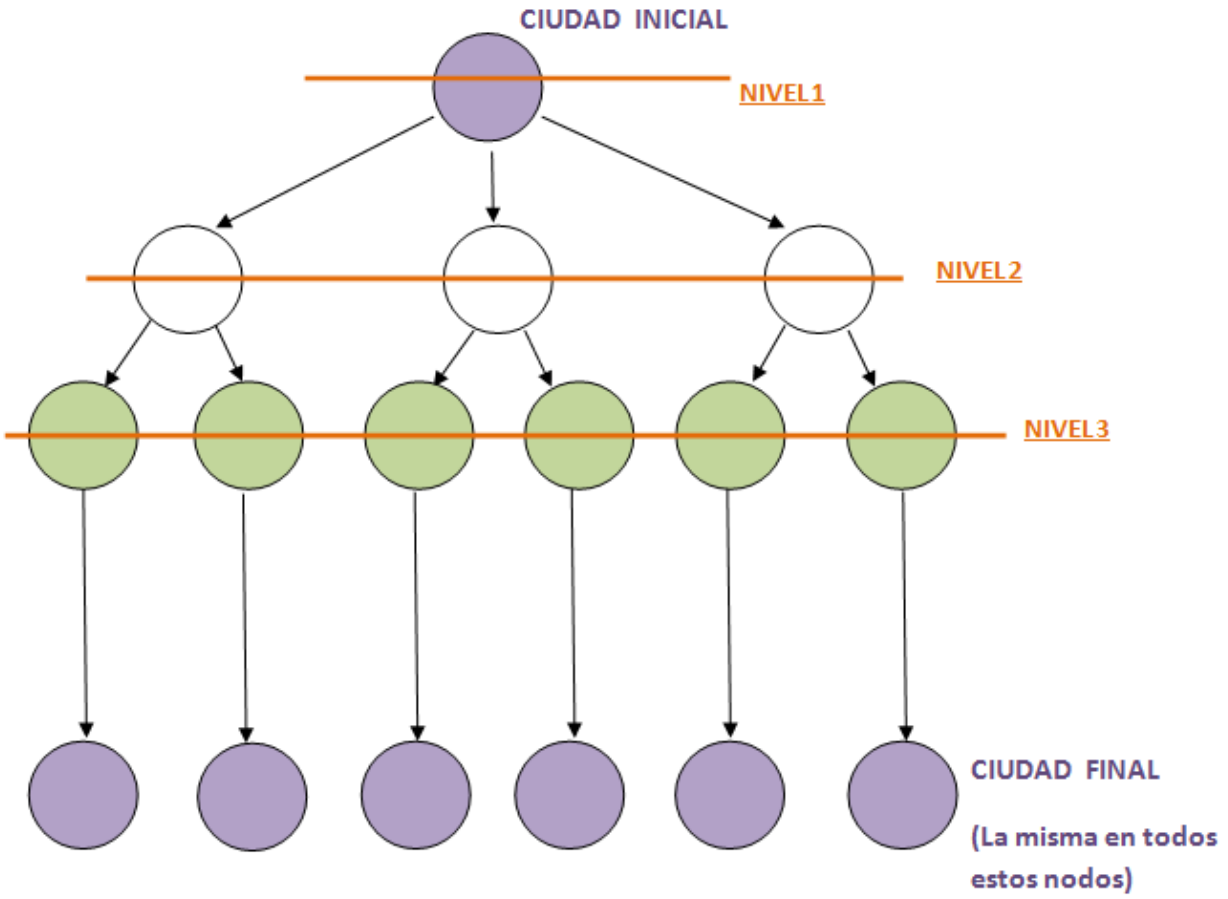

Figura 2-21. Árbol de nodos: "tope" toma el valor del nivel 3.

En este caso, tan solo pasaríamos por dos ciudades intermedias antes de llegar a la ciudad final ¿Y si ponemos el tope en el nivel 2?:

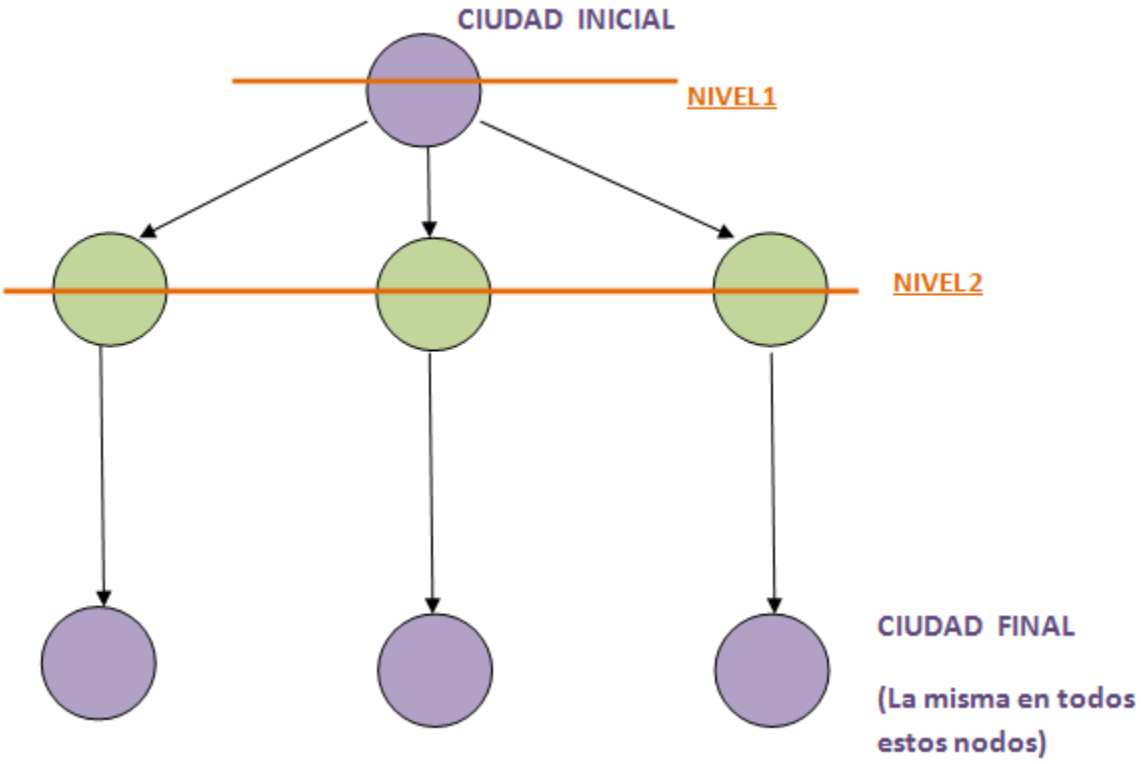

Figura 2-22. Árbol de nodos: "tope" toma el valor del nivel 2.

Entonces pasaríamos solo por una ciudad intermedia cuando vamos desde la ciudad inicial hasta la ciudad destino.

Siempre y cuando se considere que el tope está en el último nivel, estamos teniendo en cuenta el problema del TSP que explicamos antes, y el código expuesto anteriormente queda totalmente explicado.

Después de explicar cuál es el sentido de emplear "tope", se puede resolver el problema de pasar por "c" ciudades de entre "x" ciudades intermedias. El número de recorridos que se obtendrán ahora será:

$$\frac{x!}{(x-c)!}$$

El hecho de manejar "tope" en lugar de ceñirnos al caso de x=c es importante para resolver la primera variante del MTSP que se plantea.

2.2.3 CASO 1: Solo ciudades auxiliares.

Suponiendo que partimos de un número de ciudades a repartir (auxiliares) igual o mayor al número de destinos, en esta ocasión deberemos asignar a cada viajero un número de ciudades auxiliares por las que pasar, sin especificar las ciudades en concreto.

El hecho de que se especifique por cuántas ciudades quiere pasar cada viajero sin tener predefinidas las ciudades es precisamente lo que complica este problema. Hay que considerar bien qué ciudades se le asignarán a cada viajero de manera que los trayectos no sean demasiado largos, ya que las ciudades que elige cada viajero por las que pasar influirán a los viajeros que quedan por elegir.

Hasta ahora tan solo hemos considerado un viajero; hemos modificado el TSP original de manera que solo se pase por un número determinado de ciudades, de entre todas las que se proporcionan, para llegar de la manera más óptima posible al destino.

Una vez resuelto este problema para un viajero en particular, en esta variante del mTSP, iremos anulando en el vector índices (variable global que almacena todas las ciudades auxiliares por las que aún no se ha pasado) las componentes por la que ya hayan pasado otros viajeros, en nuestro caso el primero de ellos.

El siguiente viajero deberá resolver el mismo problema, esta vez con menos ciudades restantes entre las que escoger las "c" ciudades que minimicen el recorrido. El tercer viajero lo tendrá "más difícil"… y así sucesivamente hasta que el último viajero se enfrente a nuestro TSP original, ya que las ciudades sobrantes coincidirán tarde o temprano con las ciudades que le corresponde tomar en el reparto original.

Aunque bien es cierto que el reparto equitativo de ciudades entre viajeros no asegura que las distancias asociadas a cada trayecto sean pequeñas o equitativas, al menos nos da un comienzo sobre el que continuar mejorando los resultados que se obtengan.

