Trabajo Fin de Máster
Ingeniería Electrónica, Robótica y Automática

Efficient robot cooperation using MTSP

Autor: Jesús Caballero Rodríguez

Tutores: Begoña C. Arrue Ullés y Aníbal Ollero Baturone

Dpto. Teoría de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Máster
Ingeniería Electrónica, Robótica y Automática

Efficient robot cooperation using MTSP

Autor:
Jesús Caballero Rodríguez

Tutores:
Begoña C. Arrue Ullés y Aníbal Ollero Baturone
Profesor Titular

Dpto. Teoría de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Máster: Efficient robot cooperation using MTSP

Autor: Jesús Caballero Rodríguez
Tutores: Begoña C. Arrue Ullés y Aníbal Ollero Baturone

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Resumen

El objetivo principal de este trabajo consiste en implementar una solución del denominado *Problema del Viajero (TSP)* en el recorrido de un UAV simulado en el entorno de ROS (*Robotic Operating System*). En dicho problema, concebiremos al UAV que recorre una serie de waypoints de su recorrido como el viajero que se plantea recorrer una serie de ciudades, optimizando la distancia recorrida en el viaje.

Con ayuda de dicha analogía, supondremos que el viajero parte de una ciudad origen y pretende llegar a una ciudad destino pasando tan solo una vez por las ciudades intermedias del recorrido. Posteriormente, abordaremos la variante del MTSP (*Multiple Traveling Salesman Problem*) en la que se consideran varios viajeros que deben llegar a sus respectivos destinos, partiendo del mismo origen y repartiéndose entre ellos las ciudades intermedias del recorrido con el fin de obtener una solución aceptable conjuntamente.

Índice

Introducción

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.

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*.

Haciendo alusión a técnicas basadas en este algoritmo, se han desarrollado estudios cuyo propósito es desarrollar una solución eficiente basándose en el comportamiento de una colonia de hormigas [1].

Al márgen de desarrollar metodologías para resolver este problema de manera eficiente y con bajo coste computacional [2], también existen aplicaciones de gran interés. Entre ellas, es interesante considerar este algoritmo de optimización para aplicarlo en el reparto de mercancías a clientes, tratando de minimizar la distancia total en el recorrido. [3].

En el ámbito industrial, en concreto en el contexto de vigilancia y monitorizacion industrial por parte de robots aéreos, se desea optimizar la distancia recorrida por los robots o, de manera análoga, el tiempo empleado en su tarea de inspección o mantenimiento.

En lo relacionado con dichas tareas de inspección, se ha empleado la solución de este problema de optimización para aplicaciones relacionadas con tareas de vigilancia en entornos urbanos [4], llevadas a cabo por UAVs. En este caso, también se pretende optimizar el recorrido de dichos UAVs para lograr abarcar el espacio de inspección en el menor tiempo posible.

De igual manera, se ha recurrido a técnicas que emplean TSP con restricciones temporales [5], de manera que deba garantizar el cumplimiento de la misión que el robot de inspección está llevando a cabo en un determinado intervalo temporal especificado.

También es conocida, en tareas de inspección, una variante del TSP denominada *Dubins Traveling Salesman Problem with Neighborhoods (DTSPN)* [6], mediante la cual se consideran regiones distintas del entorno de inspección para resolver el problema de optimización. Cuanto más cercanas se encuentren las regiones entre sí, mejor opera el algoritmo, aunque bien es cierto que tiene un elevado coste computacional asociado. Este algoritmo considera varios puntos de cada una de las regiones del entorno para resolver el problema.

El *Dubins Traveling Salesman Problem* también está aplicado a problemas en los que participan vehículos con restricciones de curvatura asociadas a su cinemática [7]. Dicho problema se puede generalizar como un problema *k-DTSPN*, en el cual el algoritmo pretende encontrar $\boldsymbol{k}$ recorridos como solución, uno para cada vehículo. Con esta metodología, se pueden obtener soluciones en un intervalo de tiempo lo suficientemente corto como para poder emplearse en planificación en tiempo real.

De la misma manera, el *Polygon-Visiting DTSP* [8] es una variante alternativa que permite al algoritmo operar más rápido, que persigue el objetivo de pasar por al menos un punto de cada una de las regiones en las que se considera dividido el entorno de inspección [9]. Dichas regiones, como el propio nombre de la variante del *Travelling Salesman Problem* indica, se consideran como polígonos por simplicidad.

Este método del *DTSP* proporciona una camino de referencia (recorrido solución) en tiempo real para tareas ISR (*Intelligence, Surveillance and Reconnaissance*) de inspección, como se ha aplicado en numerosos casos a día de hoy [10].

Lo que aporta el algoritmo que se propone y explica a lo largo de esta memoria, y que no especifican las demás referencias señaladas hasta ahora, es la posibilidad de resolver el problema de optimización más rápido, teniendo la posibilidad de operar con varios subprocesos para llegar a soluciones del algoritmo de manera concurrente, y abarcar el grafo del viajero en menor tiempo, dando la misma solución óptima de recorrido requerido en la supuesta inspección de UAV.

Se tratará más adelante con profundidad de qué manera se ofrece esta mejora con nuestra concepción del algoritmo.

En lo relativo al reparto de ruta por parte de **varios UAVs**, se ha implementado el ***Multiple Depot UAV Routing Problem (MDURP)*** [11]. Mediante este método, dada una serie de UAVs que comienzan su recorrido partiendo de diferentes puntos, se trata de asignar un camino a cada UAV, considerando que cada uno de los robots tiene un destino asignado exclusivamente para él, y que además tan sólo se puede pasar una vez por cada punto de su recorrido, sin que distintos UAVs pasen por el mismo punto. Como coste en esta variante, se toma la distancia total recorrida por todos los viajeros (UAVs) a lo largo de la supuesta inspección.

En nuestro caso particular que se aborda en la memoria, en lugar de considerar el coste total de todas las rutas, y en vez de suponer que cada UAV tiene un origen distinto en la inspección, se ha supuesto que todos los UAVs salen del mismo sitio y que se debe minimizar el máximo recorrido de entre todos los considerados en la solución, en lugar de considerar la distancia total acumulada considerándose todos los UAVs. Con esta metodología que se ha adoptado en nuestra solución propuesta, aseguramos que no obtendremos una ruta excesivamente larga en nuestra inspección, mientras que de la manera presentada en la referencia citada anteriormente no aseguramos eso, ya que considera la distancia total y no la de cada viajero. Nuestra solución será mejor en ese aspecto, teniendo en cuenta que nuestro algoritmo irá mejorando el resultado global en cada iteración (como se explicará en el apartado correspondiente).

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* [12], 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 Traveling 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 distintos viajeros del problema han de repartirse el recorrido, por lo que no debe haber demasiada varianza entre las distintas distancias obtenidas en el resultado.
- 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.

La solución del *Problema del Viajero* tendrá como objetivo generar el orden en el que un supuesto UAV deba recorrer sus waypoints asignados, de manera que se recorran dichos puntos con la distancia total mínima considerándose su velocidad como constante.

Dadas las coordenadas por las que el UAV debe realizar una supuesta misión de inspección, y considerando tanto puntos de partida como de destino de dicha misión, se genera el orden en el que el UAV debe recorrer los puntos de la trayectoria para minimizar el recorrido total del viaje para, con ello, conseguir minimizar el gasto adicional en batería y en tiempo de la misión.

La solución del *Problema del Viajero* basada en nuestra propuesta, que se explicará a continuación, puede permitir facilitar una supuesta misión de inspección de UAVs, en la que sea necesario abarcar un territorio de extensión considerable dadas coordenadas de puntos del recorrido. Por ejemplo, en un entorno de placas solares sería interesante usar esta técnica para decidir el orden de inspección y tratamiento de defectos.

En el apartado correspondiente, una vez que se haya expuesto y resuelto el problema del viajero, se explicará de qué manera se incluye la librería del MTSP desarrollada en este ejemplo de UAV.

Metodología

Se ha llevado a cabo la programación necesaria para resolver el problema, tanto del *TSP* como del *MTSP*, empleando el lenguaje de programación C.

Se han empleado nuevas herramientas de programación, con respecto a la solución propuesta en *MATLAB* del algoritmo, que simplifican su resolución. Además de esta simplicidad, otra ventaja asociada al empleo de estas nuevas herramientas consiste en la sencillez con la que se puede implementar el código en la librería de ROS.

Se explicará posteriormente en qué consisten dichas técnicas de programación y en qué medida mejora el algoritmo con respecto al caso sintetizado anteriormente en *MATLAB*.

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 recordará a grandes rasgos cómo se resolvió en problema empleando el lenguaje de programación MATLAB.

Una vez hecho esto, se propondrá la nueva solución empleada en el entorno de ROS, empleando el lenguaje de programación C.

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. Hablaremos, en primer lugar, de la solución usada con Matlab para compararla posteriormente con la solución hecha en el entorno de ROS.

Una vez se hayan explicado cada una de las variantes, qué se quiere conseguir y cómo queremos lograrlo, presentaremos las **clases usadas para resolver tanto TSP como MTSP** en ROS con el empleo del lenguaje C.

Tras esto, se explicará de qué manera se implementa la solucion de este problema de optimización en el **vuelo de un UAV** (*Vehículo Aéreo no Tripulado*) en una simulación en *Gazebo*, que forma parte del entorno de ROS.

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, algo que servirá de punto de partida para discutir tanto las ventajas como los inconvenientes del método propuesto.

Descripción del sistema y algoritmia: TSP

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 embargo, ¿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.

Resumen del método anterior (MATLAB)

En la implementación que se llevó a cabo en MATLAB, considerábamos un grafo que ilustraba los posibles recorridos que el viajero podía llevar a cabo desde la ciudad de origen hasta el destino.

Los nodos que conformaban dicho grafo almacenaban información bastante relevante en cuanto al recorrido. Principalmente, proporcionaban información acerca de la distancia total recorrida hasta la ciudad actual del viaje, nos decía por qué ciudades habíamos viajado anteriormente (y en qué orden) y en la ciudad en la que el viajero se encontraba en ese momento. Además, cada nodo del grafo almacenaba más información relacionada con las posibles rutas que quedaban por explorar en el grafo para obtener más soluciones del problema de optimización.

Sin embargo, en cuanto a las transiciones entre nodos, estas no guardaban ningún tipo de información; tan sólo aparecían en el grafo del viaje para ilustrar la conexión entre varios nodos o estados del árbol.

Por ejemplo, una representación del grafo para tres ciudades intermedias podía ser la siguiente:

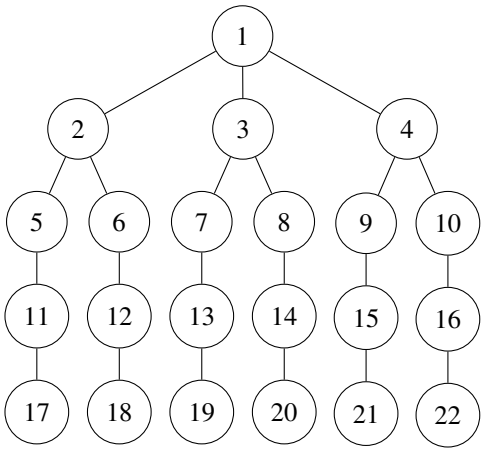

Figura 2.1 Grafo de estado para tres ciudades intermedias.

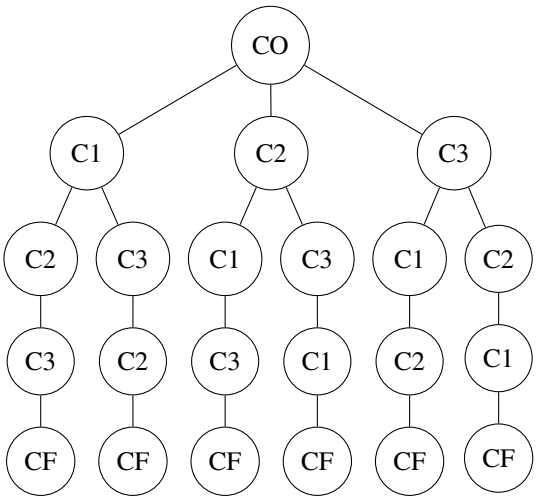

Figura 2.2 Grafo de estado para tres ciudades intermedias; ciudades.

En esta última representación, se muestra el grafo de estados con el nombre de la ciudad asociada a cada nodo.

Por ejemplo, el nodo padre del árbol (*CO*) almacena toda la información asociada a ese punto del viaje; el viajero ha pasado por la ciudad origen y puede ir a las ciudades *C1*, *C2*, o *C3*.

Toda esta información, como se comentó, asociada a los nodos. De esta menera, se obtiene una base de datos considerable con estructuras de nodos, en las que se guarda toda la información.

Por otra parte, los hijos de un nodo padre siempre aparecen ordenados de menor a mayor distancia en el viaje entre ciudades, de manera que el viaje *CO-C1* es el que almacena menor distancia.

Una posible solución del grafo se podría representar de esta otra forma:

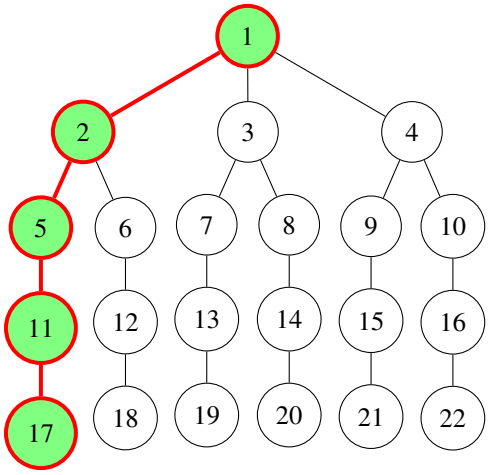

Figura 2.3 Posible solución del grafo para 3 ciudades intermedias.

En concreto, la última solución mostrada en imagen representa el recorrido del viajero pasando a la ciudad más cercana desde la que de encuentra en ese momento. **Esto no implica que la mejor solución sea la rama situada más a la izquierda del grafo, pero sí que suela proporcionar una buena solución**.

A continuación, a partir del próximo apartado, se explicará cómo se ha reformulado el problema para obtener soluciones con una complejidad a nivel de programación mucho menor.

Incluso, se baraja la posibilidad de servirse del paralelismo entre procesos que brindan los hilos o subprocesos, para poder obtener más soluciones de este problema de programación en menos tiempo.

Método de implementación en ROS: empleo de clases

Para reestructurar el problema desde cero, como se acaba de comentar en el apartado anterior, es necesario adoptar un enfoque distinto al que se ha llevado a cabo hasta ahora para resolver este problema de optimización.

En concreto, a partir de ahora nos serviremos de la programación orientada a objetos para concebir distintos tipos de variables con información que se complemente entre sí; todo esto para poder resolver este problema de una manera mucho más liviana.

Por ejemplo, a partir de ahora emplearemos objetos de la **clase Node** que no tendrán por qué almacenar toda la información del recorrido que ha llevado a cabo el viajero hasta ese momento (como se hacía anteriormente).

En lugar de ello, la **clase Branch** definirá un tipo de objeto que será capaz de obtener información acerca del recorrido completo del viajero, sirviéndose a su vez de la **clase Node**.

También emplearemos una **clase** llamada **Path** que almacenará la distancia entre ciudades del recorrido, de manera que los nodos no almacenen más información de la necesaria (que es algo que ocurría en la implementación con Matlab).

En los próximos apartados, se explicará con detenimiento el cometido de cada una de las clases definidas y qué papel jugarán cuando tratemos de abordar el problema completo.

Toda la implementación de clases se llevará a cabo, además de para disminuir la complejidad de programación, para prescindir de una base de datos de un tamaño considerable en la que se almacene toda la información, como se hacía en el caso explicado en el apartado anterior con MATLAB.

Introducción a las clases definidas

Un buen ejemplo para ilustrar el tipo de objetos que se manejará a nivel de programación lo podemos encontrar en la estructura que presenta un árbol normal y corriente.

¿Por qué razón? Nuestro objetivo principal es obtener la solución que haga que la ruta del viajero sea la de menor recorrido posible o, al menos, la de menor distancia en el viaje considerando el resto de soluciones obtenidas.

Si considerásemos el árbol mencionado anteriormente, podríamos imaginar que la ciudad de inicio del viaje se corresponde con el tronco del árbol, mientras que el extremo de todas y cada una de las ramas del árbol se corresponden con las soluciones al problema del viajero. Podríamos entender las bifurcaciones en las ramas del árbol como las ciudades intermedias, a partir de las cuales podemos viajar a otras ciudades.

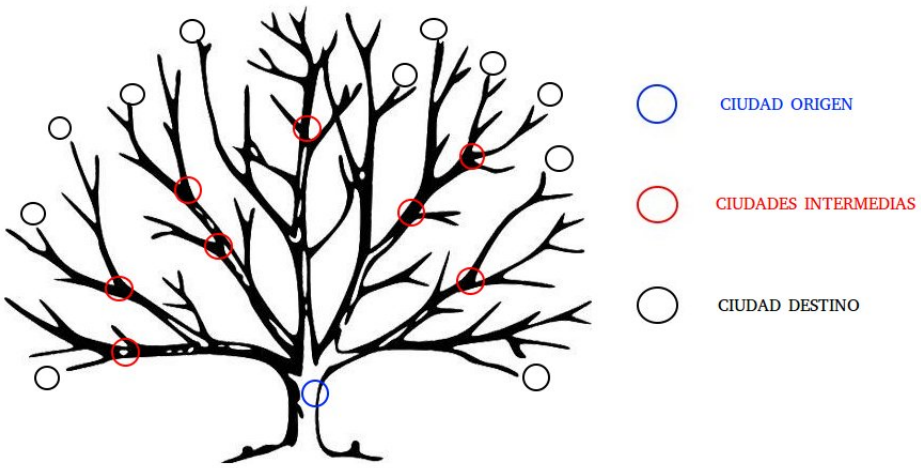

Figura 2.4 Algoritmo en clases: símil con un árbol.

Si nos interesase obtener las mejores soluciones, posiblemente la mejor opción sería buscar la ramificación más corta que parta del tronco del árbol y que llegue hasta el extremo final de una rama.

Vemos que estamos materializando el problema que tratamos de resolver en un objeto del mundo real, que se encuentra en la naturaleza: un árbol.

Este árbol está formado por lo que podrían ser objetos con características distintas. Por una parte, podemos considerar las bifurcaciones entre las ramas, así como las ramas que van uniendo bifurcaciones entre sí.

Teniendo en cuenta las bifurcaciones y las secciones de las ramas en el árbol, podrían definirse a su vez ramas más largas que partan desde el tronco y lleguen cerca del extremo final, el cual se puede definir como ciudad de destino.

Todas estas partes del árbol (así como el propio árbol) se pueden definir como objetos en un contexto de programación. A continuación, hablaremos de las clases que definen a dichos objetos y del tipo de información que tienen asociada.

City

En el árbol mencionado, podemos considerar las bifurcaciones como puntos en los que cambiar el recorrido de una rama hacia otra distinta, para obtener soluciones distintas a su vez. En estos puntos de inflexión estará la información asociada a las ciudades por las que el viajero irá pasando al llevar a cabo su recorrido.

Por ello, en primer lugar, podemos definir la **clase City** que contenga las coordenadas de las ciudades consideradas en todo el recorrido del viajero, así como un identificador unívoco que permita distinguirlas del resto de ciudades que se tienen en cuenta en el viaje.

De ahora en adelante, ilustraremos mediante diagramas UML (*Unified Modeling Language Diagram* [13]) las clases que definirán a los objetos que se emplearán en nuestro programa. Haciendo alusión a dicho diagrama, el de la **clase City** tendría esta forma:

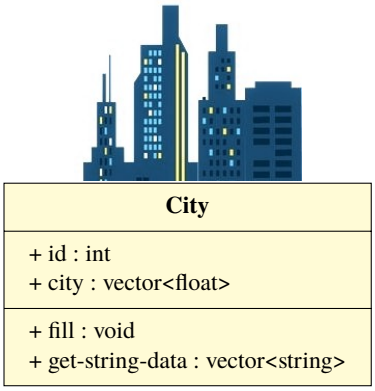

Figura 2.5 Diagrama UML: Clase City.

En el diagrama mostrado, podemos apreciar que hay tres secciones divididas horizontalmente y que definen conjuntamente la clase City, con la que se podrán definir los objetos que serán las ciudades.

En las tres divisiones, respectivamente, aparece información acerca de:

1. Nombre de la clase. (City)
2. Atributos de la clase City (valores almacenados).
3. Métodos de la clase (funciones que permiten operar sobre los atributos de la clase).

En cuanto a la marca que aparece antes de nombre del atributo o método, nos dará información acerca de la visibilidad del atributo o método al que esté haciendo referencia ("+" significa público, mientras que "-" significa privado).

A nivel de código, la **clase City** quedaría definida de esta manera en lenguaje C, en el anexo [A.1].

Node

Aunque en el subapartado anterior hayamos definido las ciudades que formarán parte del recorrido del viajero, esta información no es suficiente para definir de manera unívoca las bifurcaciones en el árbol que presentamos anteriormente.