Considerando que vamos resolviendo los sucesivos "TSP modificados" respetando el orden en el que vamos escogiendo los viajeros, en la iteración posterior a la actual asignaremos una ciudad más por la que pasar al viajero que ha recorrido menor distancia, mientras que al viajero que más ha tardado le asignaremos una ciudad menos. De esta manera, los viajeros recorrerán distancias cada vez más parecidas y con un valor máximo de conjunto cada vez menor, que es lo que estamos persiguiendo. En el código que se presentará más adelante, llamaremos a este proceso "mejora de resultados".

Hay que tener cuidado si procedemos de esta forma, ya que si la distancia máxima no dispone de puntos intermedios por los que pasar no se puede seguir con las iteraciones; no tiene puntos intermedios que ceder al camino mínimo. Tampoco seguiremos iterando cuando hayamos conseguido una máxima tolerancia admisible que respete un cierto margen.

Todo lo explicado quedará más claro una vez que mostremos el código

2.2.4 CASO 2: Solo ciudades intermedias asignadas.

Se trata del caso más sencillo que se tratará del MTSP; se considerarán "n" ciudades destino a las que se llegará desde una misma ciudad inicio, de manera que las ciudades intermedias tendrán asignado de antemano un destino en concreto.

Resulta tan sencillo como considerar por separado cada ruta (hay "n" rutas, una para cada destino) como un problema TSP. No nos encontraremos con ningún problema relacionado con la distribución de las ciudades intermedias entre los destinos, ya que cada ciudad está restringida a un viajero.

2.2.5 CASO 3: Ciudades asignadas y ciudades auxiliares.

En este apartado comienza a haber un problema: habrá ciudades por la que puede pasar cualquier viajero, ya que no está asignada de antemano a ningún destino en concreto. Llamaremos a estas ciudades "ciudades auxiliares".

Todo depende del número de ciudades auxiliares que tengamos para resolver el problema de una u otra manera. Llamemos "n" al número de destinos especificados en el MTSP, mientras que "d" será el número de ciudades auxiliares a repartir. Se darán los casos:

2.2.5.1 d>=n

Al principio, se repartirán aleatoriamente las ciudades auxiliares entre los distintos destinos, de manera que, por ejemplo, si d=7 y n=3, a dos destinos se les asignen dos de las ciudades auxiliares a cada uno mientras que al último destino se le asignen las otras 3 ciudades auxiliares. Se hará de manera equitativa.

Una vez que se repartan las ciudades auxiliares, veremos la longitud de cada recorrido en las componentes del vector "MINIMO". Si el recorrido más largo difiere demasiado del resto por una ciudad auxiliar, se intercambiará esta ciudad con alguna de otro recorrido, en concreto con una ciudad auxiliar del recorrido mínimo.

Cuando hayamos hecho este cambio en el reparto, volveremos a resolver el MTSP, veremos la distancia de cada recorrido, y así sucesivamente. Se harán sucesivans "mejoras".

Las iteraciones acabarán cuando las distancias de los recorridos sean parecidas entre sí y, a su vez, el máximo valor respete un cierto margen que definiremos en el código correspondiente.

2.2.5.2 d<n

También de manera aleatoria, se asignará una ciudad auxiliar a cada recorrido de manera ordenada, hasta que me quede sin ciudades que repartir entre los distintos recorridos.

De esta manera, habrá recorridos con ciudades auxiliares y otros sin ninguna.

Si la mayor distancia de entre todos los recorridos supera un margen predefinido, y además tiene una ciudad auxiliar, le asignaremos dicha ciudad a la menor distancia.

Este caso es bastante parecido al anterior, solo que en lugar de realizar un intercambio de ciudades entre dos recorridos distintos, simplemente se "transfiere" una ciudad auxiliar de un recorrido a otro para minimizar la distancia de la ruta más larga y aumentar el de la más larga.

Al igual que en el caso anterior, dejamos de repetir este procedimiento cuando el recorrido más largo no tenga ciudades auxiliares o cuando el mínimo respete un margen y, a su vez, todas las distancias de recorrido sean más o menos parecidas.

2.2.6 Scripts principales

A continuación se mostrará el código de dos scripts; el primero de ellos (pide_pantalla_MTSP) será el encargado de pedir por pantalla las ciudades que se considerarán en el problema y representarlas gráficamente, mientras el segundo script (MTSP_pantalla_CIUDADES) se encargará de resolver el problema del MTSP en función de los datos de entrada que hemos introducido.

El código de "pide_pantalla_MTSP" es el siguiente:

```
  close all; clear all; clc;

  %% Abro una gráfica en la se representará el primer punto que me
piden por pantalla, "first" (en coordenadas cartesianas)

  puntos=[]; figure(1); grid; hold on;
  first=input('Primer punto de la trayectoria : ');
      axis([min(first(:,1))-3    max(first(:,1))+3    min(first(:,2))-3
max(first(:,2))+3]);
      hold on;
      plot(first(1),first(2),'o')


  %% Me piden por pantalla los puntos finales de los recorridos,
  % que se almacenarán en el vector "salesman" hasta que deje de pedir
  % valores (introduzco -1 por pantalla)

   ending=0;   c_salesman=1; cont=1; conjunto=first;
  while (ending==0)
       res=input('Punto final de la trayectoria (-1 para siguiente
mensaje): ');
      if (res==-1) & (length(res)==1)
      ending=1;
      else
       salesman(c_salesman,:)=res;
```

```
        conjunto=[conjunto;salesman];
        axis([min(conjunto(:,1))-3                    max(conjunto(:,1))+3
min(conjunto(:,2))-3 max(conjunto(:,2))+3]);
      hold on;
      plot(res(1),res(2),'o')
      end



  %% Para cada destino, introduzco las ciudades restringidas asociadas
hasta cancelar con -1. Se guardarán en la estructura "puntos", que
almacena las coordenadas de la ciudad y el destino al que está
asociado.

  salir=0;
   while (salir==0) & (ending==0)
  res=input('Punto intermedios en la trayectoria (-1 para siguiente
mensaje): ');
      if (res==-1) & (length(res)==1)
          c_salesman=c_salesman+1;
           salir=1;
      else
           puntos(cont).componente=res;
           puntos(cont).destino=c_salesman;

           conjunto=[conjunto;puntos(cont).componente];
           cont=cont+1;

           axis([min(conjunto(:,1))-3                    max(conjunto(:,1))+3
min(conjunto(:,2))-3 max(conjunto(:,2))+3]);
      hold on;
      plot(res(1),res(2),'o')
      end

   end

  end

  %% Introduzco las ciudades auxiliares que tendrán que repartirse los
recorridos. Se guardarán en el vector "sobrantes".

  salir=0; sobrantes=[]; cont=1;
   while (salir==0)
  res=input('Puntos auxiliares (-1 para siguiente mensaje): ');
      if (res==-1) & (length(res)==1)
           salir=1;
      else
           sobrantes(cont).componente=res;

           conjunto=[conjunto;sobrantes(cont).componente];
           cont=cont+1;

           axis([min(conjunto(:,1))-3                    max(conjunto(:,1))+3
min(conjunto(:,2))-3 max(conjunto(:,2))+3]);
      hold on;
      plot(res(1),res(2),'o')
```

```
      end

   end

   %% NUMERO DE ESTIMACIONES QUE SE EMPLEARÁN PARA RESOLVER CADA TSP

  limit=input('¿Cuántas estimaciones de recorrido?(proporcionales al
costo computacional) : ');

   MTSP_pantalla_CIUDADES_MEJORA; % EJECUTO EL SCRIPT QUE SE EXPLICARÁ
A CONTINUACIÓN

  barrera=0;
  while(barrera==0)
  limit=input('¿Varío las estimaciones? Afectará al recorrido
recomendado (cancelar con -1): ');
  if (limit==-1)
      barrera=1;
  else
      close(2); close(3); % sobreescribire las trayectorias y la
gráfica de los recorridos.
      MTSP_pantalla_CIUDADES_MEJORA; % EJECUTO EL SCRIPT QUE SE
EXPLICARÁ A CONTINUACIÓN
  end
  end
```

El código de “MTSP_pantalla_ciudades” es el siguiente:

```
  destinos=length(salesman(:,1));
  n_sobrantes=length(sobrantes);
  v_ciudades=[];

  %% REPARTIMOS LAS CIUDADES AUXILIARES DEL ESPACIO

  % Inicialmente, las componentes de "p" serán nulas hasta que no se
  % considere que hay ciudades auxiliares.
  p=zeros(1,destinos); c_sobrantes=n_sobrantes;
  if (n_sobrantes>=destinos) % Si hay más ciudades auxiliares que
destinos:

  % vuelco en las componentes de "p" el número de ciudades auxiliares
que le correspondería a cada destino, al principio.
  for i=destinos:-1:1
  p(i)=floor(c_sobrantes/i);
  c_sobrantes=c_sobrantes-p(i);
  end

  else % en otro caso, le voy asignando una ciudad auxiliar a cada
destino hasta que no queden más

   i=1;
```

```
    while (c_sobrantes>0)   % Si no considero ciudades auxiliares, no
considero el "while"
  p(i)=1;
  c_sobrantes=c_sobrantes-1; i=i+1;
    end
  end


  %% DEFINO UN CONTADOR PARA LAS CIUDADES AUXILIARES

  c_sobrantes=1;

  %% EMPIEZA CODIGO PRINCIPAL


   C=1; %CONTADOR PARA ALMACENAR LA DISTANCIA DE CADA SALESMAN.
        % Se incrementa en "busca_recorrido" una vez que defino una
        % componente del vector "MINIMO" en ese mismo script.

   for ctd=1:destinos % para cada destino/viajero:
       %% ALMACENO LAS CIUDADES RESTRINGIDAS POR LAS QUE DEBO PASAR
PARA LLEGAR A MI DESTINO
       % Almaceno en el vector "dots_"
       dots_=[]; dots=[];
       for td=1:length(puntos)
           % Almaceno en "dots_" los ciudades restringidas que les
           % corresponda el destino que estamos considerando, "ctd".
           if (ctd==puntos(td).destino)
               dots_=[dots_;puntos(td).componente];
           end
       end

       %% ALMACENO LAS CIUDADES AUXILIARES REPARTIÉNDOLAS POR LAS RUTAS
       % Almaceno en el vector "leftovers"
       leftovers=[];

       % Le asigno a cada destino, "ctd", tantas ciudades auxiliares
como haya dicho anteriormente al redefinir "p".
       for j=1:p(ctd)
           sobrantes(c_sobrantes).destino=ctd;
           leftovers=[leftovers;sobrantes(c_sobrantes).componente];
           c_sobrantes=c_sobrantes+1;
       end

       %% VUELCO EN "dots" TODOS LOS PUNTOS POR LOS QUE PASA UN VIAJERO
EN CONCRETO.
  dots=[first;dots_;leftovers;salesman(ctd,:)]; % simplemente hay que
especificar el primero y el ultimo

  %% DEFINO LOS ÍNDICES QUE DISTINGUEN A LAS CIUDADES ENTRE SÍ (1-
>inicio)

  indice=1;
  fin=length(dots(:,1));
  indices=indice:fin;
  indices(1)=0; indices(length(indices))=0;
```

```
  %% "ciudades" contiene el número de ciudades intermedias

  ciudades=length(refresca_auxiliar_f(indices));
  tope=(length(dots)-2);  % numero de puntos intermedios por los que
quiero pasar. Se ha explicado anteriormente esta variable.

  v_ciudades(ctd)=tope;

  % se ejecuta "principal", que se ha explicado anteriormente.
  principal

   end

   %% ACABA CÓDIGO PRINCIPAL
   contador_mejora=1;
   intentos_mejora(contador_mejora,:)=MINIMO % Muestro el coste de cada
una de los recorridos.
  contador_mejora=1+contador_mejora;

   %% MEJORAR LOS RECORRIDOS O NO. SE VOLVERÁ A RESOLVER EL MTSP PARA
MEJORAR RESULTADOS

   % Mientras no se excedan un máximo de iteraciones para mejorar el
   % resultado (redistribuyendo las ciudades auxiliares) y mientras se
   % el recorrido máximo no sea lo suficientemente pequeño:

   while contador_mejora<=tope_mejora && CANCELA_WHILE==0
   % Guardo los índices del recorrido más largo (ma) y del más corto
(mi)
   for i=length(MINIMO):-1:1
       if (max(MINIMO)==MINIMO(i))
           ma=i; % Índice de la componente máxima de "MINIMO"
       elseif (min(MINIMO)==MINIMO(i))
          mi=i; % Índice de la componente mínima de "MINIMO"
       end
   end

   %% A CONTINUACIÓN, LE ASIGNO UNA CIUDAD AUXILIAR MÁS AL DESTINO CON
EL RECORRIDO
   % MÁS CORTO, MIENTRAS QUE LE ASIGNO UNA CIUDAD MENOS A LA CIUDAD CON
EL RECORRIDO MÁS LARGO.

   flag=0; flag2=0; cambiado=-1;
   % Si solo manejo ciudades auxiliares, ni siquiera entro el el
próximo "for", ya que el vector "sobrantes" está vacío.
  for j=1:length(sobrantes)
      if (ma==sobrantes(j).destino) && (flag==0)
          flag=1;
          cambiado=j;
           sobrantes(j).destino=mi;
      elseif (mi==sobrantes(j).destino) &&(flag2==0) && (j~=cambiado)
          flag2=1;
          sobrantes(j).destino=ma;
      end
  end
```

```
  % Cierro las gráficas antiguas porque voy a resolver de nuevo el
problema.
  close (2); close (3);


  %% CODIGO PRINCIPAL, lo repito
  % excepto por un detalle que se comenta más adelante

   C=1;
   for ctd=1:destinos

       dots_=[]; dots=[];
       for td=1:length(puntos)
           if (ctd==puntos(td).destino)
               dots_=[dots_;puntos(td).componente];
           end
       end

       % DISTINTO A LA PRIMERA VEZ: AHORA SOLO TENGO QUE REASIGNAR LAS
       % CIUDADES AUXILIARES A LOS DESTINOS EN LUGAR DE RECORRER EL
VECTOR "p"
       leftovers=[];

       for j=1:length(sobrantes)
           if (sobrantes(j).destino==ctd )
           leftovers=[leftovers;sobrantes(j).componente];
           end
       end

  dots=[first;dots_;leftovers;salesman(ctd,:)]; % simplemente hay que
especificar el primero y el ultimo

  indice=1;
  fin=length(dots(:,1));
  indices=indice:fin;
  indices(1)=0; indices(length(indices))=0;

  ciudades=length(refresca_auxiliar_f(indices));
  tope=(length(dots)-2);

  v_ciudades(ctd)=tope;

  principal

   end

  intentos_mejora(contador_mejora,:)=MINIMO % Muestro el coste de cada
una de los recorridos.
   contador_mejora=1+contador_mejora;
   end
```

Además se define el script "MTSP_pantalla_ciudades_mejora" en el que se emplea el script anterior: se trata de un script que baraja la mejor opción una vez que se lleva a cabo mejoras entre soluciones distintas del mTSP. Estas soluciones del mTSP variarán entre sí en la distribución de ciudades auxiliares que se ha llevado a cabo entre los destinos, como se ha indicado anteriormente.

Veremos en cuál de las mejoras el camino más largo es mínimo en comparación con el resto de las mejoras o, incluso, con la solución inicial a la que se llegó sin realizar mejoras.

El código de dicho script, que se emplea a su vez en "pide_pantalla_MTSP", es el siguiente:

```
  %% ESTE SCRIPT SERÁ MÁS ÚTIL CUANTOS MÁS DESTINOS HAYA EN EL MTSP

  tope_mejora=5;  % Llevaré a cabo un número de mejoras fijo en un
principio.
  CANCELA_WHILE=0;

  MTSP_pantalla_CIUDADES;

  %% DE TODO EL CONJUNTO DE RECORRIDOS QUE SE HAN LLEVADO A CABO CON
CADA MEJORA, BUSCO EL QUE TENGA EL RECORRIDO MÁXIMO MÁS CORTO
  for i=1:length(intentos_mejora(:,1))
      recorrido_minimo(i)=max(intentos_mejora(i,:));
  end

  %%
  % GUARDO EN ÍNDICE DE LA MEJORA QUE MEJOR RESULTADO ME HA DADO PARA
  % BUSCARLA DE NUEVO Y REPRESENTARLA GRÁFICAMENTE.
  % Para ello, vuelco en "tope_mejora" dicho índice para para llegar a
la mejora que mejor resultado me ha dado, ejecutando de nuevo
  % "MTSP_pantalla_CIUDADES"
  flag=0;
  for i=1:length(recorrido_minimo)
      if (min(recorrido_minimo)==recorrido_minimo(i)) && flag==0
          tope_mejora=i;   flag=1;
      end
  end

  %%
  % EN ESTE CASO NO HABRA QUE ENTRAR EN EL BUCLE WHILE DE LAS MEJORAS,
PORQUE LAS MEJORAS DAN PEOR RESULTADO QUE LA PRIMERAQ SOLUCIÓN
  if (tope_mejora==1)
      CANCELA_WHILE=1;
  end


  clear intentos_mejora; close (2); close (3);

  MTSP_pantalla_CIUDADES;
```

2.3 Funciones empleadas en los scripts y las funciones presentados hasta ahora.

A lo largo del código que se ha mostrado en la memoria, se han empleado funciones propias que se han creado exclusivamente para facilitar distintas tareas que se han tenido que llevar a cabo, como ordenar un vector, ordenar los nodos hijos en función de la distancia acumulada entre ciudad y ciudad, etc.

Se irán mostrando tanto el propósito de cada una de las funciones, así como el código que las define:

2.3.1 "ordena_nodo" ("devuelve_hijo").

Tomando el vector "camino" de un nodo, se pretende reordenar el vector "punto_sig" de ese mismo nodo de manera que, para un mismo índice, a cada distancia que se sume al recorrido total con "camino" le corresponda el índice de la ciudad futura correspondiente.

```
  function [s_]=ordena_nodo(s)

  [auxiliar]=ordena_vector(s.camino);
  [s.punto_sig]=reordena_vector2(s.camino,auxiliar,s.punto_sig);
  [s.camino_activo]=reordena_vector2(s.camino,auxiliar,s.camino_activo)
;
    s.camino=auxiliar;
    s_=s;

   end
```

2.3.2 "ordena_vector".

Ordena de menor a mayor las componentes de un vector.

```
  function [v_]=ordena_vector(v)
      v_=[];
  lon=length(v);

   for j=1:lon

      flag=0;
      for i=1:length(v)
         if min(v)==v(i) && flag==0
             v_=[v_,v(i)];
             flag=1;
             v(i)=0;
         end
      end
          v_aux=[];

      for i=1:length(v)
         if v(i)==0
         else
             v_aux=[v_aux,v(i)];
         end
      end
      v=v_aux;


   end

   end
```

2.3.3 "reordena_vector2".