Dicho de otra manera: necesitamos un identificador que distinga una misma ciudad por la que se ha pasado en rutas diferentes del algoritmo. Para ello, consideraremos y definiremos una nueva clase en la que se almacenarán tanto la ciudad del recorrido como su identificador en en árbol. El identificador, a nivel de programación, lo definiremos como un entero.

A esta nueva clase la llamaremos **clase Node**. En el siguiente esquema, tratamos de ilustrar mediante un diagrama UML la definición que acabamos de aportar:

Como consideraremos un grafo de estados, igual que hicimos en el caso del algoritmo resuelto con MATLAB, también declararemos un atributo que almacene el nivel que ocupa el nodo dentro del grafo. Esto se hará para facilitar la implementación de la **clase Branch** (que presentaremos más adelante).

Con respecto a los métodos de la clase, hemos definido los siguientes:

Node
+ label : int + level : int + city : City
+ Node: + fill: void + convert-element : void

Figura 2.6 Diagrama UML: Clase Node.

1. **Node:** Constructor de la clase.
2. **fill:** Dados los atributos como argumentos de entrada, define el nodo.
3. **convert-element:** Convierte un elemento a la clase Node. Más tarde explicaremos la **clase Element**.

A nivel de código, los métodos de la **clase Node** se han definido de la siguiente manera en el anexo [A.2].

Path

Los objetos definidos con la **clase Node** representaban las bifurcaciones en el árbol, y almacenaban las coordenadas de las ciudades del recorrido del viajero.

Ahora, definiremos la unión entre dichas bifurcaciones; lo que serán porciones de la rama del árbol que representan los caminos que unen las bifurcaciones (los nodos).

Para ello, daremos nombre a una nueva clase; la **clase Path**. En primer lugar, mostraremos el diagrama UML que ilustra esquemáticamente tanto los atributos como los métodos de la clase. Posteriormente, explicaremos brevemente en qué consiste cada campo.

Path
+ origin-id: int + destiny-id: int + distance: float
+ Path: + fill: void

Figura 2.7 Diagrama UML: Clase Path.

- **ATRIBUTOS:**
  1. **origin-id:** Entero que almacenará el id de origen del objeto definido. Posteriormente, asociaremos este identificador al identificador del nodo del que parte este camino.
  2. **destiny-id:** Entero que almacenará el id de destino del objeto definido. Posteriormente, asociaremos este identificador al identificador del nodo al que llega este camino.
  3. **distance:** Almacena la distancia recorrida en este camino.
- **MÉTODOS:**
  1. **Path:** Constructor de la clase.
  2. **fill:** Dados los atributos como argumento de entrada, define el objeto de la **clase Path**.

A nivel de código, los métodos de la **clase Path** se han definido de la siguiente manera, en el anexo [A.3].

Branch

Necesitamos crear un tipo de objeto nuevo que contenga tanto objetos tipo **Node** como objetos tipo **Path**, para saber por qué ciudades ha pasado el viajero, el orden en que las ha recorrido y la distancia que ha acumulado hasta ese momento.

Necesitaremos objetos de la clase **Node** para conocer las dos primeras características de la rama, mientras que la distancia total de las ramas la obtendremos acumulando las distancias de los muchos objetos de la clase **Path**.

De esta manera, se definirá la clase **Branch**, que tendrá una asociación con la clase **Elements**, la cual explicaremos en el próximo apartado.

El diagrama UML de la clase **Branch** presenta esta forma:

Branch
+ elements : deque <Element> + last-element : int vector<float>
+ Branch : + get-cities : vector<City> + get-route : vector<Node> + add-element : void + get-total-distance : float + get-total-distance-destiny-id : float + search-node : bool + search-path : bool

Figura 2.8 Diagrama UML: Clase Branch.

- **ATRIBUTOS:**
  1. **elements:** Se trata de una cola que almacena los elementos que conforma la rama. Reseñaremos que los elementos pueden ser o bien objetos de la **clase Node**, u objetos de la **clase Path**.
  2. **last-element:** Puede tomar tres valores "NODE", "PATH" o "-1", si no se encuentra definido.
- **MÉTODOS:**
  1. **Branch:** Constructor de la clase: inicializa "last-element" a "-1". Es decir, creamos una rama que aún no tiene elementos.
  2. **get-cities:** Devuelve un vector con las ciudades de la rama que estamos considerando. Los elementos del vector (las ciudades) están ordenados desde el inicio del recorrido del viajero hasta la última ciudad de la rama.
  3. **get-route:** Devuelve un vector con los nodos del recorrido de la rama que estamos considerando, hasta llegar al nodo de la rama que se toma como argumento de entrada de este método.
  4. **add-element:** Inserta un elemento al final de la rama. Cuando indaguemos en el código de este método, se explicarán más detalles.
  5. **get-total-distance:** Devuelve la distancia total de la rama, acumulando las distancias de los elementos que son **Path**.
  6. **get-total-distance-destiny-id:** Devuelve la distancia de la rama hasta llegar a un objeto **Path** con el **destiny-id** especificado como argumento de entrada del método.
  7. **search-node:** Nos dice si, en la rama considerada, se encuentra el nodo (objeto de la clase Node) que se toma como argumento de entrada del método.
  8. **search-path:** Nos dice si, en la rama considerada, se encuentra el camino (objeto de la clase Path) que se toma como argumento de entrada del método.

El código de la cabecera de esta **clase Branch** tiene el siguiente aspecto, en el anexo [A.4].

Hay que señalar que el tamaño de las distintas ramas que se definen en el algoritmo no tendrán el mismo tamaño necesariamente. Esto significa que, en ciertas ocasiones, se pueden llevar a cabo inserciones de distintas ramas entre ellas, con el fin de conformar una sola rama quu abarque el recorrido completo del viajero, desde la ciudad de inicio hasta la ciudad de destino.

Element

Explicaremos la clase introducida anteriormente, a la que hemos llamado **Element**.

Esta clase define un tipo de objeto que puede tomar la forma o bien de un **Nodo**, o bien de un **Camino** (ambos explicados anteriormente en sus apartados correspondientes).

El constructor de la clase **Element** es el que nos dirá cuál de los dos objetos será el que estamos utilizando.

Se recurre, de esta manera, a una técnica denominada **polimorfismo**, mediante la cual tenemos la posibilidad de definir clases diferentes que tienen métodos o atributos denominados de forma idéntica, pero que se comportan de manera distinta. En nuestro caso en particular, las clases **Node** y **Path** comparten los mismos atributos, que se inicializarán o no en función de cómo declaremos el elemento en cuestión (nodo o camino). De esta manera, podemos decir que se da herencia [14] entre clases.

¿Por qué llevamos a cabo esta técnica para definir las clases **Node** y **Path** a partir de **Element**?

Resulta sencillo de entender: ya que hemos definido anteriormente la clase **Branch**, lo más adecuado es que los objetos que se definan almacenen **nodos** y **caminos**, recordando la analogía con el árbol que se hizo anteriormente.

Así mismo, los nodos y caminos que conformen la rama deben estar definidos como un mismo tipo de objeto, aunque obviamente tengan tanto atributos como métodos diferentes. La razon fundamental para proceder de esta manera es que, si definimos un atributo de la clase **Branch** que almacene un vector con los objetos que definen la rama, desde el nodo de inicio hasta el nodo final, todos los objetos deben ser del mismo tipo. La clase que define dicho tipo de objeto es la clase **Element**.

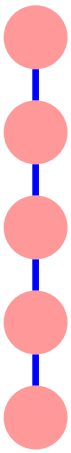

Figura 2.9 Elementos de la rama. En rojo los nodos. En azul los caminos.

A continuación, se muestra el diagrama UML que ilustra el polimorfismo que se ha tratado de explicar, con el fin de tratar de comprender por qué hemos decidido definir los **nodos** y los **caminos** a partir de una misma clase:

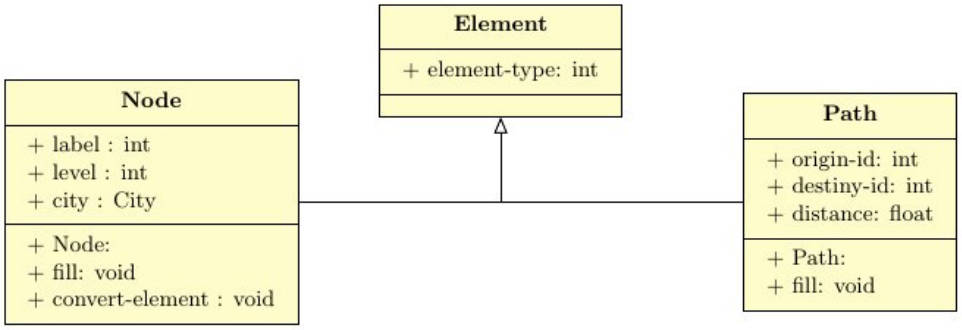

Figura 2.10 Diagrama UML: Herencia de la clase Element.

- **ATRIBUTOS:**
  1. **element-type:** Puede tomar tres valores "NODE", "PATH" o "-1", si no se encuentra definido.

En el anexo [A.5], se presenta el código que define la cabecera de esta clase.

Graph

En esta sección definiremos la **clase Graph**, que representa el árbol completo de recorridos con el que se trató de introducir todas las clases vistas hasta ahora.

La **clase Graph** se servirá de ramas completas para almacenar las distancias recorridas por el viajero en cada una de las consideraciones posibles del algoritmo. Para llevar a cabo esta tarea, esta clase define una serie de atributos que permitirán definir, para cada solución del TSP, los nodos y los caminos considerados para esa solución en concreto. Una vez que se expliquen los métodos definidos en esta clase (cosa que haremos a continuación), entenderemos el cometido de estos atributos.

A nivel de código, en lenguaje C, la **clase Graph** queda definida de esta manera en el anexo [A.6].

Podemos apreciar que hay un gran número de atributos y, sobre todo, de métodos que permiten hacerle diversas modificaciones al árbol de recorridos. Abordemos de manera superficial cada uno de los elementos definidos por la clase, explicando su significado sin obviar los detalles más relevantes:

- **ATRIBUTOS:**
  1. **traveler:** Puntero de la **clase Node** que almacena la dirección de memoria del nodo en el que el viajero se encuentra en cada momento. Será uno de los nodos que conforman la cola a la que hemos llamado **nodes**.
  2. **break-down:** Puntero de la **clase Node** que almacena la dirección de memoria del nodo perteneciente a la cola nodes **a partir del cual se ha comenzado a desglosar la rama del árbol que se considera actualmente**.
  3. **nodes:** Cola que almacena los nodos del árbol considerados en la rama completa con la que opera el algoritmo actualmente. Se ha definido como cola en lugar de como vector (al igual que otros atributos) debido a que **la dirección de memoria de los elementos a los que apuntan los punteros no se ve alterada, aunque quitemos elementos de la cola conforme la vayamos modificando**. Esta cola se va reseteando conforme cambiamos la rama que consideramos.
  4. **paths:** Cola que almacena los caminos del árbol considerados en la rama completa con la que opera el algoritmo actualmente. Se ha definido como cola por el mismo motivo por el cual el atributo **nodes** se definió también como cola. Esta cola se va reseteando conforme cambiamos la rama que consideramos.

5. **global-node-id:** Etiqueta con la cual se van numerando los nodos creados en la cola **nodes**. Resulta de gran utilidad, ya que al desglosar nodos de un nivel inferior, se asignan etiquetas a los nodos creados en función de la distancia que añadimos al recorrido total.
6. **global-path-id:** Etiqueta con la cual se van numerando los caminos creados en la cola **paths**. Su utilidad es análoga a la explicada para anteriormente para la cola de los nodos.
7. **branches-id:** Etiqueta con la caul se van numerando las ramas creadas en el árbol completo.
8. **branches:** Ramas creadas en el objeto definido mediante la **clase Graph**.
9. **cities:** Ciudades que considera el problema del TSP. Dichas ciudades son la **ciudad de origen**, la **ciudad de destino** y las **ciudades intermedias**.

Variables de salida (se emplearán posteriormente por objetos de otro tipo de clase):

10. **n-branches-private:** Número de ramas que se han creado al finalizar el algoritmo del TSP (y, por tanto, número de soluciones obtenidas).
11. **factorial:** Nos dice cuántas soluciones se pueden obtener. Este valor se obtiene en función del número de ciudades intermedias.
12. **distances:** Vector que almacena la distancia total obtenida de las ramas.
13. **level-:** Etiqueta requerida por la **clase WholeGraph**, de la que hablaremos más adelante.
14. **route-out:** Vector con los nodos que conformarán el recorrido final seguido por el viajero; aquel en el cual la distancia obtenida es la mínima de entre todas las soluciones consideradas.
15. **begin:** Almacena el instante en el que empieza a ejecutarse el algoritmo principal. Se utiliza en el método correspondiente que trata de medir el tiempo que tarda en ejecutarse el algoritmo del TSP (método **main-algorithm**).
16. **min-distance:** Distancia mínima obtenida, de entre todas las ramas que se han almacenado en el árbol.

- **MÉTODOS:**
  1. **Graph:** Constructor de la clase. Aquí, se inicializan los atributos principales de la clase y, por simplicidad, posteriormente se llama al algoritmo principal que obtiene la solución del TSP.
  2. **print-nodes-data-base:** Imprime por terminal los nodos obtenidos hasta el momento, conforme se va sintetizando la rama en la que se encuentra viajando el viajero (la rama con un nodo al que apunta el atributo **traveler**) .
  3. **print-paths-data-base:** Imprime por terminal los caminos obtenidos hasta el momento, conforme se va sintetizando la rama en la que se encuentra viajando el viajero (la rama con un nodo al que apunta el atributo **traveler**) .
  4. **print-data-base:** Muestra las dos bases de datos; la de nodos y la de caminos.
  5. **explore-and-order:** Tomándose como entrada el nodo de la cola al que apunta **traveler**, se obtienen todos los nodos hijos de aquel en el que se encuentra el viajero y se inicializan con sus atributos correspondientes, definidos anteriormente en el apartado de la **clase Node**. Esta es la parte que se dedica a la exploración, aunque en este método también hay una sección de código encargada de ordenar los nodos hijos del nodo al que apunta el viajero (se les asignan etiquetas de menor a mayor distancia acumulada en este viaje).
  6. **distance-between-nodes:** Con este método, calculamos la distancia entre dos nodos cualesquiera que se tomen como entrada (es decir, la distancia entre las ciudades que los nodos almacenan).
  7. **travel-nearest-city:** El puntero **traveler** pasa a apuntar a su nodo hijo con el cual se acumule la menor distancia posible en este viaje (el nodo con la menor etiqueta de entre todos sus hijos).
  8. **main-algorithm:** Algoritmo principal que resuelve el TSP, dado los parámetros necesarios para ello.
  9. **remove-branches-elements:** Conforme se van creando nuevas ramas a lo largo del algoritmo para obtener nuevas soluciones, tratamos de descartar nodos hijos que ya se hayan considerado anteriormente para obtener otras ramas, ya existentes.

10. **break-down-son:** Con este método, buscamos el nodo hijo de aquel al que **break-down** apunta, y que tenga la etiqueta de valor mínimo. Este método se incluye en otro que resuelve un problema de mayor nivel de abstracción; se comprenderá mejor en el contexto apropiado.
11. **create-branch:** Crea y añade una nueva rama al atributo **branches**.
12. **refresh-graph-data:** Inicializa los valores pertinentes del árbol para poder sintetizar la siguiente rama en el algoritmo principal.
13. **move-break-down:** Cambia la dirección de memoria del puntero **break-down**, con el fin de que apunte al nodo a partir del cual se obtendrá la siguiente solución del árbol.
14. **get-route:** Dado un nodo del árbol completo, este método devuelve un vector de nodos con el recorrido completo del viajero **hasta llegar al nodo de entrada de este método**.
15. **print-results:** Imprime los resultados obtenidos al ejecutar el algoritmo del TSP. Entre los datos más importantes se encuentran:
    - Tiempo de ejecución.
    - ID de la rama con la mejor solución.
    - Número de ramas requeridas en la solución del algoritmo.
    - Número de ramas obtenidas por el algoritmo.
    - Ciudades recorridas por el viajero y orden en el cual se han recorrido.
    - Distancia mínima obtenida.
16. **get-total-distance:** Proporciona la distancia total de una rama concreta del grafo completo.
17. **route-output:** Devuelve la ruta con los nodos que conforman la mejor ruta obtenida por el algoritmo.
18. **min-distance-output:** Menor distancia obtenida con el algoritmo del TSP (y, por tanto, mejor solución).

Tengamos en cuenta que, con el fin de simplificar el apartado, se ha decidido explicar únicamente los campos definidos por la **clase Graph** sin considerar un diagrama auxiliar.

WholeGraph: empleo de varios hilos

Una vez que hemos explicado la **clase Graph** que define por completo el problema del viajero (y aporta la mejor solución dadas las entradas correspondientes), podemos ir un paso más allá.

Ya que, en el lenguaje de programación C, se pueden definir hilos para abordar varios problemas en paralelo, emplearemos esta misma técnica cuando queramos resolver el problema del viajero.

Pensando en la analogía que empleamos con el árbol al principio de este apartado, podemos pensar que varias personas tratan de buscar el extremo de las ramas del árbol partiendo desde su tronco.

A nivel de programación, esto se puede traducir en que hay varias tareas o subprocesos (hilos) que, de manera simultánea, buscan distintas soluciones al problema.

¿Con qué fin hacemos esto? Lo que buscamos es, simplemente **obtener más soluciones al problema en menos tiempo**. En nuestro caso en particular, se ha optado por emplear 4 hilos simultáneamente con el fin de mostrar la mejora notable que se lleva a cabo, comparando los resultados con el caso en el que se emplease un solo hilo.

La definición de la **clase WholeGraph**, a nivel de código, se muestra en el anexo [A.7].

- **ATRIBUTOS:**
  1. **res-cities:** Vector que almacena las ciudades restringidas que se repartirán entre los viajeros del *MTSP*.
  2. **aux-cities:** Vector que almacena las ciudades auxiliares que se repartirán entre los viajeros del *MTSP*.
  3. **min-distance:** Distancia mínima obtenida en el *MTSP*, que se corresponde con el viajero que ha tardado menos en su recorrido (de entre todos los que consideramos en el *MTSP*.)

- **MÉTODOS:**

  1. **output-little-branches:**

  Muestra los resultados para una de las subramas del grafo. La siguiente imagen muestra un ejemplo de la zona del grafo en la que se obtienen soluciones, señalado en rojo:

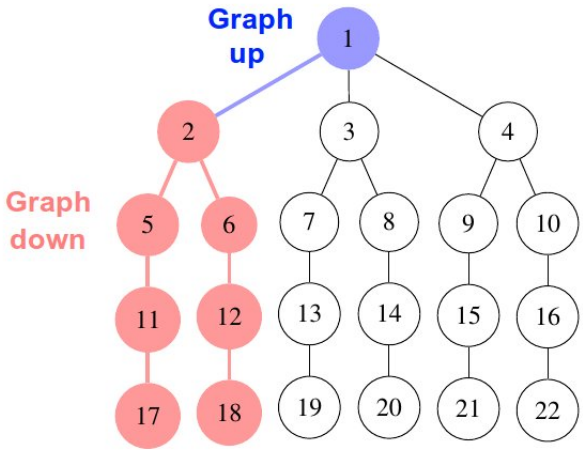

Figura 2.11 Algoritmo en clases: símil con un árbol.

  La zona roja se corresponde con el argumento del método al que hemos llamado *Graph down*. Por otro lado, la zona negra se corresponde con el argumento *Graph up*.

- **output-whole-graph:** Consideramos el caso en el que empleamos un solo hilo para resolver el problema del viajero. En este caso, tan solo se consideraría el árbol *Graph down*, de manera que que se consideraría el árbol de nodos al completo.
- **main-algorithm:** Encargado de resolver el *MTSP*, especificados el número de iteraciones deseado, ciudades por las que viajar, etc.
- **add-aux-cities:** De entre todas las ciudades especificadas como argumento de entrada, se añaden a las ciudades auxiliares (*aux-cities*) aquellas que tienen ids coincidentes con los identificadores especificados como argumento de entrada (*ids-to-include*).
- **add-res-cities:** De entre todas las ciudades especificadas como argumento de entrada, se añaden a las ciudades restringidas (*res-cities*) aquellas que tienen ids coincidentes con los identificadores especificados como argumento de entrada (*ids-to-include*).

Hay que señalar que **la decisión de resolver el problema con varios hilos o no se resuelve con el método del algoritmo principal; main-algorithm**.

Descripción del sistema y algoritmia: MTSP

Introducción

Una vez resuelto el problema del 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, tratan de llegar cada uno a su respectiva ciudad destino.

A esta variante del TSP la denominamos MTSP (*Multiple Traveling Salesman Problem*).

Se tratará en profundidad la manera en la que hemos abordado el problema con los conceptos de clase introducidos en apartados anteriores, para implementar el algoritmo en ROS.

Debido a que la idea empleada es similar al caso del MTSP resuelto con MATLAB, se explicará la nueva clase **Mtsp** para resolver esta variante, y de qué manera maneja los objetos del tipo **WholeGraph** que representan los grafos de cada viajero.

MTSP: variantes consideradas

Se considerarán tres variantes para resolver el *Multiple Traveler Salesman Problem*, definidas en función del tipo que tendrán las ciudades intermedias en el problema.