Esta función descubre el cambio que sufre el vector "v_original" para que sus componentes se redistribuyan de manera que se obtenga "v_ordenado". Se pretende aplicar el mismo cambio a "v_a_cambiar" para que se obtenga "v_cambiado".

```
   function
[v_cambiado]=reordena_vector2(v_original,v_ordenado,v_a_cambiar)
  v_cambiado=[]; flag=0;

 for i=1:length(v_original)

  for j=1:length(v_ordenado)

     if v_original(j)==v_ordenado(i) && flag==0

         v_cambiado(i)=v_a_cambiar(j);
         v_original(j)=-1; flag=1;
     end

  end
 flag=0;

 end
 end
```

2.3.4 "devuelve_activo" ("devuelve_hijo").

A menos que todas las componentes de un vector sean nulas, devuelve un 1. Sirve para decidir si un nodo sigue vivo al obtener un hijo.

```
 function [res]=devuelve_activo(vector)

 flag=0;
 for i=1:length(vector)
     if flag==0 && vector(i)==1
        flag=1; res=1;
     end
 if i==length(vector) && flag==0
     res=0;
 end
 end
  end
```

2.3.5 "refresca_auxiliar_f" ("devuelve_hijo").

A medida que se van visitando ciudades intermedias, hay que eliminar del vector "indices" las ciudades por las

que se haya pasado. Esta función se encarga de quitar del vector las ciudades visitadas.

```
function [salida]=refresca_auxiliar_f(entrada)

 indices_aux=[]; indices=entrada;
for x=1:length(indices)
    if indices(x)==0
    else
    indices_aux=[indices_aux,indices(x)];
    end
end
salida=indices_aux;
end
```

2.3.6 "es_ultimo" ("repeticion").

Gracias a esta función sabemos si el nodo que tratamos está o no en el último nivel. Si lo está y sólo falta viajar a ciudad destino, entonces el vector "punto_sig" de la estructura estará vacío y se devolverá la distancia total de todo el recorrido. En otro caso, se devuelve un 0 para indicar que el nodo no está en el último nivel.

```
function [distancia]=es_ultimo(nodo,dots,fin)
 if isempty(nodo.punto_sig)==1
 c=[(nodo.punto_act) (fin)]; % es el vector "conecta el punto 1 con
el 'recorrido'"
distancia=norm(dots(c(1),:)-dots(c(2),:));
 else
 distancia=0;
 end

end
```

2.3.7 "componente_rep" ("repeticion").

Devuelve un 1 en el caso de que "numero" pertenezca al vector "v". Devuelve un 0 en el caso contrario.

```
function [res]=componente_rep(v,numero)
res=0;
for i=1:length(v)
    if numero==v(i)
        res=1;
    end
end

end
```

3 EXPERIMENTACIÓN Y COMPARATIVA

3.1 Introducción

3.1.1 Experimentación

Mostraremos algunos ejemplos que han sido resultado de ejecutar el script principal de nuestro programa, "pide_pantalla_MTSP". Enseñaremos tanto las ciudades que se han introducido por pantalla junto con tres gráficas distintas: una en la que se muestran las coordenadas de las ciudades dibujadas en el espacio, otra en la que se dibuja el recorrido sobre esos tres puntos y, por último, el valor que toma cada recorrido para cada iteración considerada. Además de los distintos tipos de ciudades (inicial, final, auxiliar y restringida), también se pedirá un número de estimaciones que de terminará lo buena o mala que es la solución representada. A mayor número de iteraciones, más tiempo se tarda el llegar a la solución.

3.1.1.1 TSP (10 iteraciones)

3.1.1.1.1 Ciudades introducidas

```
Primer punto de la trayectoria : [0 0]
Punto final de la trayectoria (-1 para siguiente mensaje): [20 20]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [6 3]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [3 15]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [2 -3]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [15 0]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [14 6]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [17 13]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [10 12]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [12 4]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [19 6]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [7 19]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): -1
Puntos auxiliares (-1 para siguiente mensaje): -1
```

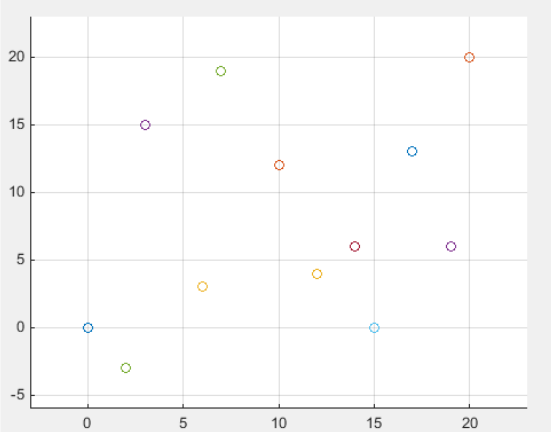

Figura 3-1. Ciudades introducidas en el TSP con 10 iteraciones.

3.1.1.1.2 Recorrido

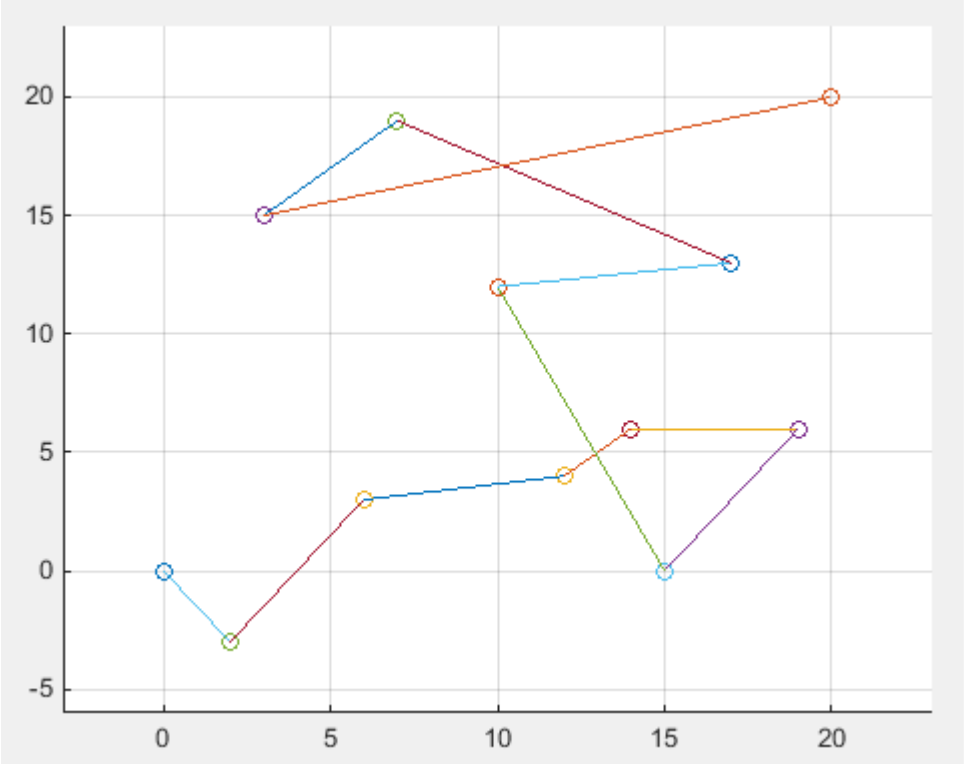

Figura 3-2. Recorrido en el TSP con 10 iteraciones.

El recorrido comienza en [0 0] y finaliza en [20 20].

3.1.1.1.3 Distancia en cada iteración

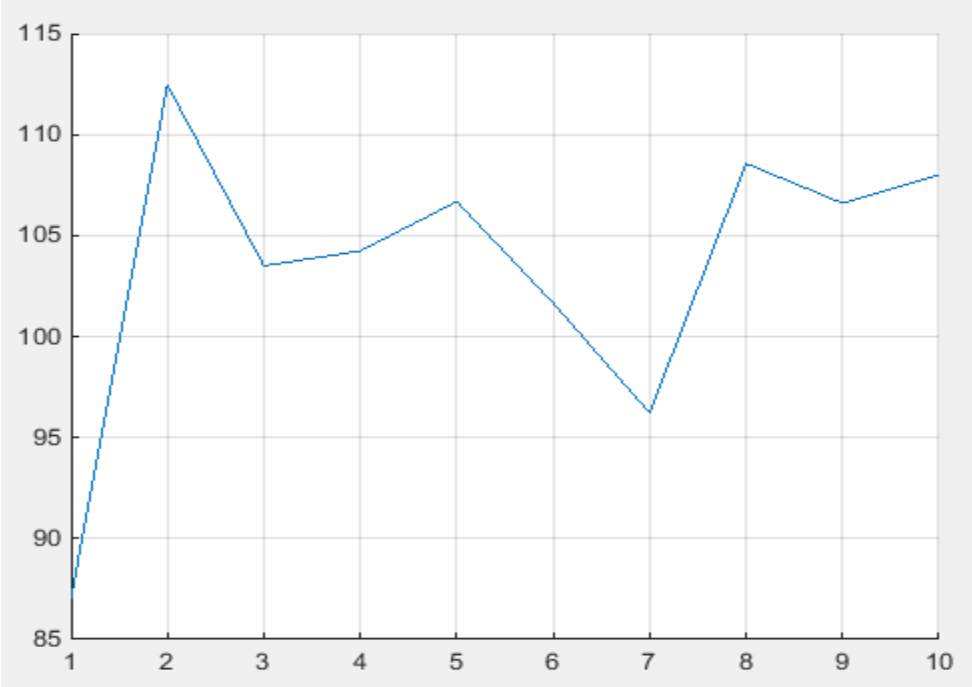

Figura 3-3. Distancias en el TSP con 10 iteraciones.