Las ciudades intermedias podrán ser:

- **Ciudades Restringidas**: aquellas en las que se sabe de antemano qué viajero debe pasar por ella.
- **Ciudades Auxiliares**: aquellas que no tienen un viajero asignado de antemano para que pase por ellas. Es este tipo de ciudad la que hará que el algoritmo del MTSP deba barajar qué viajero debe pasar por qué ciudad intermedia.

A continuación se explican brevemente cada uno de los casos considerados.

Sólo ciudades restringidas

Todas las ciudades intermedias del problema son restringidas y, por tanto, cada viajero ya tiene asignadas las ciudades por las que debe pasar hasta llegar a su destino. Resolver este problema equivale a resolver **n** problemas del viajero simples (TSP); uno para cada viajero del MTSP considerado.

Sólo ciudades auxiliares

En esta variante, todas las ciudades deben repartiese entre los distintos viajeros que definen el MTSP.

En función del número total de viajeros (suponemos que es **n**), se hará un reparto equitativo del número de ciudades auxiliares del problema (**aux**) entre los viajeros. En caso de que la división $\frac{aux}{n}$ no tenga resto, no habrá inconveniente y cada viajero tendrá el mismo número de ciudades auxiliares asignadas que cualquier otro. Si, por el contrario, el resto es mayor a 0, repartiremos dicho valor de manera equitativa entre los viajeros, sin mostrar especial preferencia en el reparto.

Una vez asignado el número de ciudades intermedias a cada viajero, se resolverá el problema del TSP para cada viajero sucesivamente.

¿De qué manera? El primer viajero considerará $\mathbf{aux_1}$ ciudades auxiliares **de entre todas las ciudades intermedias del MTSP**. El siguiente viajero, considerará $\mathbf{aux_2}$ ciudades auxiliares solo que, a diferencia del caso anterior, **no considera las ciudades intermedias ya asignadas a anteriores viajeros (primer viajero, en este caso)**. Se procederá de esta manera, resolviendo el TSP para cada viajero hasta llegar al último viajero, que se enfrente al problema del viajero simple en el cual **su número de ciudades auxiliares asignado coincide con el número de ciudades intermedias sobrantes (las que no se asignaron a viajeros anteriores en el reparto)**. Todo lo explicado se aprecia mucho mejor en el código, en el cual se define una lista con el número total de ciudades auxiliares de la que se van quitando aquellas que se van asignando a viajeros anteriores.

Una vez que se obtenga un resultado con la distancia optimizada de cada viajero, se observará cual es el viajero con mayor distancia recorrida, y se asignará al viajero con menor distancia en el recorrido esa ciudad que retrasó al otro viajero.

Se volverá a resolver el algoritmo con esta nueva asignación, e iremos procediendo de esta manera iterativamente hasta alcanzar un punto en el cual no se produzcan más mejoras.

Ciudades restringidas y auxiliares

En este caso, se consideran tanto ciudades restringidas para cada viajero como ciudades auxiliares a repartir.

El reparto de las ciudades auxiliares se llevará a cabo de la manera que se explicó en el apartado anterior.

Método de implementación en ROS: empleo de clases

Tal y como se hizo en el apartado en el que se explicó el TSP simple, a continuación se explicará de qué manera se ha implementado en el lenguaje C la solución de este problema de optimización recurriendo al empleo de clases.

Tan solo consideramos una clase, a la que se ha llamado **Mtsp**. En el próximo subapartado, explicaremos su significado, sus atributos asociados y los métodos que realizan modificaciones en los objetos de esta clase.

Todas las explicaciones serán introductorias, ya que se indagará con mayor profoundidad en las secciones del código en un apartado posterior de la memoria.

Mtsp

En el anexo [A.8] se presenta el código de la cabecera de la clase **Mtsp** con el atributo y los métodos que definen la clase.

- **ATRIBUTOS:**
  1. **min-distance:** Vector en el que se almacenarán las distancias mínimas obtenidas para cada viajero. El orden de las componentes se corresponderá por el orden de los viajeros definidos inicialmente en el código.
- **MÉTODOS:**
  1. **solve-only-res:** Tomando las ciudades intermedias y las ciudades origen y destino del problema, además de otros parámetros que se explicarán en el siguiente apartado, se trata de resolver n problemas TSP. La variable **res-ids** almacena una serie de vectores, donde cada uno de ellos almacena los ids de las ciudades por los que debe pasar un viajero determinado.
  2. **solve-only-aux:** También toma los mismos argumentos de entrada que el método anterior, reemplazando **res-ids** por **aux-ids**. El nuevo argumento del método hace alusión a los ids de las ciudades a repartir. Nótese que esta variable es un vector con los ids, mientras que **res-ids** era un vector de vectores enteros, ya que la **componente iésima** se correspondía con el **viajero iésimo** a resolver. En este propio método se hace el reparto, por lo que no existe una correspondencia inicial entre viajero e identificadores.
  3. **solve-aux-and-res:** Se mezclan ambos métodos anteriores para resolver el problema conjunto. Una vez que se asignan ciudades auxiliares con los ids almacenados en **res-ids**, me olvido de dicha variable y realizo la asignación con **aux-ids** de la misma manera en la que se procedió en el método **Mtsp::solve-only-aux**.

4. **solve-tsp:** Se recurre a este método dentro del código de los tres anteriores. Define un vector de viajeros (objetos de la clase **WholeGraph**). A cada componente de este vector (a cada viajero) se le asignan ciudades auxiliares y/o ciudades restringidas para resolver el problema del viajero con ellas. Una vez hecho esto, se vuelca en el vector **return-cities-ids** los ids de las ciudades por las que ha viajado cada viajero para obtener su mejor solución, considerando inicialmente todos los identificadores de las ciudades auxiliares por las que se puede pasar (**aux-ids-list**).

Fichero principal

Código

En el anexo [A.9] se incluye el código del fichero principal de extensión ".cpp", en el cual se ha incluido la librería del *MTSP* para resolver el problema del viajero, con el procedimiento explicado hasta ahora.

En él, se detalla de qué manera de ha llevado a cabo la declaración de las ciudades que se consideran en el problema del viajero. Así mismo, se detalla en el código asociado al hilo **tsp-thread** la declaración de los ids asociados a las ciudades auxiliares y a las ciudades restringidas, tomados en base a todas las ciudades declaradas previamente. Posteriormente, se resuelve el algoritmo con el método apropiado dentro de la **clase Mtsp**.

Visualización RVIZ

En este caso consideraremos un solo hilo, mediante el cual se trata de resolver el problema del *MTSP* con tres viajeros y únicamente ciudades restringidas. Como se señaló anteriormente, equivale a resolver tres *TSP* por separado.

Se define un nuevo método en la **clase Mtsp**, con el cual se devuelven los ids de las ciudades por las que pasa cada viajero. A dicho método lo hemos llamado **return-cit-ids**.

En el anexo [A.10] se definen, a nivel de código, las ciudades consideradas en esta visualización en *RVIZ*.

Por otra parte, en el anexo [A.11] se especifican las líneas de código con las cuales se devuelven los ids mencionados anteriormente por la clase **Mtsp**.

Ya que tenemos definidas las ciudades con sus coordenadas y sus respectivos identificadores al comienzo de este fichero principal, y que además conocemos los ids del resultado del algoritmo (volcados en la variable **def-track-points**), podemos visualizar en *RVIZ* fácilmente las ciudades por las que pasa cada viajero.

Por un lado, el código que se encarga de representar en *RVIZ* el resultado del algoritmo sería el mostrado en el anexo [A.12].

El resultado, en este caso, quedaría de la siguiente manera:

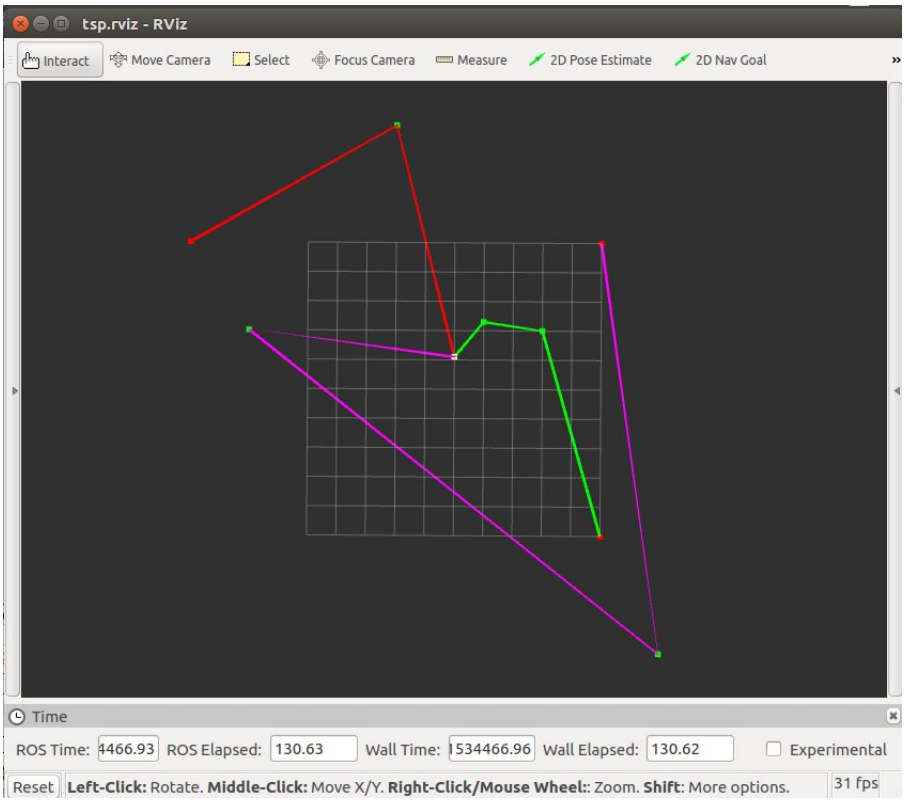

Figura 4.1 Resultado en *RVIZ* con tres viajeros.

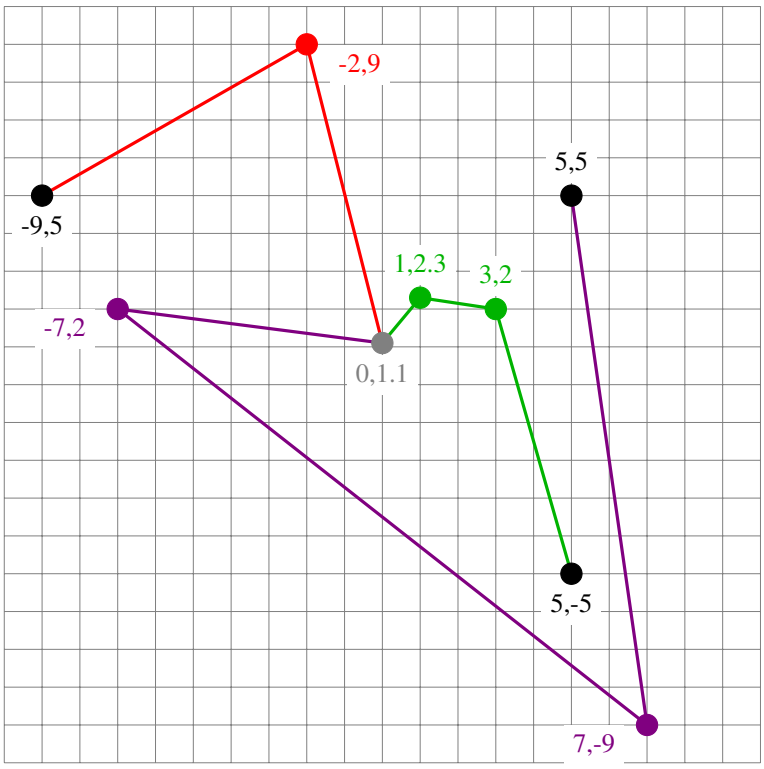

Figura 4.2 Coordenadas de las ciudades. Resultados *RVIZ*.

Se aprecia la representación de cada una de las rutas del viajero en un color distinto.

En la segunda imagen, se especifican las coordenadas de cada ciudad y las rutas de los tres viajeros, en colores diferentes. El punto gris se corresponde con la ciudad de origen del viaje (común para todos los

viajeros), mientras que los puntos negros representan las ciudades de destino de cada una de las rutas.

Por simplicidad en la representación del resultado, se ha supuesto que trabajamos en dos dimensiones, asignando un valor nulo a la tercera coordenada de las ciudades.

Resultados de los algoritmos

A continuación, se mostrarán ejemplos que se han puesto en el código principal tanto para *TSP* como para *MTSP*.

En el caso del *TSP*, se mostrarán dos ejemplos: uno en el que se resuelve el *Problema del Viajero* con un solo hilo de ejecución, y otro caso en el que se resuelve el mismo problema con cuatro hilos de ejecución.

Por otra parte, abordaremos el caso del *MTSP* con un solo hilo de ejecución. Se mostrarán los resultados de terminal para los tres casos: MTSP resuelto solo con ciudades restringidas, solo con ciudades auxiliares, o con ambas ciudades.

Además, para todos los casos, se enseñará por terminal la salida que se obtiene con los resultados más significativos.

TSP

En el apartado anterior, en el que se mostraba el código principal, aparecía una sección de código en la que se definían como vectores las ciudades por las que pasaría el viajero (o varios viajeros, en el caso de que nos enfrentaśemos al *MTSP*).

A lo largo de todo el apartado del *TSP*, consideraremos las mismas ciudades de origen y de destino, así como las siete ciudades intermedias que se detallan a continuación en el código. De dichas ciudades intermedias, tan solo usaremos algunas de ellas para resolver el *Problema del Viajero*, asignando los ids de ciudades restringidas.

Todas las ciudades consideradas se definen de la forma especiifacada en el anexo [A.13]:

A pesar de no comentarlo anteriormente, se define un hilo en el código en el cual hay un objeto de la clase *MTPS* con el cual se resuelve el problema de optimización. Se especifican, entre otros parámetros, ciudades restringidas del problema, número de iteraciones del algoritmo, etc.

El código que define el hilo se encuentra en el anexo [A.14].

Señalemos que el mutex que toma el método **Mtsp::solve-only-res** como argumento de entrada se emplea para que la salida por pantalla de los resultados de los distintos hilos no se mezclen, y se puedan ver por separado en el terminal a la salida.

Tsp con un solo hilo (Graph)

Veamos la salida por terminal en el caso de que, con los ids de las ciudades restringidas especificados anteriormente, se resuelva el *TSP* simple.

En lo referente al código, las líneas especificadas en el anexo [A.15] tienen asociadas la salida por terminala anterior.

Comentemos los resultados más relevantes:

- El tiempo de ejecución del algoritmo es de unos 12 milisegundos. Más adelante, compararemos con el caso multihilo para apreciar mejor la mejora que se verá posteriormente.
- Ya que hemos considerado cuatro ciudades intermedias en el problema, se observa a la salida por terminal el número de posibles ramas: 4! = 24.

```
WHOLE GRAPH

EXECUTION TIME: 0.011749388 seconds
Possible branches: 24
Number of branches desired: 500
Number of branches obtained: 24

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  2    <----> [1  2.3  -4]     (4.29418) uds
  3    <----> [3  2  3]        (7.28629) uds
  4    <----> [-2  8.4  4]     (8.18291) uds
  7    <----> [7  9  2]        (9.23905) uds
  8    <----> [10  10  10]     (8.60233) uds

Min distance: 37.6048  (SUCCESS)
Min distance branch (id): 12
Number of intermediate cities: 4
```

Figura 5.1 TSP: un solo hilo.

- Hemos especificado, como número deseado de soluciones, 500. Es decir, que el algoritmo deberá ser capaz de obtener 500 ramas del grafo que se está considerando. Si hay menos soluciones, como ocurre en este caso, se obtendrá el máximo número de soluciones posible (24 en este caso).
- Se muestran las ciudades en el orden en el que el viajero las recorre. La primera ciudad que se muesta es el origen, mientras que la última es el destino. Se van mostrando a la derecha de cada ciudad la distancia que se acumula con respecto la ciudad anterior, en unidades.
- Después, se muestra la distancia asociada a la mejor solución encontrada, el id de la rama con dicha solución, y por último el número de ciudades intermedias consideradas en el problema.

Con el fin de visualizar con mayor claridad la solución, a continuación se muestra el árbol de nodos que representa la solución que se considera.

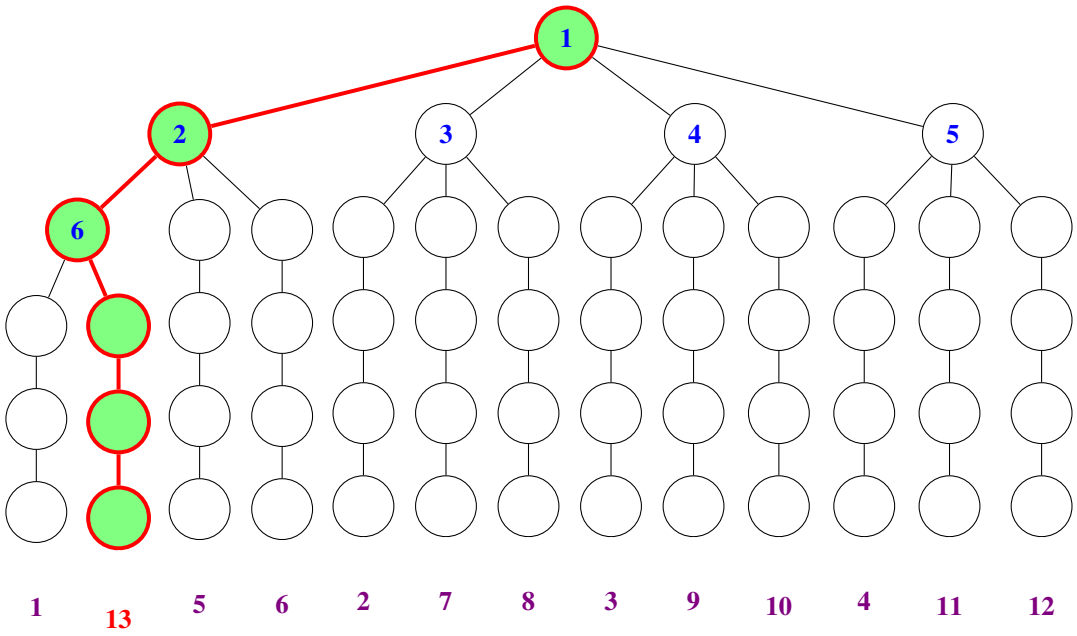

Figura 5.2 Solución del *TSP* en el caso de un solo hilo.

Los nodos que aparecen en el esquema anterior conforman las ramas que definen el árbol de nodos hasta ese momento. Aparece destacada la rama que tiene asociada la solución mostrada por terminal anteriormente (la mejor de todas). En violeta, aparecen numeradas las ramas completas del grafo, en el orden en el que se crean hasta llegar a la solución final. Aunque el árbol se siga desarrollando hasta obtener todas las soluciones, se ha mostrado el esquema anterior con el fin de ilustrar de qué manera se va desglosando hasta llegar a nuestra solución. De hecho, en azul están numerados los nodos que se han desglosado hasta ese momento.

Debido a que, conforme lo hemos definido, el id asociado a la primera rama del grafo es 0, la mejor solución obtenida ha sido la de la rama número 13.

Tsp con cuatro hilos (WholeGraph)

Resolveremos el mismo problema, pero en esta ocasión empleando cuatro hilos de ejecución.

El código se encuentra expuesto en el anexo [A.16].

A continuación, se muestra la salida por pantalla de cada una de las subramas del árbol completo. Se recuerda que se hizo uso de un *mutex* con el fin de impedir que se solapasen las salidas por terminal de los distintos hilos.

```
BRANCH NUMBER 1

EXECUTION TIME : 0.002069317 seconds
Possible branches: 6
Number of branches desired: 500
Number of branches obtained: 6

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  2    <----> [1  2.3  -4]     (4.29418) uds
  3    <----> [3  2  3]        (7.28629) uds
  4    <----> [-2  8.4  4]     (8.18291) uds
  7    <----> [7  9  2]        (9.23905) uds
  8    <----> [10  10  10]     (8.60233) uds

Min distance: 37.6048  (SUCCESS)
Min distance from little branch 1 (id): 3
Number of intermediate cities: 4
```

```
BRANCH NUMBER 2

EXECUTION TIME : 0.002366548 seconds
Possible branches: 6
Number of branches desired: 500
Number of branches obtained: 6

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  3    <----> [3  2  3]        (4.33705) uds
  2    <----> [1  2.3  -4]     (7.28629) uds
  4    <----> [-2  8.4  4]     (10.4981) uds
  7    <----> [7  9  2]        (9.23905) uds
  8    <----> [10  10  10]     (8.60233) uds

Min distance: 39.9628  (SUCCESS)
Min distance from little branch 2 (id): 0
Number of intermediate cities: 4
```

```
BRANCH NUMBER 3

EXECUTION TIME : 0.002354867 seconds
Possible branches: 6
Number of branches desired: 500
Number of branches obtained: 6

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  4    <----> [-2  8.4  4]     (8.56096) uds
  2    <----> [1  2.3  -4]     (10.4981) uds
  3    <----> [3  2  3]        (7.28629) uds
  7    <----> [7  9  2]        (8.12404) uds
  8    <----> [10  10  10]     (8.60233) uds

Min distance: 43.0717   (DIFFERENCE: 3.8147e-06)
Min distance from little branch 3 (id): 2
Number of intermediate cities: 4
```

```
BRANCH NUMBER 4

EXECUTION TIME : 0.002443384 seconds
Possible branches: 6
Number of branches desired: 500
Number of branches obtained: 6

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  7    <----> [7  9  2]        (10.7429) uds
  3    <----> [3  2  3]        (8.12404) uds
  2    <----> [1  2.3  -4]     (7.28629) uds
  4    <----> [-2  8.4  4]     (10.4981) uds
  8    <----> [10  10  10]     (13.5115) uds

Min distance: 50.1628  (SUCCESS)
Min distance from little branch 4 (id): 0
Number of intermediate cities: 4
```

Figura 5.3 TSP: cuatro hilos.