El mejor resultado se obtiene en la primera iteración, ya que es cuando se da el mínimo en la gráfica.

3.1.1.2 TSP (100 iteraciones)

Para las mismas ciudades introducidas, veremos cómo la solución en este caso es mejor que para 10 iteraciones.

3.1.1.2.1 Recorrido

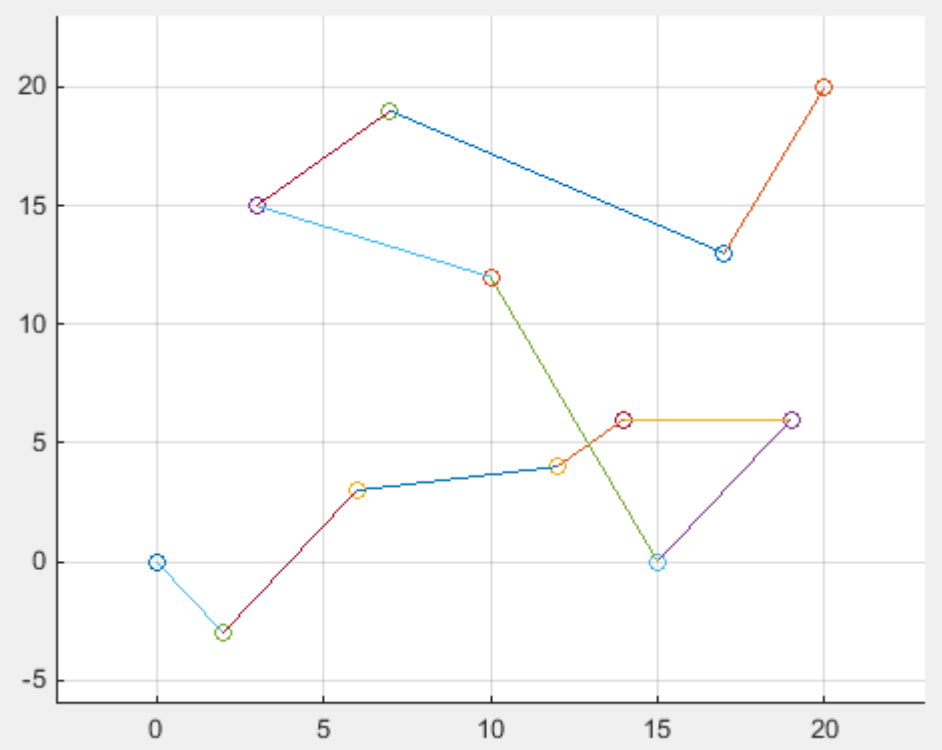

Figura 3-4. Recorrido en el TSP con 100 iteraciones.

La mejora se produce en el tramo que va desde [10 13] hasta [20 20], como se apreciará mejor en la gráfica siguiente

3.1.1.2.2 Distancia en cada iteración

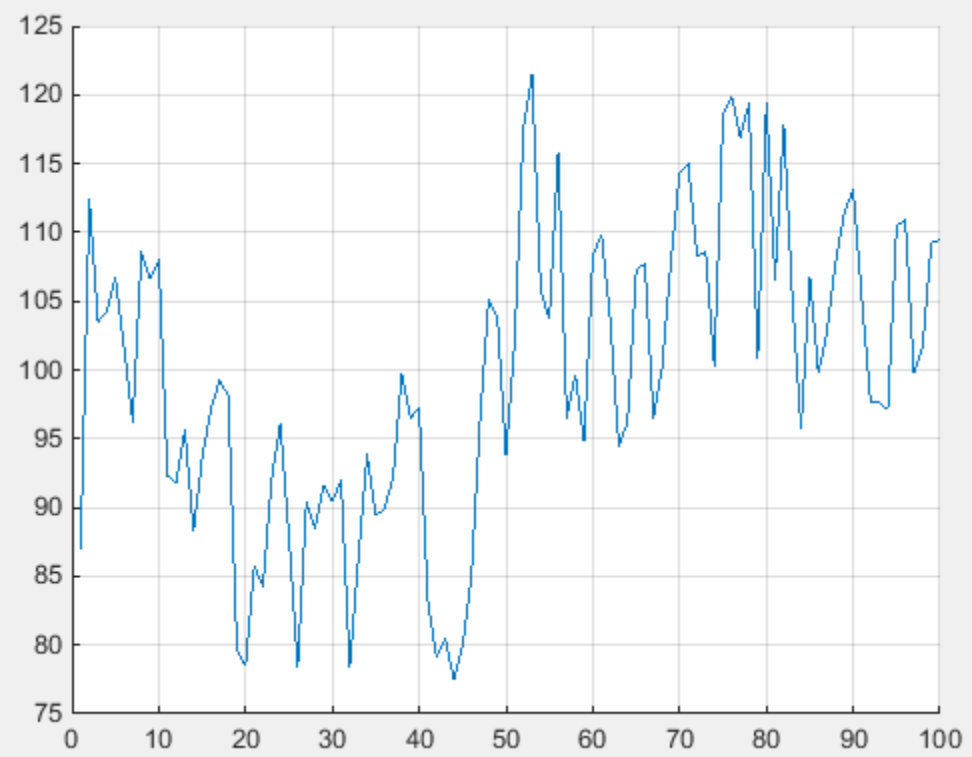

Figura 3-5. Distancias en el TSP con 100 iteraciones.

Ahora, la mejor solución se obtiene aproximadamente en la iteración número 43, que es cuando se da el mínimo.

3.1.1.3 MTSP: Solo ciudades restringidas

3.1.1.3.1 Ciudades introducidas

```
Primer punto de la trayectoria : [0 0]
Punto final de la trayectoria (-1 para siguiente mensaje): [10 10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [2 4]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [4 2]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [6 7]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [2 -1]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [8 5]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [9 13]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [2 13]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [-10 20]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-5 3]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-4 1]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-2 5]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-8 7]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-5 13]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-1 17]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [2 -15]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-10 -10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [10 -7]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [5 -6]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-3 -4]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): -1
Puntos auxiliares (-1 para siguiente mensaje): -1
```

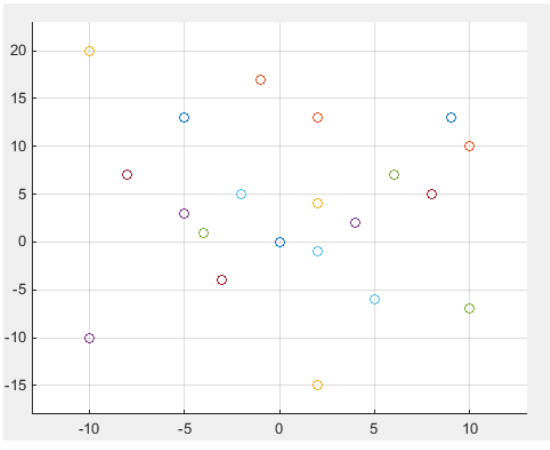

Figura 3-6. Ciudades introducidas en el mTSP con ciudades restringidas.

3.1.1.3.2 Recorrido

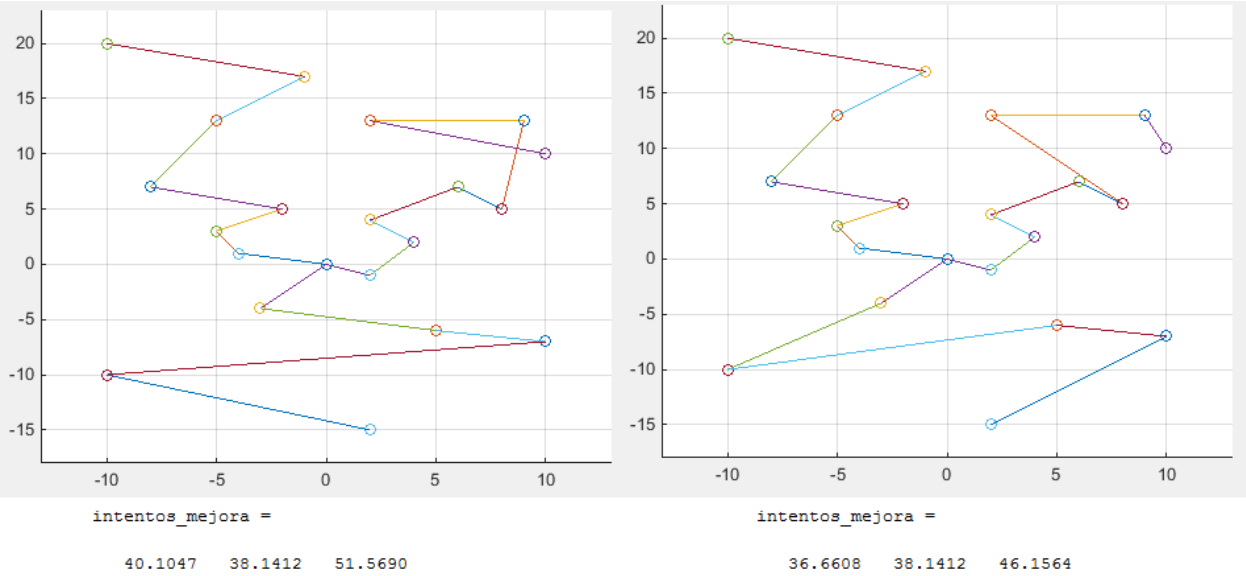

Figura 3-7. Recorridos en el mTSP con ciudades restringidas.