Se puede apreciar que, con respecto al caso en el que se resolvía el TSP empleando un solo hilo para ello, el tiempo de ejecución mejora en esta ocasión; se han requerido unos 2 ms para generar las 24 soluciones que se pueden obtener con cuatro ciudades intermedias (4! = 24) mientras que antes hacían falta unos 15 ms.

Aunque la mejora no es demasiado notable en este caso, sí que lo puede ser en el caso en el que manejemos un número de ciudades intermedias bastante mayor y, con ello, un número de soluciones posibles muy elevado. En esa ocasión, el empleo de esta técnica para explorar ramas determinadas de un grafo (que puede considerarse bastante extenso) puede facilitar la tarea de encontrar buenas soluciones en menos tiempo.

A continuación, se muestra en el grafo del viajero la solución de cada una de las cuatro subramas.

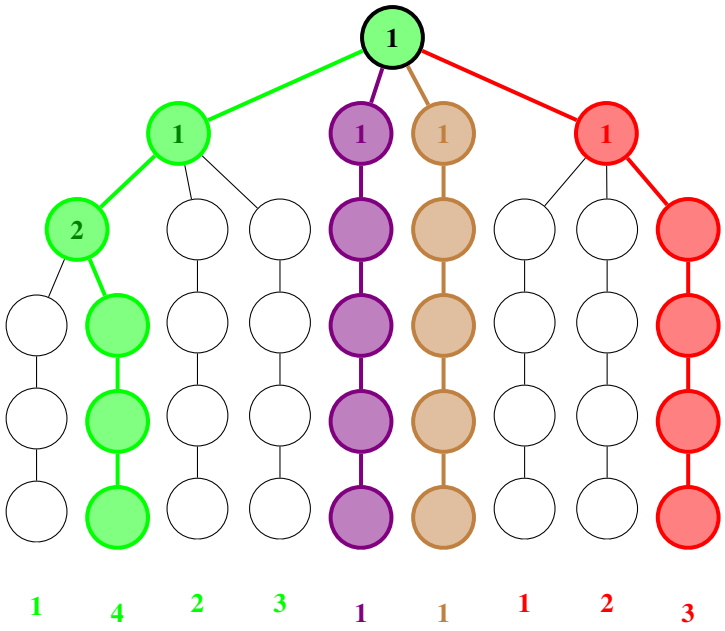

Figura 5.4 Solución del *TSP* en el caso multihilo.

En esta ocasión, se considera cada rama por separado como bien se explicó con anterioridad. El costo computacional es mucho menor, ya que hay cuatro hilos que tratan de encontrar 3! = 6 soluciones para cada una de las subramas, ya que se consideran tres ciudades intermedias tras haber viajado a la primera ciudad con la que comienza la subrama.

Al igual que en el caso para un solo hilo que se mostró previamente, se han mostrado tanto la enumeración de los nodos considerando el orden en el que se desglosan (para cada subrama) como el orden en el que se obtienen ramas completas hasta llegar a aquella que define la mejor solución.

Si observamos tanto el grafo como la salida por terminal, vemos que la mejor solución se corresponde con aquella que se obtiene para la primera subrama del árbol, destacada en verde. La misma solución que en el caso de empleo de un solo hilo, solo que, en lugar de 12 milisegundos, han hecho falta tan solo unos 2 milisegundos para hallar la solución sin dejar de explorar otras partes del grafo.

Como también se aclaró para el otro esquema del *TSP* para un hilo, el grafo se recorre por completo por las 500 iteraciones especificadas antes de comenzar el algoritmo. Se ha mostrado de esta forma para ilustrar cómo van surgiendo ramas conforme avanzan las iteraciones en cada hilo, al menos hasta llegar a la mejor solución.

MTSP

En el anexo [A.17] se muestra el código modificado para el caso del *MTSP*. En esta ocasión, se consideran varios destinos y, por tanto y como define el propio problema múltiple, varios viajeros.

Por simplicidad, se considerará un solo hilo en el apartado de resultados del *MTSP*. Presentaremos las variantes del *Multiple Travelling Salesman Problem* y sus respectivos resultados.

Solo ciudades auxiliares

El código del hilo de ejecución tiene la forma que se detalla en el anexo [A.18].

Considerando dos viajeros con tres ciudades intermedias a repartir, se muestran los siguientes resultados por terminal:

```
Traveler number 1:

WHOLE GRAPH

EXECUTION TIME: 0.001274603 seconds
Possible branches: 6
Number of branches desired: 500
Number of branches obtained: 6

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  3    <----> [3  2  3]        (4.33705) uds
  4    <----> [-2  8.4  4]     (8.18291) uds
  9    <----> [10  10  10]     (13.5115) uds

Min distance: 26.0314  (SUCCESS)
Min distance branch (id): 0
Number of intermediate cities: 2
```

```
Traveler number 2:

WHOLE GRAPH

EXECUTION TIME: 0.000092488 seconds
Possible branches: 1
Number of branches desired: 500
Number of branches obtained: 1

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  6    <----> [2  -9  9]       (13.6752) uds
  9    <----> [1.5  -2  1]     (10.6419) uds

Min distance: 24.3171  (SUCCESS)
Min distance branch (id): 0
Number of intermediate cities: 1
```

Figura 5.5 MTSP: primera iteración para *ciudades auxiliares*.

```
max_value: 1e+09
*it_dis: 26.0314

(GO ON) MAX ID VALUE (0): 26.0314
```

Figura 5.6 MTSP: resultados de primera iteración (*ciudades auxiliares*).

```
Traveler number 1:

WHOLE GRAPH

EXECUTION TIME: 0.000324926 seconds
Possible branches: 3
Number of branches desired: 500
Number of branches obtained: 3

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  3    <----> [3  2  3]        (4.33705) uds
  9    <----> [10  10  10]     (12.7279) uds

Min distance: 17.065  (SUCCESS)
Min distance branch (id): 0
Number of intermediate cities: 1
```

```
Traveler number 2:

WHOLE GRAPH

EXECUTION TIME: 0.000276011 seconds
Possible branches: 2
Number of branches desired: 500
Number of branches obtained: 2

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  4    <----> [-2  8.4  4]     (8.56096) uds
  6    <----> [2  -9  9]       (18.5408) uds
  9    <----> [1.5  -2  1]     (10.6419) uds

Min distance: 37.7436  (SUCCESS)
Min distance branch (id): 0
Number of intermediate cities: 2
```

Figura 5.7 MTSP: segunda iteración para *ciudades auxiliares*.

```
max_value: 26.0314
*it_dis: 37.7436

(FINISHED) MAX ID VALUE: 26.0314
```

Figura 5.8 MTSP: resultados de segunda iteración (*ciudades auxiliares*).

Analicemos brevemente los resultados obtenidos.

Como se comentó anteriormente, el caso del *MTSP* con ciudades auxiliares era el más complejo de todos. Se debe a que hay que barajar posibles mejoras en el caso de que la solución obtenida tenga una ruta demasiado larga, para uno de los viajeros que se consideran.

De esta manera, para los resultados obtenidos por terminal:

- En la **primera iteración**, se observa cómo el primer viajero tiene las ciudades 3 y 4, dejando al otro viajero con la ciudad número 6. Se almacena el valor de la ruta más larga de entre las dos obtenidas (en este caso es la del primer viajero, de de unas 26 unidades).
- Cuando considederamos la mejora en la **segunda iteración**, el primer viajero cede al segundo la ciudad número 6. De esta manera, observando qué distancia obtiene cada viajero en su recorrido, vemos que

```
previous_cities_ids DATA:
Traveler number 1: 3  4
Traveler number 2: 6
```

Figura 5.9 MTSP: resultados globales (*ciudades auxiliares*).

el resultado empeora bastante con respecto a la primera iteración. Ahora el segundo viajero tiene un coste de casi 38 unidades en su recorrido.

- Viendo esto, el algoritmo concluye que la mejor solución es la obtenida en la primera iteración. Se observa en la última figura por terminal qué ids de ciudades auxiliares se corresponden con este reparto, que era el inicial.

Solo ciudades restringidas

El código del hilo de ejecución tiene la forma que se detalla en el anexo [A.19]. La única modificación con respecto al apartado anterior está en los ids que se le proporcionan al viajero para resolver el problema de optimización. En este caso, sólo son ids de ciudades restringidas, por lo que este problema equivale a resolver dos *TSP* simples.

La salida por teminal es la siguiente, para cada viajero.

```
WHOLE GRAPH

EXECUTION TIME: 0.001905447 seconds
Possible branches: 6
Number of branches desired: 500
Number of branches obtained: 6

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  6    <----> [2  -9  9]       (13.6752) uds
  3    <----> [3  2  3]        (12.5698) uds
  4    <----> [-2  8.4  4]     (8.18291) uds
  9    <----> [10  10  10]     (13.5115) uds

Min distance: 47.9393  (SUCCESS)
Min distance branch (id): 2
Number of intermediate cities: 3
```

```
WHOLE GRAPH

EXECUTION TIME: 0.000291383 seconds
Possible branches: 2
Number of branches desired: 500
Number of branches obtained: 2

Cities from route and distances (ordered):
  1    <----> [0  1.1  0]
  2    <----> [1  2.3  -4]     (4.29418) uds
  5    <----> [7  9  9]        (15.8079) uds
  9    <----> [1.5  -2  1]     (14.6714) uds

Min distance: 34.7735  (SUCCESS)
Min distance branch (id): 0
Number of intermediate cities: 2
```

Figura 5.10 MTSP: resultados con *ciudades restringidas*.

No hay conflicto alguno en repartición de ciudades intermedias, ya que las ciudades restringdas están asignadas de antemano a cada viajero.

Ciudades restringidas y ciudades auliliares.

El código del hilo de ejecución tiene la forma que se detalla en el anexo [A.20]. El algoritmo, en este caso, toma tanto identificadores de ciudades restringidas como de ciudades auxiliares.

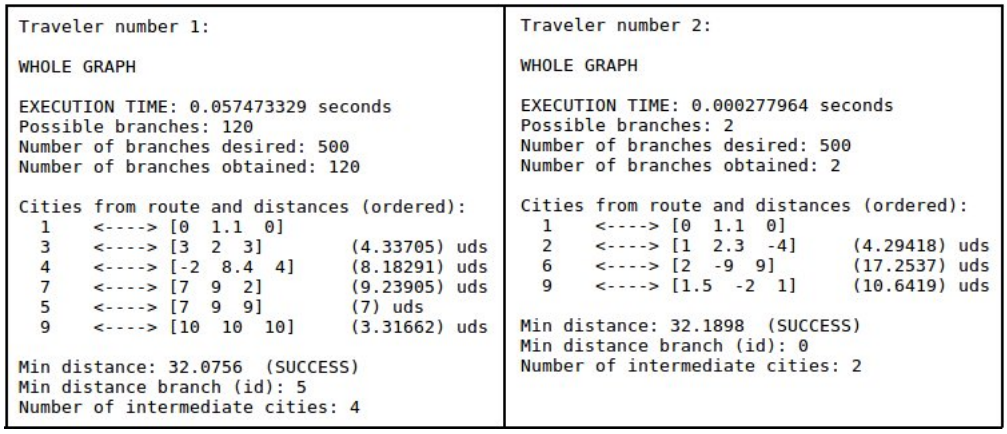

Figura 5.11 MTSP: primera iteración para *ciudades auxiliares y restringidas*.

Como salida por terminal obtenemos, para primera y segunda iteración:

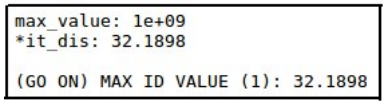

Figura 5.12 MTSP: resultados de primera iteración (*ciudades auxiliares y restringidas*).

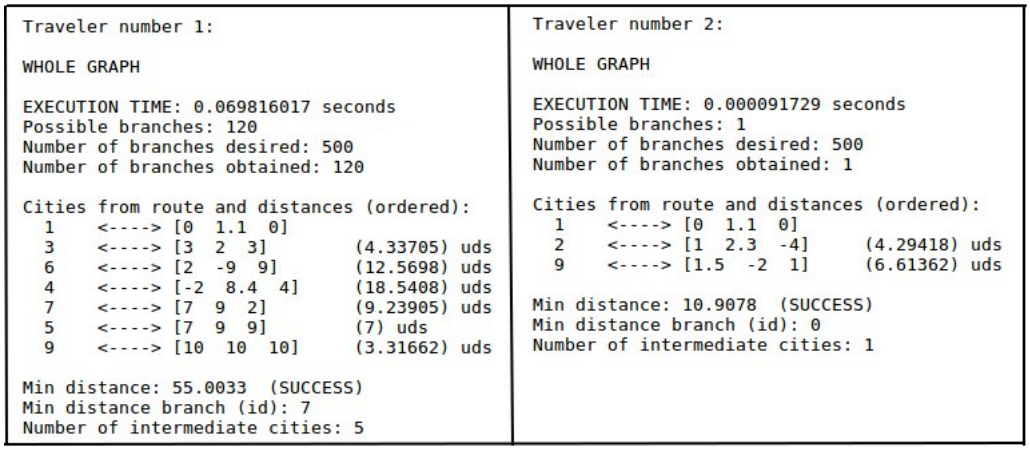

Figura 5.13 MTSP: segunda iteración para *ciudades auxiliares y restringidas*.

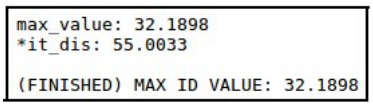

Figura 5.14 MTSP: resultados de segunda iteración (*ciudades auxiliares y restringidas*).

```
previous_cities_ids DATA:
Traveler number 1: 3  4  7  5
Traveler number 2: 2  6
```

Figura 5.15 MTSP: resultados globales (*ciudades auxiliares y restringidas*).

Se procede de manera análoga al caso en el que sólo hay ciudades auxiliares, con la única diferencia de que las ciudades restringidas no se consideran en el reparto entre iteraciones para mejorar la solución.

Métodos implementados en el algoritmo

A lo largo de este apartado, iremos presentando los métodos asociados a cada una de las clases con las que se ha implementado el *Problema del Viajero* en el lenguaje de programación C.

Llevaremos a cabo una descripción más exhaustiva del código que la que hemos hecho en los apartados anteriores, en los cuales tan sólo presentábamos las clases y mencionábamos los métodos que se definían.

Clase "City"

fill

La sección de código correspondiente aparece referenciada en el anexo [A.21].

Con este método, asignamos valores a los atributos de la clase **City**.

get-string-data

La sección de código correspondiente aparece referenciada en el anexo [A.22].

Devuelve una cadena con dos componentes; en la primera de ellas, el id de la ciudad considerada, miestras que en el segundo elemento aparecen las coordenadas de la ciudad en cuestión.

Clase "Node"

fill

La sección de código correspondiente aparece referenciada en el anexo [A.23].

Con este método, asignamos valores a los atributos de la clase **Node**.

convert-element

La sección de código correspondiente aparece referenciada en el anexo [A.24].

Dado un objeto del tipo **Element** como argumento del método, volcamos en los atributos del nodo considerado los campos del elemento de entrada.

Este método se encarga de inicializar un nodo a partir de un objeto del tipo **Element**.

Clase "Path"

fill

La sección de código correspondiente aparece referenciada en el anexo [A.25].

Con este método, asignamos valores a los atributos de la clase **Node**.

Clase "Branch"

add-element

La sección de código correspondiente aparece referenciada en el anexo [A.26].

Añade un elemento a la rama considerada. Es decir, volcamos en el componente último del atributo **elements** el elemento a añadir, considerando lo siguiente:

- El primer elemento de **elements** (la rama considerada) siempre debe ser un nodo.
- Los elementos añadidos han de ser del tipo opuesto al del último elemento añadido al atributo. Es decir, elementos del tipo **Node** y del tipo **Path** deben ir alternándose conforme se añaden a la rama.

get-total-distance

La sección de código correspondiente aparece referenciada en el anexo [A.27].

Recorre todos los elementos de la rama considerada, con el fin de acumular las distancias de cada uno de los caminos que ha tomado el viajero. Para ello, en el bucle de iteración, solo se consideran los elementos que se corresponden con caminos.

get-total-distance-destiny-id

La sección de código correspondiente aparece referenciada en el anexo [A.28].

Devuelve la distancia acumulada en los caminos de la rama, **hasta llegar al id especificado como argumento del método**. Hay que señalar que se empiezan a acumular distancias de los viajes desde la ciudad origen, hasta llegar al elemento de la clase **Node** con el id especificado.

get-cities

La sección de código correspondiente aparece referenciada en el anexo [A.29].

Devuelve un vector de tipo **City** con las ciudades por las que el viajero ha pasado, de la rama considerada. Para lograrlo, se consideran los elementos **Node** y se toma su atributo asociado a la clase **City**.

search-node

La sección de código correspondiente aparece referenciada en el anexo [A.30].

Devuelve **true** o **false** en función de si el nodo que toma el método como entrada se encuentra en la rama considerada. Este método resulta bastante útil si se desea encontrar un nodo determinado en el árbol completo; basta con aplicarlo a todas las ramas que almacene el árbol.

search-path

La sección de código correspondiente aparece referenciada en el anexo [A.31].

Devuelve **true** o **false** en función de si el camino que toma el método como entrada se encuentra en la rama considerada. Como comentamos para el método anterior, este método resulta bastante útil si se desea encontrar un camino determinado en el árbol completo; basta con aplicarlo a todas las ramas que se encuentren almacenadas en el árbol.

get-route

La sección de código correspondiente aparece referenciada en el anexo [A.32].

Devuelve un vector con los nodos de la rama que forman parte de la ruta **hasta llegar al nodo n, que es el argumento de entrada del método**. En este método, se emplea **Node::convert-element** para convertir los elementos nodos de la rama a objetos de la clase **Node**.

Clase "Graph"

Graph (constructor)

La sección de código correspondiente aparece referenciada en el anexo [A.33].

Se explicará por orden las secciones de código empleadas para definir el constructor:

- Almacenamos en **cities** (atributo de la clase Graph) todas las ciudades del viaje completo del viajero, incluyendo origen y destino.
- Se inicializa la base de datos con el nodo padre.
- Se inializan los punteros del viajero y de desglose (**traveler** y **break-down**, respectivamente).
- Se crea la primera rama del grafo.
- Se ejecuta el algoritmo principal en este mismo constructor (**main-algorithm**).

explore-and-order

La sección de código correspondiente aparece referenciada en el anexo [A.34].

Por sencillez, se irán mencionando las secciones del código que se presenta, con su número de línea correspondiente, para hacer alusión a cada una de las partes que se vaya comentando a continuación:

- **Líneas 2-8**: Se definen, en este orden:
  - **res:** Salida que almacenará los nodos y los caminos obtenidos en este método.
  - **route:** Vector que almacena los nodos por los que se ha pasado hasta llegar hasta donde se encuentra el viajero en este momento.
  - El resto de variables se irán comentando posteriormente, conforme se vayan empleando en el método.
- **Líneas 10-13**: Se define una lista en la que se vuelcan las ciudades almacenadas en el grafo. Se emplea una lista como variable local de este método por mayor facilidad en eliminar elementos determinados durante su uso.
- **Líneas 15-17**: En la lista de ciudades que se acaba de definir (**list-of-cities**), descartamos aquellas por las que el viajero ya ha pasado.
- **Líneas 21-29**: Se crean tanto nodos como caminos (almacenados, respectivamente, en **nodes-created** y **paths-created**) con la información asociada a la lista de ciudades definida anteriormente.
- **Líneas 33-50**: Mientras quede algún nodo o camino almacenado en sus respectivas listas (**nodes-created** y **paths-created**), se llevan a cabo las siguientes instrucciones:
  - Buscamos el camino de la lista con la distancia mínima, de entre todos los caminos creados. Lo almacenamos en el iterador **path-it**.
  - Posteriormente, se busca el nodo que tenga la misma etiqueta que la etiqueta origen del camino almacenado en **path-it**.
  - Añadimos a **nodes-right** y **paths-right** los objetos ordenados por menor distancia. Eliminamos de las listas desordenadas los objetos que acabamos de añadir a las listas ordenadas, con el fin de no volver a considerarlos en posteriores iteraciones del bucle **while**.
- **Líneas 52-54**: Devolvemos **res**, con los caminos y nodos ordenados.

distance-between-nodes

La sección de código correspondiente aparece referenciada en el anexo [A.35].

Dados dos nodos a la entrada, se calcula y devuelve a la salida el módulo de la distancia entre sus ciudades almacenadas.

travel-nearest-city

La sección de código correspondiente aparece referenciada en el anexo [A.36].

Por sencillez, se irán mencionando las secciones del código que se presenta, con su número de línea correspondiente, para hacer alusión a cada una de las partes que se vaya comentando a continuación:

- **Línea 3**: Se definen dos listas en las que se vuelcan los nodos y caminos obtenidos del desglose del nodo del viajero, realizado con el método **Graph::explore-and-order**.
- **Línea 8**: Añadimos el nodo viajero a la rama del grafo (su atributo **branches**).

- **Líneas 11-13**: Buscamos el mínimo camino que parta del nodo en el que se encuentra el viajero (al que apunta **traveler**) y lo almacenamos en **p-it**.
- **Línea 16**: Añadimos dicho camino a la rama del grafo.
- **Líneas 19-20**: Buscamos el nodo con el mismo id que el atributo **id-destiny** del camino mínimo que hemos encontrado anteriormente, y lo guardamos en el iterador **n-it**.
- **Líneas 24-25**: Se realiza una búsqueda en la base de datos de nodos del grafo (**nodes**). Buscamos el nodo almacenado en **n-it**, y modificamos la dirección de memoria del puntero **traveler** por la de este nuevo nodo.
- **Líneas 28**: Añadimos a la rama del grafo este último nodo.

A modo de resumen, observamos que con este método actualizamos la dirección de memoria del puntero viajero conforme se va viajando a la siguiente ciudad. Además, se van insertando en la rama del grafo los elementos correspondientes que van definiendo el viaje hasta el momento.

Cabe señalar que, ya que empleamos el método **Branch::add-element**, en caso de que queramos insertar el nodo del viajero de la siguiente iteración una vez que acabamos de insertar el mismo nodo en la iteración anterior, no podremos hacerlo, y el siguiente elemento insertado en la rama será el próximo camino.

refresh-graph-data

La sección de código correspondiente aparece referenciada en el anexo [A.37].

En este método, se llevan a cabo las siguientes modificaciones en el árbol:

- Se modifica la dirección de memoria del puntero **move-bd** en función de si decidimos hacerlo o no durante la ejecución de **Graph::main-algorithm**. Una vez que epliquemos el algoritmo principal en este mismo apartado, se comprenderá mejor en qué caso cambiamos el puntero de desglose.
- Actualizamos la dirección de memoria del puntero del viajero, **traveler**.
- Creamos una nueva rama en el árbol, con el fin de llenar sus elementos en las iteraciones posteriores de **Graph::main-algorithm**.
- Reseteamos las bases de datos, tanto **paths** como **nodes**.
- Insertamos el primer nodo en la base de datos; será el nodo al que apunta el puntero del viajero.
- Tanto **break-down** como **traveler** apuntan a dicho nodo que se ha insertado en la base de datos.

remove-branches-elements

La sección de código correspondiente aparece referenciada en el anexo [A.38].

Este método se aplica justo después de **Graph::explore-and-order**. Su cometido principal consiste en eliminar de los elementos de entrada (nodos y caminos) aquellos objetos que ya formen parte de otras ramas creadas hasta ahora.

¿Por qué es importante este método? Porque todas las ramas nuevas que se vayan creando deben ser distintas a las creadas anteriormente, ya que que vamos buscando soluciones nuevas constantemente.

En el caso de que todavía no se hayan creado ramas anteriormente, este método no modifica de manera alguna el atributo que almacena las ramas del grafo; **branches**.

Por sencillez, se irán mencionando las secciones del código que se presenta, con su número de línea correspondiente, para hacer alusión a cada una de las partes que se vaya comentando a continuación:

- **Líneas 11-15**: Se eliminan los **nodos** que formen parte de otras ramas anteriores del grafo.
- **Líneas 18-22**: Se eliminan los **caminos** que formen parte de otras ramas anteriores del grafo.
- **Líneas 25-27**: Se actualiza la base de datos de los nodos.
- **Líneas 30-32**: Se actualiza la base de datos de los caminos.
- **Líneas 34**: Devuelvo los nodos y los caminos almacenados en la variable local **ret-elements**.

break-down-son

La sección de código correspondiente aparece referenciada en el anexo [A.39].

Con este método, devolvemos el identificador adecuado para seguir etiquetando a los nodos que se seguirán creando en el grafo.

En primer lugar, buscamos la rama en la que se encuentre el nodo al que apunta **break-down**. Una vez que la hayamos encontrado, guardamos en el iterador **e-it** el elemento que se corresponde con el nodo al que apunta el puntero de desglose.

Hay que tener en cuenta que, considerando el orden en el que se han ido defieniendo las ramas del grafo, se asegura que este método devolverá el hijo del nodo de desglose con el camino de la menor distancia.

En caso de que no se haya encontrado el nodo con la dirección de **break-down**, el método devuelve un **-1**.

create-branch

La sección de código correspondiente aparece referenciada en el anexo [A.40].

Llegamos al método que crea las ramas del árbol. Analizaremos paso a paso las secciones del código y comentaremos el cometido de las variables más importantes que se declaran al comienzo.

Antes de ello, es importante resaltar lo siguiente. Las ramas del grafo almacenadas en *branches* **no tienen por qué abarcar todo el recorrido del viajero**.

El primer nodo de la primera rama almacenará la ciudad de origen del viaje, y el último nodo de dicha rama almacenará la ciudad de destino. Sin embargo, conforme vamos creando mas y más ramas, su longitud disminuye, ya que el primer nodo de las mismas tienen asignado un nivel más profundo en el grafo.

Una imagen que intenta ilustrar esta explicación puede tener la siguiente forma:

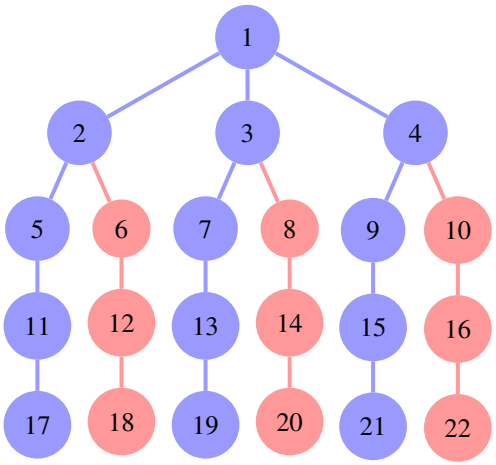

Figura 6.1 Subramas en el caso de tres ciudades intermedias.

En cuanto a la imagen anterior, las tres ramas azules que parten del nodo padre del grafo son las tres primeras ramas creadas en el grafo. Estamos desglosando el *NODO 1*, por lo que estamos obteniendo ramas que parten de dicho nodo hasta la ciudad de destino (nodos 17,19 y 21, respectivamente).

Una vez desglosado el *NODO 1*, tratamos de hacer lo mismo con el *NODO 2*. Como la primera rama pasa por el *NODO 2*, cuando tratemos de desglosar dicho nodo tan solo obtendremos una rama. Dicha rama tendrá como nodo final el *NODO 18*, y partirá del *NODO 2*. Es una de las ramas que aparece representada en rojo, más corta que las tres anteriores.

En el caso en el que nos encontremos ante un grafo con más ciudades intermedias, los desgloses de los nodos irán asociados con una creación de ramas cada vez más cortas. Esto se traduce en una menor carga de información almacenada por el programa (lo cual es una ventaja).

Comencemos a abordar el código que se presenta a continuación:

- **El código se compone por un bucle do-while**, sujeto a una condición que depende del cumplimiento de una expresión booleana, llamada **c**. Esta condición estará definida de una manera u otra según se aborde el problema del TSP simple o el problema con múltiples viajeros y múltiples ciudades destino; el MTSP. En concreto, dependerá de una variable externa a este método, **threshold**.

Dicha variable define el nivel límite hasta el cual queremos que las ramas del grafo almacenen elementos. Dicho de otra manera, con **threshold** decidimos cuántas ciudades intermedias queremos incluir en el problema.

En función de esto, hay una condición **if-else** dentro de este bucle que distingue dos casos:

- Añado la ciudad de destino a la rama y la doy por finalizada (**líneas 11-24**). En este caso, el viajero apunta al nodo con el nivel denotado por **threshold**.
- Se van creando las ramas en el caso de que el viajero no haya llegado a **threshold** (**líneas 25-42**).

Dicho esto comentaremos ambas condiciones.

- **Líneas 11-24**:
  1. Creamos el nodo **n** que almacene la ciudad destino del viajero; lo incluimos en la base de datos **nodes** y hacemos que el puntero **traveler** apunte a dicho nodo de la base de datos.
  2. Del mismo modo, creamos el camino **p** que crea el tramo que tiene como **destiny-id** la etiqueta asociada al nuevo nodo del viajero.
  3. Añadimos el camino **p** y el nodo **n** al la rama actual.
  4. **Definimos la condición c: el bucle while se ejecuta mientras el número de elementos de la rama actual sea menor al tamaño deseado de la rama, teniéndose en cuenta desde dónde se empieza el desglose (*break-down*) y el valor de *threshold*.**
- **Líneas 25-42**:
  1. En primer lugar, refrescamos la variable **global-node-id** en función del nodo al que apunte **break-down**. En el caso de que apunte al nodo del viajero, el **global-node-id** con el que se etiquetan los nuevos nodos toma el valor del id del hijo de **break-down**, como se comentó en el método **Graph::break-down-son**. En otro caso, se toma el id con el que se etiquetó al último nodo para seguir etiquetando al resto.

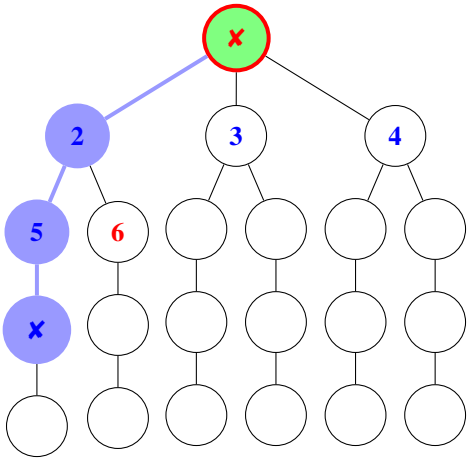

Figura 6.2 Numeración de ramas: viajero y desglose en distintos nodos.

En la figura anterior, se ilustra el caso en el cual el puntero de desglose (marca roja) tiene una dirección distinta a la del puntero del viajero (marca azul). Por ello, a la hora de hacer el próximo desglose, se tomará el último identificador que se ha asignado a un nodo en la base de datos, en este caso el número 6 (marcado en rojo). De esta manera, el nodo actual al que apunta el viajero se numera con un 7 y se sigue incrementando el indicador hasta completar la rama actual.

Hay que señalar que en el grafo representado tan solo se han creado los nodos numerados (y los caminos que unen los nodos existentes entre sí). Aquellos que no aparecen sombreados y que están numerados, se corresponden con los nodos que **solo pertenecen a la base de datos de la rama actual**. Por otro lado, los nodos que aparecen sombreados pertenecen a la rama actual (y también a los de la base de datos, que se crearon antes). Para los caminos, se aplica el mismo razonamiento.

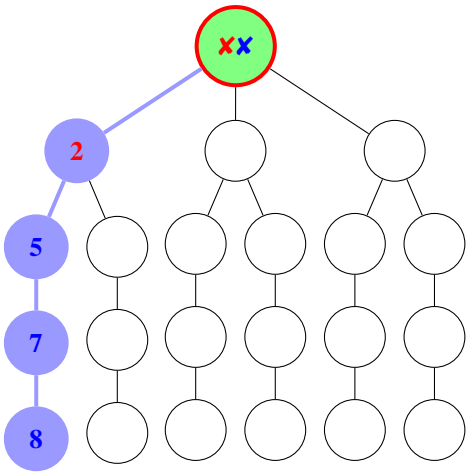

Figura 6.3 Numeración de ramas: viajero y desglose en el mismo nodo.

En la segunda imagen del grafo, se contempla el caso en el que el viajero y el desglose apuntan al mismo nodo. Por ello, se toma como último identificador aquel que se corresponde con el primer nodo hijo del nodo de desglose actual. Se procede de esta manera porque hay que crear una nueva base de datos, ya que se empieza a crear una nueva rama que parte desde el nodo padre del grafo. De esta manera, se numeran los nodos del segundo nivel de manera idéntica con respecto a la imagen anterior, y los nodos adquieren un identificador unívoco en el grafo.

2. Creo caminos y nodos desglosando el nodo del viajero actual. En caso de que no obtenga elementos en **elements-created**, significa que que he llegado al final de la rama y que debo cambiar la dirección del puntero de desglose **break-down**.
3. Actualizamos el id de numeración de los nodos, en función de la consideración previa que se ha llevado a cabo con respecto a los hijos de **break-down**.
4. **Definimos la condición c: el bucle while se ejecuta mientras el número de elementos de la rama actual sea menor al tamaño deseado de la rama, teniéndose en cuenta desde dónde se empieza el desglose (*break-down*)**.

move-break-down

La sección de código correspondiente aparece referenciada en el anexo [A.41].

Actualizamos el valor del puntero de desglose; apuntará al nodo con su identificador una unidad mayor que el nodo de desglose actual.

print-results

No es necesario mostrar código; imprime los valores más significativos que resultan de ejecutar el algoritmo del TSP.

get-total-distance

La sección de código correspondiente aparece referenciada en el anexo [A.42].

Devuelve la distancia total de la rama del grafo que tenga el identificador **branch-id**.

Comentaremos las secciones más reseñables del código:

- **Líneas 7-13**: Se definen los condiciones booleanas **minor** y **equal**, las cuales se emplearán más adelante en el propio código.

  1. **Líneas 7-10**: En caso de que el límite **threshold** no tome el valor del último nivel del grafo, se define las condiciones **equal** y **minor** como verdadera siempre y cuando el número de elementos de la rama actual sea menor al deseado **considerando el threshold**.
  2. **Líneas 11-13**: En caso de que no se considere **threshold** (y, por tanto, su valor coincidiese con el último nivel del grafo) se definen **minor** y **equal sin considerarse el valor del threshold**.

- **Líneas 19-32**: En caso de que la rama tenga elementos y de que se cumpla **minor**, añadimos al acumulador **ret-distance** la distancia de la rama actual.

  Posteriormente, trataremos de buscar en todas las ramas del grafo hasta encontrar una rama con un camino **con el destiny-id que sea igual al origin-id del primer camino de la última rama considerada; de id *branch-id-aux***.

  Con la siguiente imagen, trataremos de ilustrar cómo queremos obtener la distancia total de la rama, partiendo del nodo destino hasta llegar al nodo origen, buscando ramas intermedias con coincidencias de id.

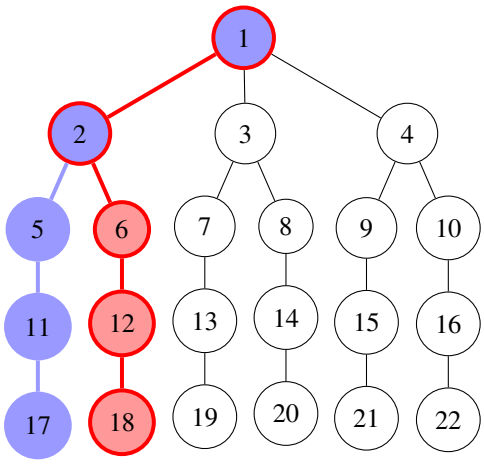

Figura 6.4 Elementos de una rama completa del grafo.

  Una vez que lleguemos a una rama en la que su primer nodo sea el nodo padre del grafo, significa que ya tenemos la distancia total del viaje y que podemos devolver la distancia total acumulada.

- **Líneas 35-38**: Si se cumple equal, significa que la rama en la que estamo contiene los nodos que almacenan, respectivamente, la ciudad **origen** y la ciudad **destino**.
- **Líneas 41-42**: En caso de que la rama esté vacía (sin elementos), devuelvo un 0 en la distancia total acumulada.

get-route

La sección de código correspondiente aparece referenciada en el anexo [A.43].

Como se comentó en apartados anteriores con menor detalle, con **Graph::get-route** pretendemos devolver un vector con los nodos que conforman la ruta del viajero hasta llegar al nodo **n**, que es la entrada del método considerado.

Básicamente, buscamos en las ramas que conforman el grafo de manera similar a como hacíamos en **Graph::get-total-distance**, solo que ahora no consideramos los caminos de las ramas, sino los nodos. Dichos nodos se van añadiendo al vector que devolvemos a la salida; **out-**.

main-algorithm

La sección de código correspondiente aparece referenciada en el anexo [A.44].

Algoritmo principal, que implementa la mayoría de los métodos explicados hasta el momento.

Analicemos las líneas más importantes del código:

- **Línea 12**: Damos valor a **begin**, para contabilizar el tiempo de ejecución del algoritmo posteriormente.
- **Líneas 15-17**: Definimos el **factorial-value**, que nos da información acerca del número de posibles soluciones del problema del viajero. Lo hemos llamado de esta manera aunque realmente no sea un factorial; ***factorial-value* toma el valor del factorial del número de ciudades intermedias consideradas en el problema, que no tiene por qué coincidir con el número de ciudades intermedias totales**. La distinción entre ambas consideraciones la lleva a cabo la variable **threshold**, tal y como se contempla en las líneas de código.

- **Líneas 20-25**: Bucle encargado de la creación de una rama en el grafo. También se vuelca en **distances** el valor de la distancia total asociada a la rama asociada, **considerando la explicación que se dio con respecto al funcionamiento de *Graph::get-total-distance***. El bucle **do-while** sigue creando ramas **mientras el número de ramas creadas no haya alcanzado el número deseado, o mientras dicho número no haya llegado al número de soluciones posibles del problema**.
- **Líneas 31-35**: Por un lado, se almacena en **f-it** el número de componente del vector **distances** con la distancia mínima conseguida y, por tanto, asociado a la mejor solución.

  Por otro lado, almaceno en **n** el nodo de la rama asociada a la mejor solución, y obtengo la ruta completa del viajero usando **Branch::get-route**.

route-output

La sección de código correspondiente aparece referenciada en el anexo [A.45].

Se encarga de devolver la ruta de nodos que conforma la solución del problema del viajero. Se almacena en el atributo denominado **route-out**.

min-distance-output

La sección de código correspondiente aparece referenciada en el anexo [A.46].

Devuelve la distancia mínima asociada a la mejor solución del algoritmo.

Tanto este método como el del apartado anterior se implementaron con el fin de emplearse en la clase **WholeGraph**.

Clase "WholeGraph"

output-little-branches

Muestra la salida de los resultados más significativos del algoritmo para el caso en el que se emplean **varios hilos**.

output-whole-graph

La sección de código correspondiente aparece referenciada en el anexo [A.47].

Muestra la salida de los resultados más significativos del algoritmo para el caso en el que se emplea **un solo hilo hilos**.

main-algorithm

La sección de código correspondiente aparece referenciada en el anexo [A.48].

Algoritmo principal de la clase **WholeGraph**.

En función del valor de **sub-branch-id**, resolveremos el grafo completo o ramificaciones concretas del mismo. Se ha supuesto que, en caso de que su valor sea 99, **se resuelve el grafo completo**. En el caso de que **sub-branch-id** tome el valor de un id asociado a alguna de las ciudades del tsp, se resolveran solo las ramificaciones del árbol **con el nodo hijo del nodo padre que tenga *sub-branch-id* como identificador de su ciudad**. Dicho de otra manera; en este caso se resuelve el tsp partiendo de uno de los hijos del nodo con la ciudad origen. De esta manera, no se consideran las soluciones vinculadas a los otros hijos del nodo con la ciudad de origen, ya que parten de otras ramas.

Con las siguientes imágenes, se ilustra la explicación anterior:

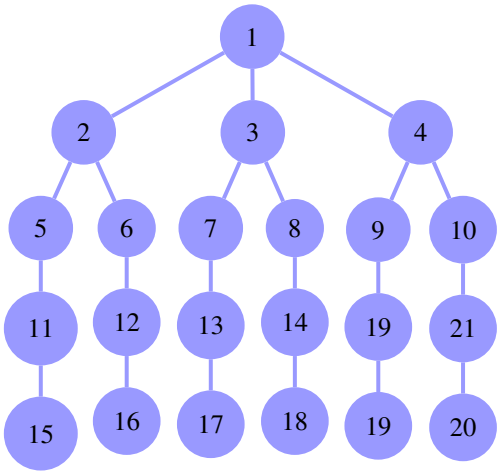

Figura 6.5 Grafo del viajero completo.

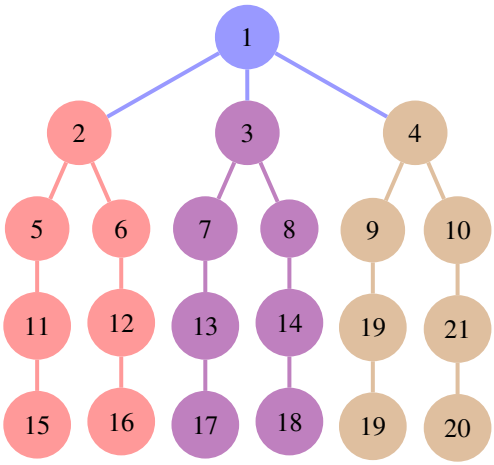

Figura 6.6 División en subramas del grafo del viajero.