Ya que solo consideramos ciudades restringidas, no se llevan a cabo mejoras del recorrido porque carecemos de ciudades auxiliares. Sin embargo, sí podemos mejorar el resultado aumentando el número de iteraciones consideradas.

En la figura superior, se puede ver a la izquierda el recorrido que se obtiene para la primera iteración y la longitud de cada tramo en el vector "intentos_mejora". El máximo valor del vector es 51.56, por lo que esa es la distancia del recorrido más largo.

Sin embargo, si consideramos 50 iteraciones el máximo de dicho vector ahora vale 46.15, por lo que se ha logrado disminuir la máxima distancia de entre todas las consideradas en los distintos recorridos. Esto mismo se puede comprobar comparando las rutas de la izquierda y la derecha, siendo las de la derecha mejores que las de la izquierda.

3.1.1.3.3 Distancia en cada iteración

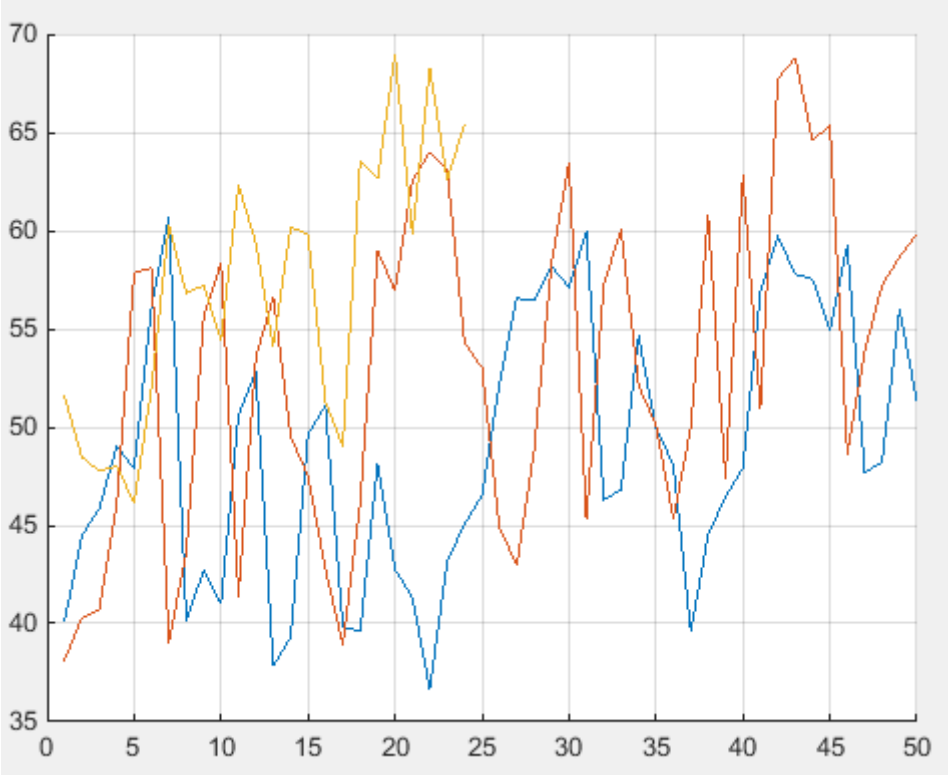

Figura 3-8. Distancias en el mTSP con ciudades restringidas.

Cada color se corresponde con una de las tres rutas consideradas en el MTSP de 50 iteraciones, con los datos considerados.

3.1.1.4 MTSP: Ciudades restringidas y auxiliares

3.1.1.4.1 Ciudades introducidas

```
Primer punto de la trayectoria : [0 0]
Punto final de la trayectoria (-1 para siguiente mensaje): [10 19]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [10 10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [5 6]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [8 13]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [3 4]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [-10 -18]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-2 -3]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [-20 10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [-15 6]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [5 -22]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): [0 -10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): -1
Puntos auxiliares (-1 para siguiente mensaje): [-5 -20]
Puntos auxiliares (-1 para siguiente mensaje): [0 20]
Puntos auxiliares (-1 para siguiente mensaje): [-7 15]
Puntos auxiliares (-1 para siguiente mensaje): [-15 -10]
Puntos auxiliares (-1 para siguiente mensaje): [-7 0]
Puntos auxiliares (-1 para siguiente mensaje): -1
```

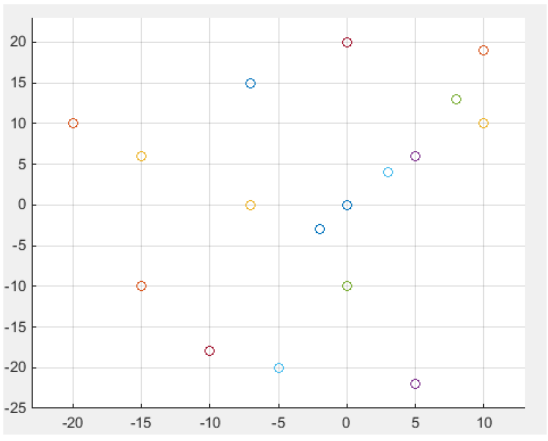

Figura 3-9. Ciudades introducidas en el mTSP con ciudades restringidas y auxiliares.

3.1.1.4.2 Recorrido

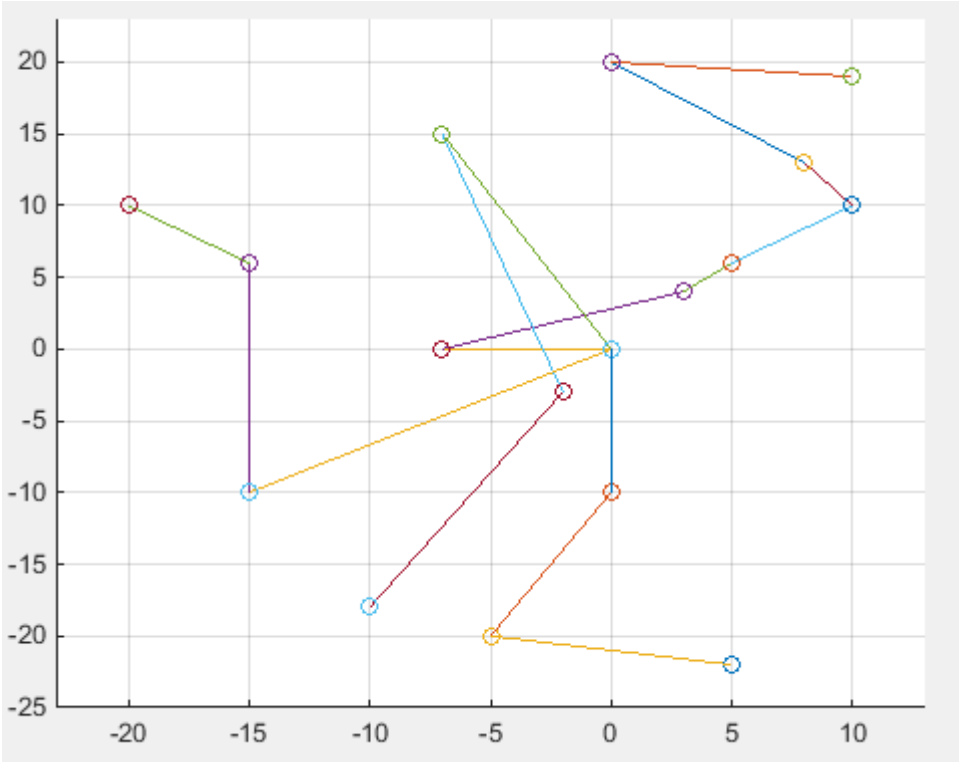

Figura 3-10. Recorridos en el mTSP con ciudades restringidas y auxiliares.

Se puede apreciar que salen 4 caminos desde la ciudad inicial, [0 0], hasta llegar cada una a su destino correspondiente.

3.1.1.4.3 Salida por pantalla

```
MINIMO =

   79.4309   52.2345   40.4309   32.2066

intentos_mejora =

   51.2875   52.2345   40.4309   31.3784

intentos_mejora =

   51.2875   52.2345   40.4309   31.3784
   51.2875   26.2534   40.4309   55.5145

intentos_mejora =

   51.2875   52.2345   40.4309   31.3784
   51.2875   26.2534   40.4309   55.5145
   51.2875   52.2345   40.4309   31.3784

intentos_mejora =

   51.2875   52.2345   40.4309   31.3784
   51.2875   26.2534   40.4309   55.5145
   51.2875   52.2345   40.4309   31.3784

intentos_mejora =

   51.2875   52.2345   40.4309   31.3784
   51.2875   26.2534   40.4309   55.5145
   51.2875   52.2345   40.4309   31.3784
   51.2875   26.2534   40.4309   55.5145
   51.2875   52.2345   40.4309   31.3784

MINIMO =

   79.4309   52.2345   40.4309   32.2066

intentos_mejora =

   51.2875   52.2345   40.4309   31.3784
```

Figura 3-11. Resultados en el mTSP con ciudades restringidas y auxiliares.

Los resultados de las dos primeras mejoras se repiten cíclicamente porque se intercambian ciudades las dos mismas rutas, la más larga y la más corta. La mejor mejora es la primera, la que tiene como máximo 52.23.

A continuación se mostrará el caso en el que sólo existen ciudades auxiliares, en el cual las mejoras pueden llegar a resultar más interesantes.

3.1.1.5 MTSP: Solo ciudades auxiliares

3.1.1.5.1 Ciudades introducidas

```
Primer punto de la trayectoria : [0 0]
Punto final de la trayectoria (-1 para siguiente mensaje): [10 10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [-9 10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [-13 -11]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [10 -10]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): [2 -15]
Punto intermedios en la trayectoria (-1 para siguiente mensaje): -1
Punto final de la trayectoria (-1 para siguiente mensaje): -1
Puntos auxiliares (-1 para siguiente mensaje): [1 3]
Puntos auxiliares (-1 para siguiente mensaje): [4 5]
Puntos auxiliares (-1 para siguiente mensaje): [-5 -5]
Puntos auxiliares (-1 para siguiente mensaje): [-7 5]
Puntos auxiliares (-1 para siguiente mensaje): [-4 -1]
Puntos auxiliares (-1 para siguiente mensaje): [2 -6]
Puntos auxiliares (-1 para siguiente mensaje): [6 -6]
Puntos auxiliares (-1 para siguiente mensaje): [-7 -10]
Puntos auxiliares (-1 para siguiente mensaje): [-4 -12]
Puntos auxiliares (-1 para siguiente mensaje): -1
```

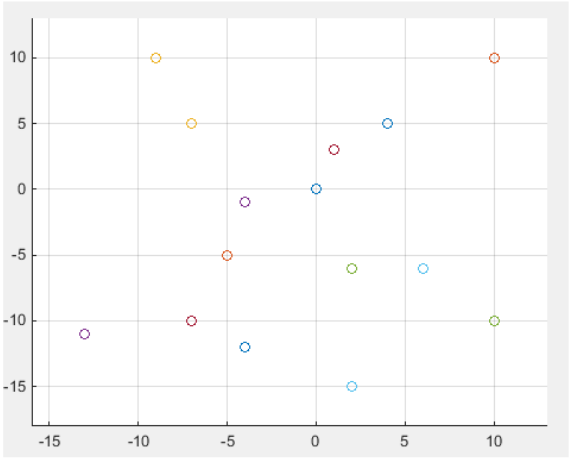

Figura 3-12. Ciudades introducidas en el mTSP con ciudades auxiliares.

3.1.1.5.2 Recorrido

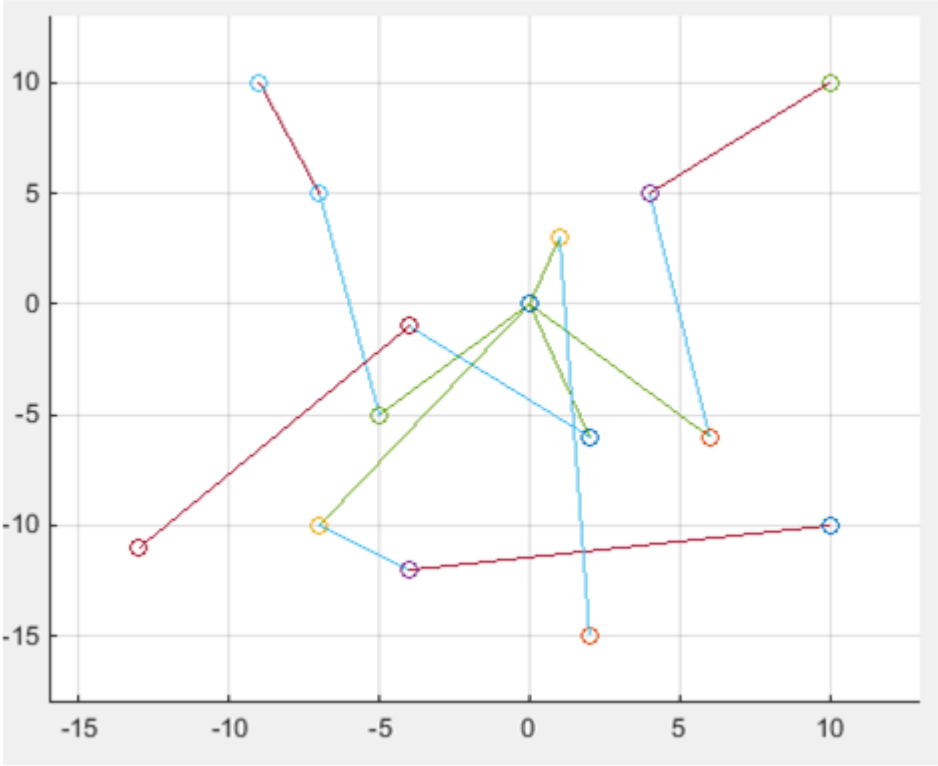

Figura 3-13. Recorridos en el mTSP con ciudades auxiliares.

Resultado final una vez llevadas a cabo las mejoras: salen cinco recorridos desde [0 0] y se reparten todos los puntos auxiliares como explicamos anteriormente. Ya que cuesta distinguirlos un poco, a continuación se muestra la misma figura con cada ruta de un color:

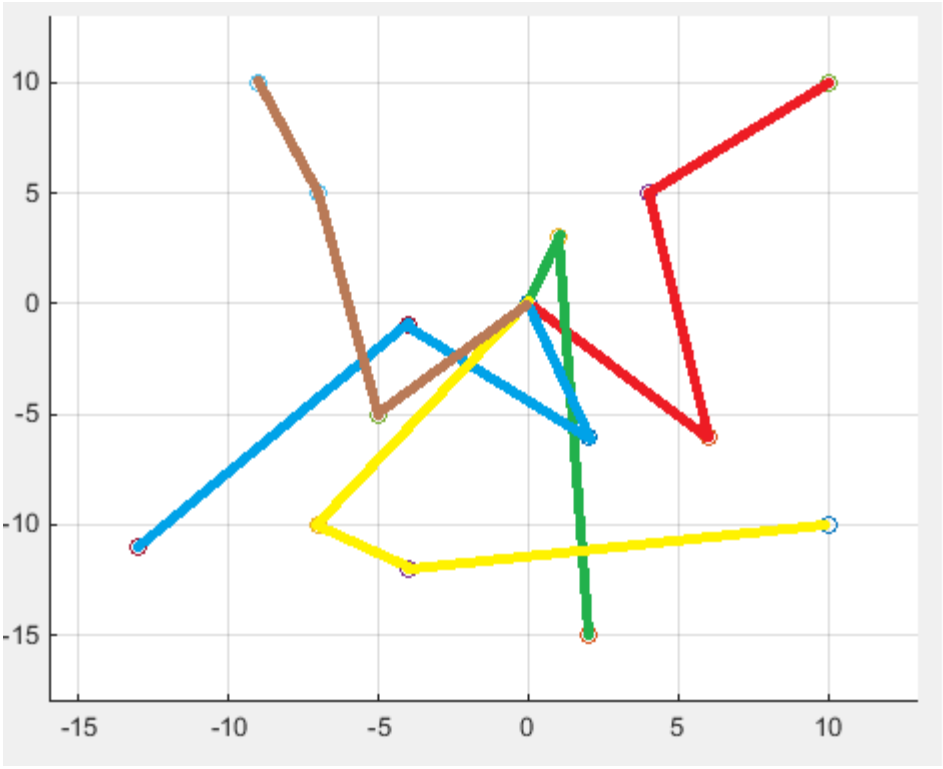

Figura 3-14. Recorridos en el mTSP con ciudades auxiliares, resaltado.

Ya que como máximo se pueden dar dos iteraciones en cada una de las rutas (porque como máximo hay 2 ciudades intermedias por ruta), la siguiente gráfica mostrará solo dos iteraciones para cada ruta, excepto para la de color verde porque solo posee una ciudad intermedia.

3.1.1.5.3 Distancia en cada iteración

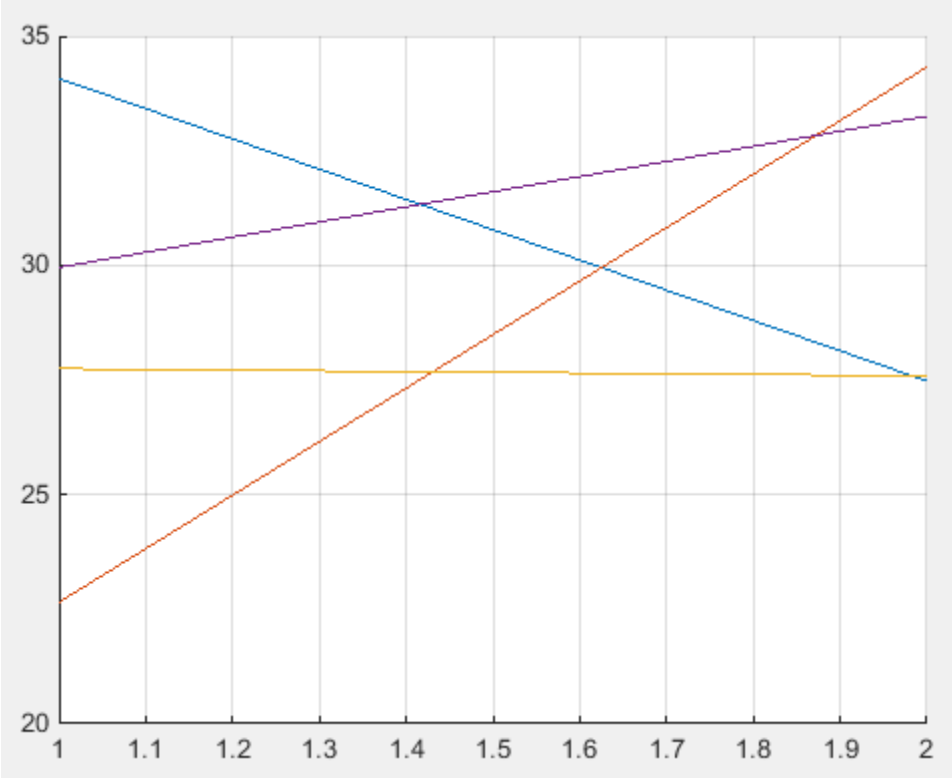

Figura 3-15. Distancias en el mTSP con ciudades auxiliares.