A lo largo de todo el código presentado, se definen listas locales en las que se manejan las ciudades intermedias del problema de manera que se consideren aquellas adecuadas en cada uno de los casos que define **sub-branch-id**.

add-aux-cities

La sección de código correspondiente aparece referenciada en el anexo [A.49].

Añade ciudades auxiliares en **aux-cities**. Dichas ciudades se especifican a la entrada.

add-res-cities

La sección de código correspondiente aparece referenciada en el anexo [A.50].

Añade ciudades restringidas en **res-cities**. Dichas ciudades se especifican a la entrada.

Clase "Mtsp"

solve-only-res

La sección de código correspondiente aparece referenciada en el anexo [A.51].

Resuelve el problema del viajero en el caso de que solo se consideren **ciudades restringidas** en el problema. Como las ciudades restringidas, como la propia definición indica, están asociadas a un viajero del MTSP, resolver este problema equivale a resolver **n** TSPs en los que no hay que llevar a cabo repartición alguna de ciudades entre viajeros.

solve-only-aux

La sección de código correspondiente aparece referenciada en el anexo [A.52].

En este método hay que tomar la decisión de cómo asignar ciudades auxiliares entre los distintos viajeros.

Se explicará, de manera general y sin ser excesivamente detallados comentando el código, cómo se ha llevado a cabo dicho reparto y de qué manera actúan los métodos y las variables más importantes.

- Introducimos en el vector de ciudades **c** las coordenadas de las ciudades intermedias junto con sus respectivos identificadores.
- Asignamos **solo el número** de ciudades por las que cada viajero pasará, sin tener en cuenta de qué ciudades en concreto estamos hablando aún. Esta asignación se hace de modo que el reparto sea lo más equitativo posible.
- **Posteriormente, entramos en un bucle while en el que se lleva a cabo a asignación de ciudades a los distintos viajeros**. Se lleva a cabo de la siguiente manera:

  1. Reseteamos, en caso de que se hayan modificado en las iteraciones anteriores, los ids de las ciudades devueltas por **Mtsp::solve-tsp** con información del los ids de ciudades que se asignan a cada viajero (**return-cities-ids**). Hacemos lo mismo con las distancias condicionadas con esta decisión (**min-distances**).
  2. Recurrimos al método que explicaremos posteriormente, **Mtsp::solve-tsp**. Se encarga de resolver el MTSP completo, considerándose en su código cada viajero por separado.
  3. Si la mayor distancia recorrida, de entre todas las distancias obtenidas de los distintos viajeros (**puntero it-dis**), consigue situarse por debajo de un límite superior **max-value**, significa que continuaremos iterando en el bucle **while**. Actualizaremos el límite superior con la distancia almacenada en **it-dis**, y también modificaremos el número de ciudades asignadas a cada viajero en la siguiente iteración (líneas de código 79-84). Se asignará al viajero con el recorrido más corto una ciudad del viajero con el peor resultado, de manera que se mejore el resultado global en la siguiente iteración del bucle **while**.
  4. En caso de que la mayor distancia obtenida con **Mtsp::solve-tsp** sea igual o mayor a la obtenida en la iteración anterior, significará que se encontró la mejor solución en la iteración anterior (**previous-distances**). Aquí finaliza el bucle **while**.

- Tras todos estos pasos, imprimimos por pantalla los ids de las ciudades asignadas a cada uno de los viajeros del MTSP.

solve-aux-and-res

La sección de código correspondiente aparece referenciada en el anexo [A.53].

Este método es idéntico al explicado en el apartado anterior; la única diferencia radica en que, a cada viajero, se le asignan también las ciudades restringidas que hemos especificado previamente. Como se puede hacer esto con **WholeGraph::add-res-cities** antes de aplicar este método, no hace falta llevar a cabo ninguna modificación.

solve-tsp

La sección de código correspondiente aparece referenciada en el anexo [A.54].

Se encarga de modificar el argumentos **return-cities-ids**, de manera que almacene los ids de las ciudades que se asignará a cada viajero del MTSP.

En el código que se muestra a continuación, se puede apreciar que se define un **vector de viajeros *tsp*** a los que se van asignando tanto **ciudades auxiliares** como **ciudades restringidas** antes de aplicar su algoritmo principal de **WholeGraph::main-algorithm**.

Como cada viajero toma a la entrada el número de ciudades por las que tiene que pasar, de entre todas las ciudades intermedias, vemos que cobra sentido la variable **threshold** que tanto se ha mencionado con anterioridad, que definía el nivel de finalización del grafo del viajero.

Tras resolver el viajero del bucle iterativo *for*, se refresca **aux-ids** para descartar las ciudades que se le acaban de asignar al último viajero resuelto. Esto se hace con el fin de que no se vuelva a asignar la misma ciudad a viajeros de las siguientes iteraciones del bucle *for*.

Simulación en Gazebo

En este apartado, trataremos de aplicar el algoritmo del *Problema del Viajero* en el recorrido de un UAV a lo largo de su trayecto, en una simulación en *Gazebo*.

Se irá explicando a lo largo de los siguientes apartados de qué manera se ha implementado la solución del problema, así como el proceso de instalación del paquete de ROS con el que se hace la simulación del UAV, cómo incluir la librería del *MTSP*, y de qué manera emplearla en el código principal del UAV.

Se hará uso del paquete de UAL que provee el Departamento de Automática de la Universidad de Sevilla [15], con el cual hacemos uso del simulador del UAV que llevará a cabo el recorrido especificado por nuestro algoritmo.

Inclusión de la librería MTSP

Añadimos en el CMakeLists, ubicado en el directorio */catkin-grvc*, las siguientes líneas para incluir la librería desarrollada del mtsp:

Código 7.1 Modificación en el CMakeLists.

```
1
2 file(GLOB_RECURSE COMMON_FILES "src/mtsp_files/clases/*.cpp" "src/mtsp_files/clases
      /*.h" "src/mtsp_files/*.cpp" "src/mtsp_files/*.h")
```

Con esto, añadiremos los ficheros y las cabeceras necesarios para obtener una solución del *Travelling Salesman Problem*.

Especificando las siguientes lineas por terminal, hacemos **build** de los paquetes necesarios:

Código 7.2 Compilación de los paquetes requeridos.

```
1
2 catkin build --no-deps px4_bringup robots_description ual_teleop
      uav_abstraction_layer
```

Una vez llegados a este punto, proseguiremos con la implementación en el código de la librería del *MTSP*. Mostraremos los resultados del terminal más significativos y, a su vez, se mostrará la simulación correspondiente del vuelo del UAV.

Código

Se referenciarán, a continuación, dos secciones de código del fichero principal en las cuales se especifica el recorrido del UAV.

Es importante señalar que, en este caso, nos enfrentaremos al *Problema del Viajero* simple (*TSP*).

Para empezar, especificamos las ciudades que se consideran en el problema en el anexo [A.55].

El bucle que se incluye posteriormente en el código (anexo [A.56]) se encarga de definir los waypoints de la misión con la solución del *TSP*. Para la coordenada Z, se ha definido una altura de vuelo constante, por simplicidad.

Se toma la primera componente del vector **def-track-points**, para considerar las **ciudades intermedias** del recorrido. Se procede de la misma manera para considerar las ciudades **origen** y **destino** en el bucle *for* considerado anteriormente. Como se aprecia, vamos incluyendo en cada iteración el waypoint en el recorrido total del UAV (llamado *path*).

Tras esto, simplemente se recorre el camino completo definido hasta ahora. Una vez hecho esto, el UAV aterriza y concluye la misión.

El movimiento del UAV, a nivel de código, se puede apreciar en el anexo [A.57].

Simulación

A continuación se adjuntan capturas de la simulación llevada a cabo en *Gazebo* en las que se aprecia, por un lado, la solución por terminal del *Problema del Viajero*, para posteriormente ir desarrollándose la misión por el UAV, pasando por las ciudades especificadas por el algoritmo. En el terminal, se aprecia cómo el UAV va moviéndose de un waypoint a otro, en el recorrido definido en la variable *path*.

El enlace completo, en el que se aprecia la simulación al completo, es el siguiente:

https://drive.google.com/drive/folders/1BC47YYFt7W4xd6z8HElh4RlO_tOnzE7l

En una de las dos pantallas, se aprecia la simulación en *Gazebo*, mientras que en la otra se observa por terminal la solución del *TSP* y la misión que va desarrollando el UAV.

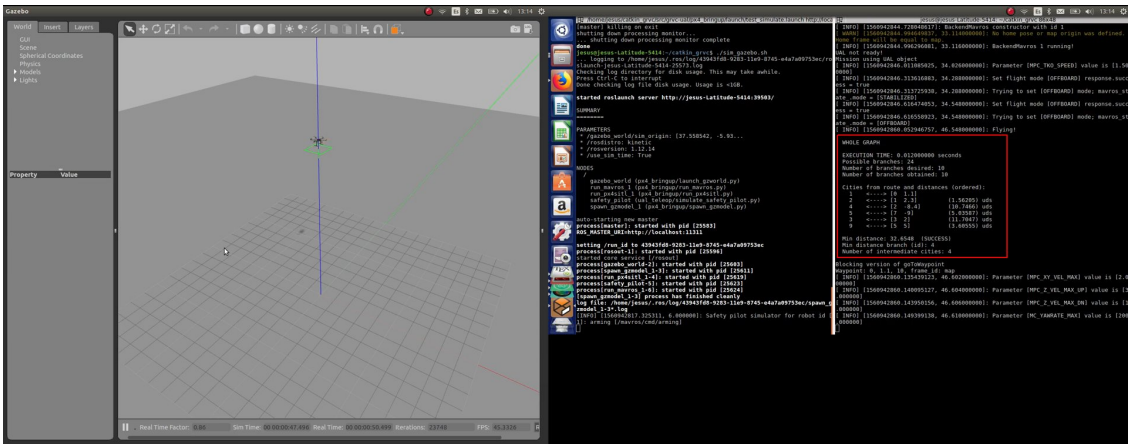

Figura 7.1 Simulación en *Gazebo*: solución del *TSP*.

En esta primera captura de la simulación, destacada con marco rojo en el terminal, se observa la solución obtenida por el algoritmo. Tiene la misma estructura que en el apartado de resultados:

- Por un lado, se muestra el tiempo que tarda el algoritmo en dar la solución del *Problema del Viajero*: unos 12 milisegundos.
- También, aparecen las soluciones del recorrido del UAV a lo largo de su misión, desde ciudad de inicio hasta ciudad destino pasando por los puntos intermedios.

Una vez obtenida esta solución, asignamos la coordenadas en el orden especificado por el algoritmo del viajero, tal y como se expuso en el apartado anterior del código.

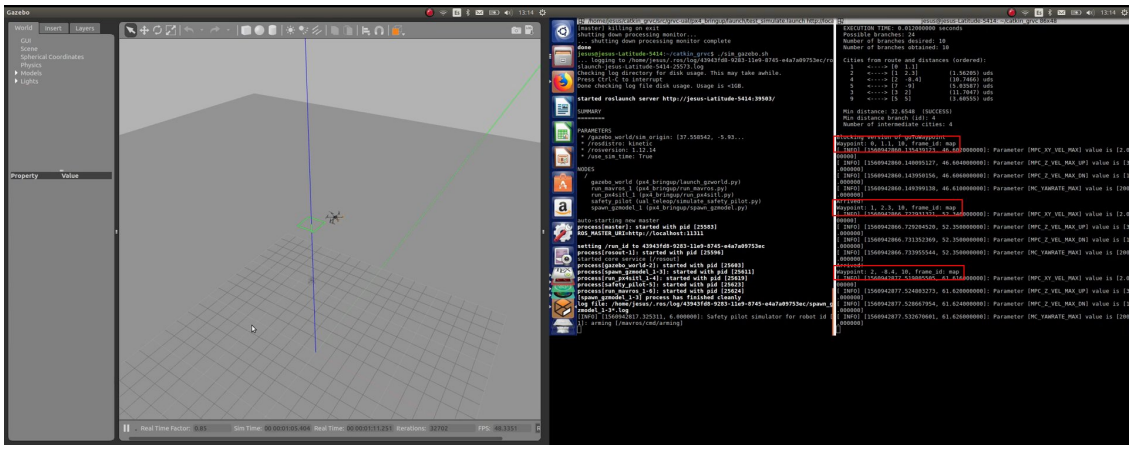

Figura 7.2 Simulación en *Gazebo*: primera parte del recorrido.

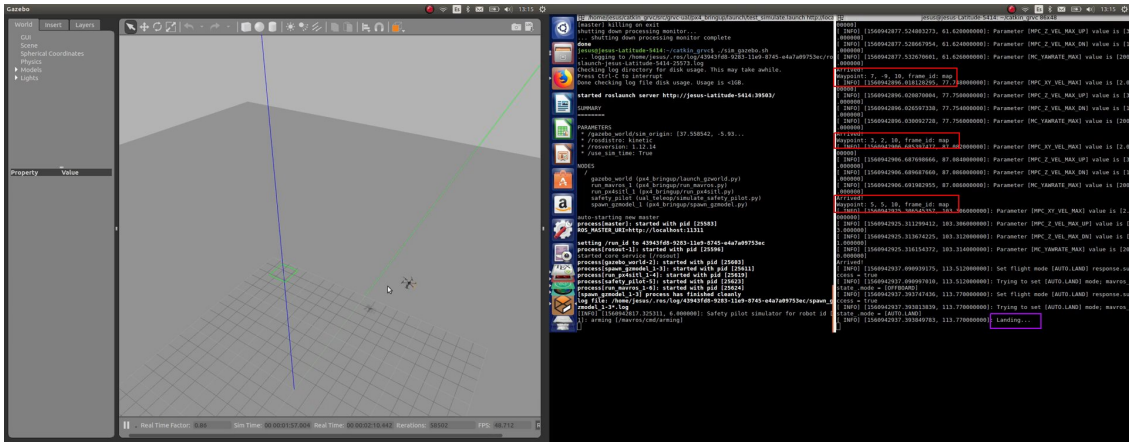

Figura 7.3 Simulación en *Gazebo*: segunda parte del recorrido y aterrizaje.

Como se destaca en el terminal en las dos imágenes anteriores, el UAV va llevando a cabo el recorrido especificado hasta llegar a la ciudad de destino. Una vez alcanzada, se lleva a cabo el aterrizaje.

Conclusiones

Comparando con la solución desarrolada anteriormente en el entorno de *MATLAB*, se puede afirmar lo siguiente:

- **No se emplea una base de datos enorme**, sino que se almacenan las ramas dentro de cada grafo. Tan solo se emplea una base de datos al definir cama rama, en la clase *Graph*.
- Con respecto a MATLAB, **se emplean hilos** (que aportan concurrencia) para poder resolver el mismo problema en menor tiempo para el mismo número de soluciones deseado. Ejecución **paralela** del código en lugar de **secuencial**.
- En el caso en el que nos encontremos ante un grafo con muchas ciudades intermedias, los desgloses de los nodos irán asociados con una creación de ramas cada vez más cortas. Esto se traduce en una menor carga de información almacenada por el programa (lo cual es una ventaja).
- En lugar de considerar todos los datos almacenados en una base de datos de nodos (lo que se hacía en MATLAB), en esta ocasión se define serie de clases para crear objetos que operen y almacenen sobre los datos más cruciales. Permite diferenciar tareas de alto y bajo nivel.

Una vez enumeradas estas mejoras, a modo de conclusión, se puede establecer que con esta nueva menera de abordar el algoritmo del *Problema del Viajero* se obtienen:

- **Más soluciones en menor tiempo**, sirviéndonos del paralelismo que nos proporcionan varios hilos.
- **Menor información asociada a las soluciones**, debido a que almacenamos la información más relevante en objetos del tipo rama, y que su longitud disminuye conforme avanza el algoritmo que recorre el grafo.

Posibles mejoras

De entre las principales mejoras que se podrían llevar a cabo, partiendo de la solución y los resultados propuestos a lo largo de esta memoria, se pueden considerar las siguientes:

- Emplear un número de hilos que se adecúe al procesador del ordenador para resolver el problema del viajero de la manera más eficiente posible (bajo coste computacional para el mayor número de soluciones posible).
- Volviendo al tema de los hilos, considerar que se puede emplear esta técnica para niveles inferiores del grafo. Es decir, en lugar de resolver las ramas principales del grafo (asociadas a las ciudades/nodos del primer nivel), emplear hilos para subramas del segundo o tercer nivel.
- En el caso del *MTSP con ciudades auxiliares*, mejorar el reparto de ciudades entre viajeros. En vez de hacerse de manera aleatoria, averiguar qué ciudad retrasa más a cada viajero para cederla a otro viajero. De la misma manera, podría considerarse a qué viajero perjudica menos obtener una ciudad cedida, sin tener por qué ser el viajero con menos coste asociado a su solución.
- Usar varios UAVS en el caso de la simulación en GAZEBO y resolver un MTSP.

Anexo I: Código empleado

Clases del TSP

Código A.1 Declaración de clase City.

```
1
2  class City
3  {
4
5  public:
6    int id;
7    std::vector<float> city;
8
9    // For using "find_if" with cities
10   bool operator == (const City& s) const { return city == s.city && id == s.id; }
11   bool operator != (const City& s) const { return !operator==(s); }
12
13   void fill(int id_, std::vector<float> city_);
14   std::vector<std::string> get_string_data();
15     };
```

Código A.2 Declaración de clase Node.

```
1
2  class Node: public Element {
3     public:
4       Node(){element_type=NODE;}
5
6       // For using "find_if" with nodes
7      bool operator == (const Node& s) const { return label == s.label && city == s.
           city && level == s.level; }
8      bool operator != (const Node& s) const { return !operator==(s); }
9
10     void fill(City city_,int label_,int level_);
11     void convert_element(Element e);
12 };
```

Código A.3 Declaración de clase Path.

```
1
2  class Path: public Element {
3     public:
4       Path(){element_type=PATH;}
5
6       // For using "find_if" with paths
7       bool operator == (const Path& s) const { return origin_id == s.origin_id &&
            destiny_id == s.destiny_id && traveled == s.traveled && distance == s.
            distance; }
8       bool operator != (const Path& s) const { return !operator==(s); }
9       bool operator < (const Path& s) const { return distance < s.distance; }
10
11     void fill(int origin_id_,int destiny_id_,bool traveled_,float distance_);
12 };
```

Código A.4 Declaración de clase Branch.

```
1
2  class Branch
3  {
4  public:
5    std::deque<Element> elements;
6    int last_element; // "NODE" "PATH" or "-1"
7
8    // CONSTRUCTOR
9    Branch(){ finished=false; last_element=-1; }
10
11   std::vector<City> get_cities();
12   std::vector<Node> get_route(Node n);
13   void add_element(Element e);
14   float get_total_distance();
15   float get_total_distance_destiny_id(int d_id);
16   bool search_node(Node n);
17   bool search_path(Path n);
18
19     };
```

Código A.5 Declaración de clase Element.

```
1
2  class Element {
3     public:
4       int element_type=-1; // NODE or PATH
5       // PATH CLASS DATA
6       int origin_id=-1;
7       int destiny_id=-1;
8       bool traveled=-1;
9       float distance=-1;
10      // NODE CLASS DATA
11      int label=-1;
12      int level=-1;
13      City city;
14 };
```

Código A.6 Declaración de clase Graph.

```
1
2  class Graph
3  {
4  private:
5
6    Node * traveler;
7    Node * break_down;
8    std::deque<Node> nodes;
9    std::deque<Path> paths;
10   int global_node_id, global_path_id;
11   int branches_id;
12
13 public:
14   std::vector<Branch> branches;
15   std::vector<City> cities;
16
17   // OUTPUT VARIABLES
18   int n_branches_private;
19   int factorial=1;
20   std::vector<float> distances;
21   int level_;
22   std::vector<Node> route_out;
23   ros::Time begin;
24   float min_distance;
25
26   Graph(std::vector<City> origin,
27        std::vector<City> cities,
28        std::vector<City> destiny,
29        int n_branches,std::mutex &mtx, int level__,int &threshold); // CONSTRUCTOR
30   void print_nodes_data_base();
31   void print_paths_data_base();
32   void print_data_base();
33   std::pair<std::list<Node>,std::list<Path>> explore_and_order();
34   float distance_between_nodes(Node n1, Node n2);
35   void travel_nearest_city(std::pair<std::list<Node>,std::list<Path>> elements);
36   void main_algorithm(int n_branches,std::mutex &mtx, int level__,int &threshold);
37   std::pair<std::list<Node>,std::list<Path>> remove_branches_elements(std::pair<std::
       list<Node>,std::list<Path>> elements);
38
39   int break_down_son();
40   void create_branch(int &global_node_id_saved,int &move_bd,int &threshold);
41   void refresh_graph_data(Element e, Node n,Node save_traveler,int &move_bd ,std::
       vector<float> &distances,int &threshold);
42   void move_break_down();
43   std::vector<Node> get_route(Node n);
44   std::vector<int> print_results(std::mutex &mtx,int &threshold);
45   float get_total_distance(int branch_id_,int &threshold);
46
47   // OUTPUT METHODS
48   std::vector<Node> route_output();
49   float min_distance_output();
50    };
```

Código A.7 Declaración de clase WholeGraph.

```
1
2  class WholeGraph
3  {
4  private:
5
6  public:
7
8    std::vector<City> res_cities;
9    std::vector<City> aux_cities;
10   float min_distance;
11
12   void output_little_branches(Graph up, Graph down,std::mutex &mtx);
13   std::vector<int> output_whole_graph(Graph down,std::mutex &mtx,int &threshold);
14   std::vector<int> main_algorithm(std::vector<std::vector<float>> origin,
15                     std::vector<float> destiny,int cities_size,
16                     int sub_branch_id,int iterations,
17                     std::mutex &mtx,int number_cities_traveled);
18
19   void add_aux_cities(std::vector<City> cities,std::vector<int> ids_to_include);
20   void add_res_cities(std::vector<City> cities,std::vector<int> ids_to_include);
21
22   };
```

Clases del MTSP

Código A.8 Declaración de clase Mtsp.

```
1
2  class Mtsp
3  {
4
5  public:
6
7     std::vector<float> min_distances;
8
9     void solve_only_res(std::vector<std::vector<float>> origin,
10                     std::vector<std::vector<float>> cities,
11                     std::vector<std::vector<float>> destiny,
12                     int sub_branch_id,int iterations,
13                     std::vector<std::vector<int>> res_ids,std::mutex &mtx);
14
15    void solve_only_aux(std::vector<std::vector<float>> origin,
16                     std::vector<std::vector<float>> cities,
17                     std::vector<std::vector<float>> destiny,
18                     int sub_branch_id,int iterations,
19                     std::vector<int> aux_ids,std::mutex &mtx);
20
21    void solve_aux_and_res(std::vector<std::vector<float>> origin,
22                     std::vector<std::vector<float>> cities,
23                     std::vector<std::vector<float>> destiny,
24                     int sub_branch_id,int iterations,
25                     std::vector<std::vector<int>> res_ids,
26                     std::vector<int> aux_ids,std::mutex &mtx);
27
28    void solve_tsp(std::vector<std::vector<float>> &origin,
29                std::vector<std::vector<float>> &cities,
```

```
30                  std::vector<std::vector<float>> &destiny,
31                  int sub_branch_id,int iterations,
32                  std::vector<int> &aux_ids,
33                  std::vector<std::vector<int>> &res_ids,
34                  std::mutex &mtx,
35                  std::vector<City> &c,
36                  std::vector<std::vector<int>> &return_cities_ids,
37                  std::vector<int> &cities_to_travel,
38                  std::list<int> &aux_ids_list);
39
40
41      };
```

Código del fichero principal

Código A.9 Fichero principal: *codigo-principal.cpp*.

```
1
2   #include "mtsp_files/clases/Mtsp.h"
3
4   using namespace std;
5   using std::vector;
6
7   std::mutex mtx;
8
9   // Required in order to split .txt data
10  void split(const string &s, char delim, vector<string> &elems) {
11      stringstream ss(s);
12      string item;
13      while (getline(ss, item, delim)) { elems.push_back(item);}}
14  vector<string> split(const string &s, char delim) {
15      vector<string> elems;
16      split(s, delim, elems);
17      return elems;}
18
19
20
21  int main( int argc, char** argv ){
22
23    ros::init(argc, argv, "codigo_principal");
24    ros::NodeHandle n;
25    ros::Publisher marker_pub = n.advertise<visualization_msgs::Marker>("
          visualization_marker", 10);
26    ros::Rate r(30);
27    float f = 0.0;
28
29    // Defining vectors with CITIES, ORIGIN and DESTINY
30    std::vector<std::vector<float>> origin, destiny;
31    std::vector<std::vector<float>> cities;
32
33    origin.push_back({0,1.1,0}); // Origin City 1
34
35    // Intermediate Cities
36    cities.push_back({1,2.3,-4}); // 2
37    cities.push_back({3,2,3});    // 3
38    cities.push_back({-2,8.4,4}); // 4
39    cities.push_back({7,9,9});    // 5
40    cities.push_back({2,-9,9});   // 6
```

```
41    cities.push_back({7,9,2});    // 7
42    cities.push_back({7,2,2});    // 8
43
44    destiny.push_back({10,10,10}); // Destiny City 1
45    destiny.push_back({1.5,-2,1}); // Destiny City 2
46
47
48  // Thread with a single TSP
49   auto tsp_thread = [](std::vector<std::vector<float>> origin,
50                        std::vector<std::vector<float>> cities,
51                        std::vector<std::vector<float>> destiny,
52                        int sub_branch_id,int iterations) {
53       Mtsp mtsp;
54
55       std::vector<std::vector<int>> res_ids ={{7,5},{2}} ;
56       std::vector<int> aux_ids={6,3,4};
57        mtsp.solve_aux_and_res(origin,cities,destiny,sub_branch_id,iterations,res_ids,
              aux_ids,mtx);
58       // mtsp.solve_only_res(origin,cities,destiny,sub_branch_id,iterations,res_ids,
             mtx);
59       // mtsp.solve_only_aux(origin,cities,destiny,sub_branch_id,iterations,aux_ids,
             mtx);
60
61
62       };
63
64
65
66      thread th01(tsp_thread,origin,cities,destiny,WHOLE_GRAPH,ITERATIONS);
67
68
69      th01.join();
70
71  return 0;
72  }
```

Visualización *RVIZ*

Código A.10 Ciudades consideradas en la visualización de *RVIZ*.

```
1
2    origin.push_back({0,1.1,0}); // Origin City 1
3
4    // Intermediate Cities
5    cities.push_back({1,2.3,0});  // 2
6    cities.push_back({3,2,0});    // 3
7    cities.push_back({2,-8.4,0}); // 4
8    cities.push_back({7,-9,0});   // 5
9    cities.push_back({-2,9,0});   // 6
10   cities.push_back({-7,2,0});   // 7
11   cities.push_back({-7,9,0});   // 8
12
13   destiny.push_back({5,5,0}); // Destiny City 1
14   destiny.push_back({5,-5,0}); // Destiny City 2
15   destiny.push_back({-9,5,0}); // Destiny City 3
```

Código A.11 Código para resolver el *Problema del Viajero* solo con ciudades restringidas.

```
1
2       Mtsp mtsp;
3       std::vector<std::vector<int>> res_ids ={{7,5},{2,3},{6}} ;
4       mtsp.solve_only_res(origin,cities,destiny,WHOLE_GRAPH,ITERATIONS,res_ids,mtx);
5       def_track_points=mtsp.return_cit_ids();
```

Código A.12 Código *RVIZ*.

```
1
2  while (ros::ok())
3  {
4
5    visualization_msgs::Marker points ,pf, po, line_list;
6    std::vector<visualization_msgs::Marker> line_strip;
7
8    line_strip.resize(destiny.size());
9
10   po.header.frame_id =pf.header.frame_id =points.header.frame_id = line_list.header.
         frame_id = "map";
11   po.header.stamp =pf.header.stamp =points.header.stamp = line_list.header.stamp =
         ros::Time::now();
12   po.ns =pf.ns =points.ns = line_list.ns = "points_and_lines";
13   po.action =pf.action =points.action = line_list.action = visualization_msgs::
         Marker::ADD;
14   po.pose.orientation.w =pf.pose.orientation.w =points.pose.orientation.w =
         line_list.pose.orientation.w = 1.0;
15
16   // FILLING "line_strip" components values
17    for(int i=0;i<destiny.size();i++){
18      line_strip[i].header.frame_id ="map";
19      line_strip[i].header.stamp =ros::Time::now();
20      line_strip[i].ns="points_and_lines";
21      line_strip[i].action=visualization_msgs::Marker::ADD;
22      line_strip[i].pose.orientation.w=1.0;
23      line_strip[i].id = i+4;
24      line_strip[i].type = visualization_msgs::Marker::LINE_STRIP;
25      line_strip[i].scale.x = 0.1;
26
27      // Line strip color
28      if(i==0){ // violet route (traveler 1)
29      line_strip[i].color.a = 1;
30      line_strip[i].color.b = 1;
31      line_strip[i].color.r = 1;
32      line_strip[i].color.g = 0;}
33      if(i==1){ // green route (traveler 2)
34      line_strip[i].color.a = 1;
35      line_strip[i].color.b = 0;
36      line_strip[i].color.r = 0;
37      line_strip[i].color.g = 1;}
38      if(i==2){ // red route (traveler 3)
39      line_strip[i].color.a = 1;
40      line_strip[i].color.b = 0;
41      line_strip[i].color.r = 1;
42      line_strip[i].color.g = 0;}
43    }
44
45
```

```
46
47    points.id = 0;
48    line_list.id = 1;
49    pf.id = 2; po.id = 3;
50
51
52
53    points.type = visualization_msgs::Marker::POINTS;
54    po.type = visualization_msgs::Marker::POINTS;
55    pf.type = visualization_msgs::Marker::POINTS;
56    line_list.type = visualization_msgs::Marker::LINE_LIST;
57  %% rviz2
58  \begin{center}
59   \begin{lstlisting}[language=C,caption={Código \emph{RVIZ}}, breaklines=true, label=
         appendix:rviz2]
60
61  while (ros::ok())
62  {
63
64    visualization_msgs::Marker points ,pf, po, line_list;
65    std::vector<visualization_msgs::Marker> line_strip;
66
67    line_strip.resize(destiny.size());
68
69    po.header.frame_id =pf.header.frame_id =points.header.frame_id = line_list.header.
          frame_id = "map";
70    po.header.stamp =pf.header.stamp =points.header.stamp = line_lis
71
72
73    // POINTS markers use x and y scale for width/height respectively
74    points.scale.x = 0.2;
75    points.scale.y = 0.2;
76
77    po.scale.x = 0.2;
78    po.scale.y = 0.2;
79
80    pf.scale.x = 0.2;
81    pf.scale.y = 0.2;
82
83    // LINE_STRIP/LINE_LIST markers use only the x component of scale, for the line
          width
84    line_list.scale.x = 0.1;
85
86    // Intermediate cities (GREEN)
87    points.color.g = 1.0f;
88    points.color.a = 1.0;
89
90    // Origin city (WHITE)
91    po.color.r = 1.0f;
92    po.color.g = 1.0f;
93    po.color.b = 1.0f;
94    po.color.a = 1.0;
95
96    // Destiny city (RED)
97    pf.color.r = 1.0f;
98    pf.color.a = 1.0;
99
100
101   // Line list is red
102   line_list.color.r = 1.0;
103   line_list.color.a = 1.0;
104
```

```
105
106
107    // Create the vertices for the points and lines
108    for (uint32_t ii = 0; ii < def_track_points.size(); ++ii){ // NUMBER OF TRAVELERS
109    for (uint32_t i = 0; i < (def_track_points[ii].size()+2); ++i) // NUMBER OF CITIES
110    {
111
112
113    if(i==0){ // ORIGIN CITY
114    geometry_msgs::Point p;
115    p.x = (float)origin[0][0] ;
116    p.y = (float)origin[0][1] ;
117    if(cities[0].size()==3){
118    p.z = (float)origin[0][2]; }
119
120    po.points.push_back(p);
121    line_strip[ii].points.push_back(p);
122   }
123
124
125     else if(i==(def_track_points[ii].size()+1)){ // DESTINY CITY
126     geometry_msgs::Point p;
127     p.x = (float)destiny[ii][0] ;
128     p.y = (float)destiny[ii][1] ;
129     if(cities[0].size()==3){
130     p.z = (float)destiny[ii][2]; }
131
132     pf.points.push_back(p);
133     line_strip[ii].points.push_back(p);
134
135
136   }
137
138     else if((i>0) && (i<(def_track_points[ii].size()+1))){ // INTERMEDIATE CITIES
139     geometry_msgs::Point p;
140     p.x = (float)cities[def_track_points[ii][i-1]-2][0] ;
141     p.y = (float)cities[def_track_points[ii][i-1]-2][1] ;
142     if(cities[0].size()==3){
143     p.z = (float)cities[def_track_points[ii][i-1]-2][2]; }
144
145     points.points.push_back(p);
146     line_strip[ii].points.push_back(p);}
147
148    }
149  }// FOR LOOP END
150
151    marker_pub.publish(po);
152    marker_pub.publish(pf);
153    marker_pub.publish(points);
154
155  for(int i=0;i<destiny.size();i++){
156    marker_pub.publish(line_strip[i]); }
157
158    r.sleep();
159
160  }
```

Resultados TSP

Código A.13 Ciudades intermedias consideradas en TSP (*main*).

```

  // Defining vectors with CITIES, ORIGIN and DESTINY
  std::vector<std::vector<float>> origin, destiny;
  std::vector<std::vector<float>> cities;

  origin.push_back({0,1.1,0}); // Origin City

  // 4 Intermediate Cities
  cities.push_back({1,2.3,-4}); // 2
  cities.push_back({3,2,3});    // 3
  cities.push_back({-2,8.4,4}); // 4
  cities.push_back({7,9,9});    // 5
  cities.push_back({2,-9,9});   // 6
  cities.push_back({7,9,2});    // 7


  destiny.push_back({10,10,10}); // Destiny City
```

Código A.14 Código de un hilo para TSP.

```

auto tsp_thread = [](std::vector<std::vector<float>> origin,
                  std::vector<std::vector<float>> cities,
                  std::vector<std::vector<float>> destiny,
                  int sub_branch_id,int iterations) {
    Mtsp mtsp;

    std::vector<std::vector<int>> res_ids ={{2,7,4,3}} ;
    mtsp.solve_only_res(origin,cities,destiny,sub_branch_id,iterations,res_ids,mtx)
        ;

    };
```

Código A.15 Ejecución en *main* para un hilo (TSP).

```

    thread th01(tsp_thread,origin,cities,destiny,WHOLE_GRAPH,ITERATIONS);

    th01.join();
```

Código A.16 Ejecución en *main* para cuatro hilos (TSP).

```

    thread th01(tsp_thread,origin,cities,destiny,1,ITERATIONS);
    thread th02(tsp_thread,origin,cities,destiny,2,ITERATIONS);
    thread th03(tsp_thread,origin,cities,destiny,3,ITERATIONS);
    thread th04(tsp_thread,origin,cities,destiny,4,ITERATIONS);
```

```
6
7       th01.join();
8       th02.join();
9       th03.join();
10      th04.join();
```

Resultados MTSP

Código A.17 Ciudades intermedias consideradas en MTSP (*main*).

```
1
2   // Defining vectors with CITIES, ORIGIN and DESTINY
3   std::vector<std::vector<float>> origin, destiny;
4   std::vector<std::vector<float>> cities;
5
6   origin.push_back({0,1.1,0}); // Origin City 1
7
8   // 4 Intermediate Cities
9   cities.push_back({1,2.3,-4}); // 2
10  cities.push_back({3,2,3});    // 3
11  cities.push_back({-2,8.4,4}); // 4
12  cities.push_back({7,9,9});    // 5
13  cities.push_back({2,-9,9});   // 6
14  cities.push_back({7,9,2});    // 7
15  cities.push_back({7,2,2});    // 8
16
17  destiny.push_back({10,10,10}); // Destiny City 1
18  destiny.push_back({1.5,-2,1}); // Destiny City 2
```

Código A.18 Código de un hilo para MTSP: *ciudades auxiliares*.

```
1
2   auto tsp_thread = [](std::vector<std::vector<float>> origin,
3                        std::vector<std::vector<float>> cities,
4                        std::vector<std::vector<float>> destiny,
5                        int sub_branch_id,int iterations) {
6       Mtsp mtsp;
7
8       std::vector<std::vector<int>> res_ids ={} ;
9       std::vector<int> aux_ids={6,3,4};
10      mtsp.solve_only_aux(origin,cities,destiny,sub_branch_id,iterations,res_ids,mtx)
            ;
11      };
```

Código A.19 Código de un hilo para MTSP: *ciudades restringidas*.

```
1
2   auto tsp_thread = [](std::vector<std::vector<float>> origin,
3                        std::vector<std::vector<float>> cities,
4                        std::vector<std::vector<float>> destiny,
5                        int sub_branch_id,int iterations) {
6       Mtsp mtsp;
```

```
7
8       std::vector<std::vector<int>> res_ids ={{6,4,3},{5,2}} ;
9       std::vector<int> aux_ids={};
10       mtsp.solve_only_res(origin,cities,destiny,sub_branch_id,iterations,res_ids,mtx
             );
11      };
```

Código A.20 Código de un hilo para MTSP: *ciudades restringidas y auxiliares*.

```
1
2   auto tsp_thread = [](std::vector<std::vector<float>> origin,
3                        std::vector<std::vector<float>> cities,
4                        std::vector<std::vector<float>> destiny,
5                        int sub_branch_id,int iterations) {
6       Mtsp mtsp;
7
8       std::vector<std::vector<int>> res_ids ={{7,3},{2}} ;
9       std::vector<int> aux_ids={5,4,6};
10      mtsp.solve_aux_and_res(origin,cities,destiny,sub_branch_id,iterations,res_ids,
            aux_ids,mtx);
11      };
```

Métodos implementados en el algoritmo

Código A.21 Clase City: *fill*.

```
1
2   void City::fill(int id_, vector<float> city_){
3     id=id_;
4     city=city_;
5   };
```

Código A.22 Clase City: *get-string-data*.

```
1
2   vector<std::string> City::get_string_data(){
3
4     // Defining strings
5     vector<std::string> return_strings;
6     ostringstream city_string,id_string;
7
8     // Filling strings with class variables
9     for(int i=0;i<city.size();i++){
10      if(i==0){city_string << "["; }
11      city_string << city[i];
12      if(i<(city.size()-1)){city_string << " "; }
13      if(i==(city.size()-1)){city_string << "]"; }}
14    id_string << id;
15
16    // Return strings in a vector
17    return_strings.push_back(id_string.str()); return_strings.push_back(city_string.
          str());
18    return return_strings;
19  };
```

Código A.23 Clase Node: *fill*.

```
1
2  void Node::fill(City city_,int label_,int level_){
3    city=city_;
4    label=label_;
5    level=level_; };
```

Código A.24 Clase Node: *convert-element*.

```
1
2  void Node::convert_element(Element e){
3    city=e.city;
4    label=e.label;
5    level=e.level;};
```

Código A.25 Clase Path: *fill*.

```
1
2  void Path::fill(int origin_id_,int destiny_id_,bool traveled_,float distance_){
3    origin_id=origin_id_;
4    destiny_id=destiny_id_;
5    traveled=traveled_;
6    distance=distance_;};
```

Código A.26 Clase Branch: *add-element*.

```
1
2  void Branch::add_element(Element e){
3    std::string s;
4    if(e.element_type==PATH){s="PATH";}
5    if(e.element_type==NODE){s="NODE";}
6
7    if(last_element==e.element_type){}
8    //  std::cout << "CANNOT ADD THE ELEMENT" << std::endl;
9    else if(last_element!=e.element_type){
10     if(last_element==-1 && e.element_type==PATH){}
11   //    std::cout << "CANNOT ADD THE ELEMENT" << std::endl;
12     else{
13   //    std::cout << "ADDED " << s << std::endl;
14       last_element=e.element_type;
15       elements.push_back(e); }} };
```

Código A.27 Clase Branch: *get-total-distance*.

```
1
2  float Branch::get_total_distance(){
3    float branch_distance=0;
4    for(int i=0;i<elements.size();i++){
5      if(elements[i].element_type==PATH){branch_distance=branch_distance+elements[i].
           distance;};}
6    return branch_distance ;};
```

Código A.28 Clase Branch: *get-total-distance-destiny-id*.

```
float Branch::get_total_distance_destiny_id(int d_id){
  float branch_distance=0;
  for(int i=0;i<elements.size();i++){
    if(elements[i].element_type==PATH){branch_distance=branch_distance+elements[i].
        distance;
    if(elements[i].destiny_id==d_id){return branch_distance ;}};}
}
```

Código A.29 Clase Branch: *get-cities*.

```
std::vector<City> Branch::get_cities(){
  std::vector<City> c;
  for(int i=0;i<elements.size();i++){
    if(elements[i].element_type==NODE){c.push_back(elements[i].city);};}
  return c ;};
```

Código A.30 Clase Branch: *search-node*.

```
bool Branch::search_node(Node n){
  for(int i=0;i<elements.size();i=i+2){ // Only search for nodes
    if((n.level==elements[i].level)
      &&(n.city==elements[i].city) &&
      (n.label==elements[i].label)){return true;}}
    return false;}
```

Código A.31 Clase Branch: *search-path*.

```
bool Branch::search_path(Path n){
  for(int i=1;i<elements.size();i=i+2){ // Only search for paths
    if((n.destiny_id==elements[i].destiny_id)){return true;}}
    return false;}
```

Código A.32 Clase Branch: *get-route*.

```
std::vector<Node> Branch::get_route(Node n){
  std::vector<Node> nodes_route; Node aux_node;
  for(int i=0;i<elements.size();i=i+2){ // Only search for nodes
    aux_node.convert_element(elements[i]);
    nodes_route.push_back(aux_node);
    if(elements[i].label==n.label){break; /*Found the element and return*/ }}
   return nodes_route; // Element not found
}
```