Como se explica en el código, se consideran 5 mejoras por defecto y se baraja cual es la mejor de ellas. En este caso, resulta ser la tercera, ya que el recorrido máximo es el mínimo en comparación con las otras mejoras (distancia de 29.96 unidades).

3.1.1.5.4 Salida por pantalla

```
intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573


intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573
   27.4759   22.6543   27.5884   35.4266   19.3573


intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573
   27.4759   22.6543   27.5884   35.4266   19.3573
   27.4759   22.6543   27.5884   29.9542   21.1900


intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573
   27.4759   22.6543   27.5884   35.4266   19.3573
   27.4759   22.6543   27.5884   29.9542   21.1900
   27.4759   22.6543   27.5884   33.1158   22.5022
```

```
intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573
   27.4759   22.6543   27.5884   35.4266   19.3573
   27.4759   22.6543   27.5884   29.9542   21.1900
   27.4759   22.6543   27.5884   33.1158   22.5022
   27.4759   22.6543   27.5884   29.9542   21.1900


intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573


intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573
   27.4759   22.6543   27.5884   35.4266   19.3573


intentos_mejora =

   14.5781   22.6543   27.5884   31.4649   19.3573
   27.4759   22.6543   27.5884   35.4266   19.3573
   27.4759   22.6543   27.5884   29.9542   21.1900
```

Figura 3-16. Salida por pantalla en el mTSP con ciudades auxiliares.

3.1.2 Comparativa

Una vez resuelta cada una de las variantes del mTSP propuestas, es interesante compararlas entre sí para decidir qué optativa resulta más atractiva si nos proponemos resolver el problema real del mTSP, en el cual varios viajeros se tengan que repartir tanto ciudades como distancia a lo largo de todo el recorrido.

Si se disponen solo de ciudades restringidas a repartir entre los distintos viajeros, como se comentó anteriormente, estamos ante la variante más sencilla. De hecho, el hecho de que cada viajero tenga asignadas las ciudades por las que debe pasar de antemano es lo que hace que el problema del mTSP cobre sentido.

Las otras dos variantes presentadas, en las que entran en juego ciudades auxiliares por las que, en principio, cualquier viajero puede pasar, tratan simplemente de aportar más flexibilidad a la primera variante del mTSP.

En el caso de que se disponga tanto de ciudades intermedias como de ciudades restringidas, es interesante que se decida a qué viajero le conviene pasar por cada ciudad auxiliar de manera que la distancia total que recorra no sea excesiva, respetando el margen de la distancia máxima del que tanto se ha hablado anteriormente.

Cuando hablamos de la última variante propuesta del mTSP, en la cual no hay ciudade restringidas, dicha variante no resuelve un problema de interés en el ámbito de aplicación real, ya que lo normal es que a cada viajero le corresponda al menos una ciudad por la que pasar obligatoriamente.

Si nos abstraemos e identificamos el viajero con un brazo robótico que solo puede coger piezas, la “ciudad obligatoria” por la que debe pasar el viajero, en este caso, se corresponde con la operación de “coger una pieza”. Resultaría absurdo, por ejemplo, asignarle a ese robot la operación de fregar el suelo si es otro robot el encargado de realizar esa operación (y diseñado para ello, además).

Por todo esto, se puede concluir que, aunque no resulte de interés real la última variante del mTSP que se ha explicado anteriormente, resulta interesante tratarla si se diese el caso en el que fuese irrelevante el tipo de ciudad por la que cada viajero tiene que pasar.

4 CONCLUSIONES Y DESARROLLOS FUTUROS

4.1 Problemas en el desarrollo de la solución

4.1.1 TSP

Para llegar a la solución presentada en este documento, se ha intentado resolver inicialmente el problema mediante algoritmos exactos, en concreto mediante el algoritmo de la "búsqueda de la fuerza bruta".

Mediante dicho algoritmo, se pretendía buscar la mejor solución considerando todas las soluciones existentes en nuestro problema, lo que resulta irrealizable debido al gran coste computacional cuando se consideran muchas ciudades intermedias. Sin embargo, se partió de este algoritmo para ir adaptando mejoras de manera progresiva hasta llegar a la solución del TSP que se ha presentado anteriormente. Para ello, se ha recurrido a árboles de nodos y los estados se han recorrido siguiendo la estrategia híbrida.

Como se dijo en el apartado correspondiente, aunque hubiese sido más eficaz recorrer el árbol de nodos siguiendo una estructura LIFO, se concibió la estrategia híbrida para simplificar el código propuesto para, al mismo tiempo, obtener buenas soluciones. Resultó más simple ir enumerando los nodos del árbol conforme se iban creando y, adoptando esta nueva estrategia, simplemente basta con aplicar el script "repetición" sobre los nodos en función de sus índices.

4.1.2 mTSP: reparto de ciudades auxiliaries

Ya se ha dicho en numerosas ocasiones que la variante del mTSP en la cual todas las ciudades son restringidas es el caso más sencillo y es el que menos varía partiendo del mTSP.

Si consideramos las otras dos variantes del mTSP en la que hay ciudades auxiliares a repartir, la dificultad radica precisamente en tener claro el reparto de dichas ciudades.

En el código que se ha propuesto anteriormente, en su correspondiente apartado, se ha considerado que la el viajero que tarde más en recorrer su ruta ha de ceder una ciudad auxiliar al viajero que menos haya tardado en su recorrido.

Como se ha visto en uno de los ejemplos de la última variante del mTSP, esto puede ocasionar que se entre en un bucle en el cual la ruta más larga que ceda una ciudad auxiliar vaya cediendo siempre la misma ciudad auxiliar al mismo viajero, por lo que se puede interpretar que no se puede encontrar una solución mejor.

Por esto mismo, en el código que se encarga de resolver este problema, simplemente se ha optado por fijar un margen de mejoras (5 en nuestro caso) y decidir la mejor sulución que se encuantra entre las cinco mejoras y la primera solución que se obtiene, sin aplicar mejora. Se ha hecho eso para simplificar las mejoras, y simplemente barajar unas pocas opciones para no estar completamente condicionados por el primer reparto de ciudades auxiliares que se lleva a cabo.

4.2 Mejoras del método propuesto

La solución que se ha presentado en este trabajo para resolver el problema del TSP, de cara a obtener la mejor solución que se pueda con el menor coste y con la mayor simplicidad posible, es el resultado de considerar inicialmente un recorrido que aporta un buena solución (un buen punto inicial) mediante el cual ir puliendo la solución final empleando una estructura de nodos.

El punto inicial que se considera, en nuestro caso, es el recorrido que el viajero sigue desde la ciudad inicial hasta la final pasando por la ciudad más cercana que se encuentra. La distancia obtenida en este recorrido, aunque es buena, no siempre es la óptima, y se puede apreciar de manera muy clara en el ejemplo que se presenta del TSP para 10 iteraciones, ya que la primera iteración es la solución (el punto inicial).

Aunque nos ha valido este punto inicial para obtener buenas soluciones del problema, también sería interesante

considerar métodos estadísticos que varíen el punto inicial, de manera aleatoria, para considerar otras soluciones que probablemente no se puedan barajar con nuestra propuesta, si limitamos a un número fijo de iteraciones.

De esta forma, se podrían barajar en el mismo tiempo de cálculo tanto una parte de las soluciones que partan de nuestro punto inicial como distintas soluciones que partan de un recorrido aleatorio.

El algoritmo genético[2], por ejemplo, se basa en esta idea para ir cambiando de una población a otra de manera aleatoria y, a partir de un nuevo "conjunto de datos", converger o no a una solución mejor.

Además, como bien se ha comentado en el apartado anterior, el empleo de una estrategia LIFO nos ayudaría a ir encontrando antes las mejores soluciones que si empleásemos la estrategia híbrida. Sin embargo, como se señaló anteriormente, se optó por desechar esta idea para no añadir complejidad al código porpuesto (aunque se podría hacer sin nigún problema).

5 BIBLIOGRAFÍA

[1] Naranjo, D., Rojas, A., & Quiceno, J. (2017). *Prezi*. Recuperado el 31 de Mayo de 2017, de https://prezi.com/zbxidng2bje4/tecnicas-de-diseno-de-algoritmos/

[2] Rudas, J., Fodor, J., & Kacprzyk, J. (2010). *Computational Intelligence and Informatics: Principles and Practice.* Berlin: Springer-Verlag, pp. 144-146.

Wolfram MathWorld. (1999-2017). Recuperado el 31 de Mayo de 2017, de http://mathworld.wolfram.com/TravelingSalesmanProblem.html

Cuevas Jiménez, E. V., Osuna Enciso, J. V., Díaz Cortés, M. A., & Oliva Navarro, D. A. (2016). *Optimización. Algoritmos programados con MATLAB.* S. A. Marcombo.

GLOSARIO

ISO: International Organization for Standardization 4

UNE: Una Norma Española 4