Código A.33 Clase Graph: *Graph (constructor)*.

```
1
2  Graph::Graph(std::vector<City> origin_,
3               std::vector<City> cities_,
4               std::vector<City> destiny_,
5              int n_branches,std::mutex &mtx, int level__,int &threshold){
6
7  // Local variables
8  Node n; Element e;
9
10 // Id that defines "label" in each node/path from the graph
11 global_node_id=1; branches_id=1;
12
13 // Fill private vector of "cities"
14 cities.push_back(origin_[0]);
15 for(int i=0;i<cities_.size();i++){ cities.push_back(cities_[i]); }
16 cities.push_back(destiny_[0]);
17
18 // Create first node in DATABASE
19 n.fill(cities[0],global_node_id,1);
20 nodes.push_back(n);
21
22 //"Traveler" and "break_down" point to the first node
23 traveler = &nodes[0];
24 break_down = traveler;
25
26 // Create first BRANCH with first ELEMENT (node; *travel)
27 e=*traveler;
28 branches.resize(branches_id);
29 branches[branches_id-1].add_element(e);
30
31 // Algorithm Code
32 main_algorithm( n_branches,mtx, level__,threshold);
33
34 };
```

Código A.34 Clase Graph: *explore-and-order*.

```
1
2  std::pair<std::list<Node>,std::list<Path>> Graph::explore_and_order(){
3    std::pair<std::list<Node>,std::list<Path>> res; // Method output (Next (*traveler)
         's paths and nodes)
4    std::vector<Node> route=get_route(*traveler); // Elements route until reaching
         father's node
5    std::list<City> list_of_cities; std::list<Node> nodes_created, nodes_right; //
         DISORDERED / ORDERED by distance
6    Node n; Path p; std::list<Path> paths_created, paths_right; // DISORDERED /
         ORDERED by distance
7    int id_variable, node_label = global_node_id; // The second label is the private
         one (global_node_id)
8    Branch b; int cities_considered; // Local variable for lambda expression & cities
         considered in the break down
9
10   // Defining "cities" in a list (except DESTINY)
11   if((traveler->level)==(cities.size()-1)){cities_considered=cities.size();} // Only
          DESTINY is missed in graph
12   else{cities_considered=(cities.size()-1);} // Previous travels than the DESTINY
         one.
```

```
13    for(int i=0;i<cities_considered;i++){ list_of_cities.push_back(cities[i]); } //
          Filling list_of_cities
14
15    // Removing cities from the "list_of_cities" (belonging to the "route")
16    for(int i=0;i<(route.size());i++){
17      list_of_cities.remove(route[i].city);}
18
19
20      // Creating both "nodes" and "paths" for the cities remaining in the "
            list_of_cities"
21      for(auto it=list_of_cities.begin();it!=list_of_cities.end();++it){
22
23        // Label for sons
24        node_label++;
25
26        // Nodes & Paths Created; DISORDERED
27        n.fill(*it,node_label,(traveler->level)+1);
28        p.fill(traveler->label,node_label,false,distance_between_nodes((*traveler),n));
29        nodes_created.push_back(n);paths_created.push_back(p);}
30
31
32        // Putting the DISORDERED data in the ORDERED data
33        while(!paths_created.empty() && !nodes_created.empty()){
34
35          // Search the path with minimum distance (store in *path_it)
36          auto path_it = find_if(paths_created.begin(), paths_created.end(), [&
                paths_created] (const Path& s)
37          { auto it=std::min_element(paths_created.begin(), paths_created.end());
38            return ((s.distance)==(it->distance));} );
39
40            // Search the node with the city associated with (*path_it)
41            auto node_it = find_if(nodes_created.begin(), nodes_created.end(), [path_it
                  ] (const Node& s)
42            { return ((s.label)==(path_it->destiny_id));} );
43
44            // Filling "node_right/path_right" , with ordered data (ids-distance
                  relation).
45            global_node_id++;
46            n.fill(node_it->city,global_node_id,(traveler->level)+1);
47            p.fill(traveler->label,global_node_id,false,path_it->distance);
48            nodes_right.push_back(n);paths_right.push_back(p);
49            paths_created.remove(*path_it); nodes_created.remove(*node_it); // Remove
                  from DISORDERED lists and go on
50          }
51
52          // Return nodes and paths ordered (_right)
53          res.first=nodes_right; res.second=paths_right;
54          return res;
55
56  }
```

Código A.35 Clase Graph: *distance-between-nodes*.

```
1
2     float Graph::distance_between_nodes(Node n1, Node n2){
3       float accum=0;
4
5       // Cities from both nodes with same length (supposed)
6       for(int i=0;i<n1.city.city.size();i++){
```

```
7       accum=accum+pow((n1.city.city[i]-n2.city.city[i]),2);}
8     return std::sqrt(accum);}
```

Código A.36 Clase Graph: *travel-nearest-city*.

```
1
2     void Graph::travel_nearest_city(std::pair<std::list<Node>,std::list<Path>>
          elements){
3       // Inserts PATH and NODE in the branch
4       std::list<Node> n=elements.first; std::list<Path> p=elements.second;
5       Path * path_pointer; Element * e;
6
7       //(Add (*traveler) to the branch
8       e=&(*traveler); branches[branches_id-1].add_element(*e);
9
10      // Search for MIN PATH below the traveler
11      auto p_it = find_if(p.begin(), p.end(), [&p] (const Path& s)
12      { auto it=std::min_element(p.begin(), p.end());
13        return ((s.distance)==(it->distance));} );
14
15      // Add MIN PATH to the branch
16      e=&(*p_it); branches[branches_id-1].add_element(*e);
17
18      // Search the node with the city associated with (*p_it)
19      auto n_it = find_if(n.begin(), n.end(), [p_it] (const Node& s)
20      { return ((s.label)==(p_it->destiny_id));} );
21
22      // (*traveler) travels
23      // (USE "NODES" TO CONSERVE ADDRESS WHEN THIS METHOD ENDS)
24      for (int i=0;i<nodes.size();i++){
25        if(nodes[i]==(*n_it)){traveler = &nodes[i];break;}}
26
27      // Add NODE to the branch
28      e=&(*traveler); branches[branches_id-1].add_element(*e);
29
30  }
```

Código A.37 Clase Graph: *refresh-graph-data*.

```
1
2   void Graph::refresh_graph_data(Element e, Node n,Node save_traveler,int &move_bd,std
        ::vector<float> &distances, int &threshold ){
3
4     // Refresh the (*break_down) pointer, and erase the last branch created
5     // (because is empty and useless)
6     if(move_bd==1){ move_break_down();
7        branches.erase(branches.end() - 1);
8        distances.erase(distances.end() - 1);
9        branches_id--;
10      move_bd=0;  }
11
12    // Refresh (*traveler) and save its address
13    traveler=break_down;
14    save_traveler=*traveler;
15    e=*traveler;
16
```

```
17    // Create another branch
18    branches_id++;
19    branches.resize(branches_id);
20    branches[branches_id-1].add_element(e);
21
22    // Clear NODE and PATH databases
23    nodes.clear(); paths.clear();
24
25    // Create first node in DATABASE
26    n.fill(save_traveler.city,save_traveler.label,save_traveler.level);
27    nodes.push_back(n);
28
29    // "Traveler" and "break_down" point to the first node in DATABASE
30    traveler = &nodes[0];
31    break_down = traveler;
32
33  }
```

Código A.38 Clase Graph: *remove-branches-elements*.

```
1
2       std::pair<std::list<Node>,std::list<Path>> Graph::remove_branches_elements(std
            ::pair<std::list<Node>,std::list<Path>> elements){
3         std::list<Node> nn=elements.first; std::list<Path> pp=elements.second;
4         std::pair<std::list<Node>,std::list<Path>> ret_elements=elements;
5         std::vector<Branch> b=branches;
6
7         std::list<Node>::iterator it_n;
8         std::list<Path>::iterator it_p;
9
10        // REMOVE NODES from OUTPUT VALUE (ret_elements)
11        for(int j=0;j<b.size();j++){
12        auto n_it = find_if(nn.begin(), nn.end(), [&b,j] (const Node& s)
13        { return b[j].search_node(s);} );
14        if(n_it!=nn.end()){
15          ret_elements.first.remove(*n_it);}}
16
17        // REMOVE PATHS from OUTPUT VALUE (ret_elements)
18        for(int j=0;j<b.size();j++){
19        auto p_it = find_if(pp.begin(), pp.end(), [&b,j] (const Path& s)
20        { return b[j].search_path(s);} );
21        if(p_it!=pp.end()){
22          ret_elements.second.remove(*p_it);}}
23
24        // UPDATE NODES DATABASE
25        it_n = ret_elements.first.begin();
26        while (it_n!=ret_elements.first.end()){
27          nodes.push_back(*it_n);it_n++;}
28
29        // UPDATE PATHS DATABASE
30        it_p = ret_elements.second.begin();
31        while (it_p!=ret_elements.second.end()){
32          paths.push_back(*it_p); it_p++;}
33
34          return ret_elements;
35      }
```

Código A.39 Clase Graph: *break-down-son*.

```
1
2       int Graph::break_down_son(){
3         std::deque<Element> e; Node *bd=break_down;
4
5
6         // Search BREAK_DOWN in branches
7         for(int j=0;j<branches.size();j++){
8           if(branches[j].search_node(*break_down)==true && branches[j].elements.size()
                >1){
9             e=branches[j].elements;
10            auto e_it = find_if(e.begin(), e.end(), [&bd,&e] (const Element& s)
11            { return bd->label==s.label;} ); // Search (*break_down) ID
12            if(e_it!=e.end() && e_it!=(e.end()-1)){return ((e_it+2)->label)-1;} //
                  Return (*break_down) son's minimum ID
13          }
14        }
15        return -1; // If I dont find (*break_down)
16  }
```

Código A.40 Clase Graph: *create-branch*.

```
1
2     void Graph::create_branch(int &global_node_id_saved,int &move_bd,int &threshold){
3       std::pair<std::list<Node>,std::list<Path>> elements_created;
4       int a; Path p; Node n; Element e; Node* traveler_aux;
5       bool c;
6
7       // (*traveler) travels in each iteration, until reaching destiny and defining
            branch
8       do{
9
10        // When I use THRESHOLD in MTSP. ADD DESTINY WHEN THIS CONDITION IS SATISFIED.
11        if(threshold!=(cities.size()) && threshold==traveler->level){
12          global_node_id_saved++;
13          traveler_aux=traveler;
14          n.fill(cities[cities.size()-1],global_node_id_saved,(traveler->level)+1);
15          nodes.push_back(n); traveler=&nodes[nodes.size()-1];
16          p.fill(traveler_aux->label,traveler->label,false,distance_between_nodes((*
                traveler),(*traveler_aux)));
17          paths.push_back(p);
18          e=p; branches[branches_id-1].add_element(e);
19          e=n; branches[branches_id-1].add_element(e);
20
21          c=((branches[branches_id-1].elements.size()))<
22          ((((2*(cities.size()-(break_down->level)))-1)+2)-(2*(cities.size()-1-
                threshold)));
23
24        }
25        else{ // NORMAL tsp (or threshold==cites.size())
26
27          // Global id (consider global or (*break_down)'s son id)
28          if(traveler==break_down){global_node_id=break_down_son();
29            a=0;
30            if(global_node_id==-1){global_node_id=global_node_id_saved;a=1;}}
31          else{global_node_id=global_node_id_saved;a=1;}
32
```

```
33          // Consider possible (*traveler)'s sons.
34          elements_created=explore_and_order(); // (*traveler)'s sons, ordered by
                distance.
35          elements_created=remove_branches_elements(elements_created); // remove
                elements from previous branches.
36          if(elements_created.first.size()==0 && elements_created.second.size()==0){
                // All elements removed
37            move_bd=1; break;} // (EXIT FROM DO-WHILE) (*break_down) goes to the next
                  global id
38          travel_nearest_city(elements_created); // (*traveler) travels to the next
                city
39          if(a==1){global_node_id_saved=global_node_id;} // Refresh global id
40
41          c=((branches[branches_id-1].elements.size()))<((((2*(cities.size()-(
                break_down->level)))-1)+2));
42        }
43
44      }while(c);
45        // Do until traveler reaches destiny (WHILE CONDITION BASED ON NUMBER OF LEVELS
              FROM GRAPH)
46      }
```

Código A.41 Clase Graph: *move-break-down*.

```
1
2       void Graph::move_break_down(){
3         std::deque<Element> e; Node *bd=break_down, nn;
4
5
6         // Search BREAK_DOWN in branches
7         for(int j=0;j<branches.size();j++){
8            e=branches[j].elements;
9            auto e_it = find_if(e.begin(), e.end(), [&bd,&e] (const Element& s)
10           { return (bd->label)+1==s.label;} );
11           if(e_it!=e.end()){
12           nn.convert_element(*e_it);
13           nodes.push_back(nn);
14           break_down=&(nodes[nodes.size()-1]);}
15        }
16
17      }
```

Código A.42 Clase Graph: *get-total-distance*.

```
1
2       float Graph::get_total_distance(int branch_id_,int &threshold){
3         float ret_distance=0; int branch_id_aux=branch_id_; // ID from branch I am
              considering
4         std::vector<Branch> b=branches; bool minor,equal;
5
6         // Condition for setting branches resize
7         if(threshold!=(cities.size())){ // THRESHOLD FOR MTSP
8           minor=((branches[branches_id-1].elements.size()))<((((2*(cities.size()-1))-1)
                +2)-(2*(cities.size()-1-threshold)));
9           equal=((branches[branches_id-1].elements.size()))==((((2*(cities.size()-1))
                -1)+2)-(2*(cities.size()-1-threshold)));
```

```
10        }
11        else{ // USING NORMAL TSP
12          minor=((branches[branches_id-1].elements.size()))<(((2*(cities.size()-1))-1)
                +2);
13          equal=((branches[branches_id-1].elements.size()))==(((2*(cities.size()-1))-1)
                +2);}
14
15
16        // BRANCH WITH ELEMENTS
17        if(branches[branch_id_-1].elements.size()>0){
18          // Branch SMALLER than the one that starts in ORIGIN and reaches DESTINY
19          if(minor){
20
21            // DISTANCE OF THE CURRENT LITTLE BRANCH
22            ret_distance=ret_distance+branches[branch_id_aux-1].get_total_distance();
23
24            // Add little branches until the ORIGIN (origin_id=1)
25            for(float i=(b.size()-1);i>-1;i--){ // SMALL TO LARGER ONES (I mean
                  BRANCHES)
26            for(float j=(b[i].elements.size()-1);j>-1;j--){ // HIGHER TO LOWER LEVELS
27               // Search for little branches that end with the same node than "
                     branch_id_aux" (and connect)
28               if(b[i].elements[j].destiny_id==branches[branch_id_aux-1].elements[1].
                     origin_id){
29                 ret_distance=ret_distance+b[i].get_total_distance_destiny_id(b[i].
                       elements[j].destiny_id);
30               if(b[i].elements[1].origin_id==1){return ret_distance;}
31               else{branch_id_aux=i+1;} }
32             }}  }
33
34         // Branch that starts in ORIGIN and reaches DESTINY
35          else if(equal){
36              return branches[branch_id_-1].get_total_distance();}
37
38        }
39
40        // BRANCH WITHOUT ELEMENTS (It won't be the case)
41        else{std::cout <<"Branch " << branch_id_ <<" doesn't have elements. Return 0 as
               distance." << std::endl<< std::endl;
42            return 0;}
43
44
45  }
```

Código A.43 Clase Graph: *get-route*.

```
1
2     std::vector<Node> Graph::get_route(Node n){
3       std::vector<std::vector<Node>> var; std::vector<Node> out_; Node nn=n;
4
5       do{
6       for(int i=0;i<branches.size();i++){ // Search all branches
7         if(branches[i].search_node(nn)==1){
8           var.push_back(branches[i].get_route(nn)); // Search branch with (*traveler)
9           nn=var[var.size()-1][0];break;}} // Refresh node I want to search...
10      }while(var[var.size()-1][0].level>1); // ... until I reach top node (FATHER)
11
12      // OUTPUT NODE VECTOR
13      for(float i=(var.size()-1);i>-1;i--){
```

```
14        for(auto j=0;j<(var[i].size());j++){
15
16          // Put in "out_" all nodes from route stored in "var"
17          // (without repeating nodes values twice)
18          if(j==0 && i==(var.size()-1)){out_.push_back(var[i][j]);}
19          else if(i>0 && j==(var[i].size()-1)){}
20          else if(i==0 && j==(var[i].size()-1)){ out_.push_back(var[i][j]);}
21          else{ out_.push_back(var[i][j]);}
22        }}
23
24        return out_; // Return route ("out_")
25  }
```

Código A.44 Clase Graph: *main-algorithm*.

```
1
2     void Graph::main_algorithm(int n_branches,std::mutex &mtx, int level__,int &
          threshold){
3       Node n, save_traveler=*traveler; int factorial_value;
4       Element e; int global_node_id_saved=global_node_id, move_bd=0;
5
6       level_=level__; // Filling private variable
7
8      n_branches++;
9      n_branches_private=n_branches;
10
11    // Start execution time
12     begin = ros::Time::now();
13
14    // Calculate factorial (number of possible solutions of the graph)
15    factorial_value=(cities.size()-2);
16    for(int i = factorial_value; i>0; i--){factorial *= i;
17    if(((factorial_value-i)+2)==threshold){break;}}
18
19    // Create a single branch for the graph
20    do{ create_branch(global_node_id_saved,move_bd,threshold); // Creates the branch
21       distances.push_back(get_total_distance(branches_id,threshold)); // Vector of
             graph branches' distances
22       // print_data_base(); // See database (PATHS AND NODES) if needed to debug
23       refresh_graph_data(e,n,save_traveler,move_bd,distances,threshold); // Refreshes
              (*break_down) and (*traveler)
24
25    } while(branches.size()<(n_branches) && (branches.size()<factorial+1)); // Limit
          of solutions calculated
26
27    //std::cout << "ENTRO EN Graph::main_algorithm" << std::endl;
28
29
30    // Minimum distance
31    auto f_it=std::min_element(distances.begin(), distances.end());
32
33    // Getting route of cities
34    n.convert_element(branches[f_it-distances.begin()].elements[branches[f_it-
          distances.begin()].elements.size()-1]);
35    route_out=get_route(n);
36
37  }
```

Código A.45 Clase Graph: *route-output*.

```
1
2   std::vector<Node> Graph::route_output(){
3     return route_out;}
```

Código A.46 Clase Graph: *min-distance-output*.

```
1
2   float Graph::min_distance_output(){
3     return min_distance;}
```

Código A.47 Clase WholeGraph: *output-whole-graph*.

```
1
2   std::vector<int> WholeGraph::output_whole_graph(Graph down,std::mutex &mtx,int &
        threshold){
3     std::vector<int> return_cities_ids=down.print_results(mtx,threshold);
4     min_distance=down.min_distance_output();
5     return return_cities_ids;}
```

Código A.48 Clase WholeGraph: *main-algorithm*.

```
1
2   std::vector<int> WholeGraph::main_algorithm(std::vector<std::vector<float>> origin,
3                               std::vector<float> destiny,
4                               int cities_size,
5                               int sub_branch_id,int iterations,
6                               std::mutex &mtx,int number_cities_traveled){
7
8     std::vector<City> origin_, origin_2, intermediate, destiny_;
9     std::list<std::vector<float>>::iterator it_cities;
10    std::list<City> all_cities; std::list<City>::iterator it_all;
11    City c; int cities_traveled,threshold;
12    std::vector<int> return_cities_ids;
13
14
15    // Volcar en "cities" todas las ciudades de "aux_cities" y "res_cities"
16    // cities=add_aux_and_res_cities();
17
18    sub_branch_id--; // For displaying id branches STARTING IN 1, NOT IN 0.
19
20    // Filling "all_cities"
21    // Adding ORIGIN
22    c.fill(1,origin[0]);
23    all_cities.push_back(c);
24
25    // Adding RES cities
26    for(int i=0;i<res_cities.size();i++){
27      all_cities.push_back(res_cities[i]);}
28
29    // Adding AUX cities
```

```
30      for(int i=0;i<aux_cities.size();i++){
31        all_cities.push_back(aux_cities[i]);}
32
33      // Adding DESTINY cities
34      c.fill(cities_size+2,destiny); all_cities.push_back(c);
35
36
37      // Defining threshold level for MTSP
38      //if(number_cities_traveled>0 && number_cities_traveled<all_cities.size()-2){
39      //cities_traveled=number_cities_traveled;}
40      //else{cities_traveled=all_cities.size()-2;}
41      threshold=number_cities_traveled+1;
42
43
44      // Considering whole TSP (sub_branch_id==99) or LITTLE BRANCHES (sub_branch_id
            !=99)
45      if(sub_branch_id!=99){ // LITTLE BRANCHES
46         it_all=all_cities.begin(); std::advance (it_all,sub_branch_id+1);
47         origin_.push_back(*it_all);
48         all_cities.remove(*it_all);
49         threshold--;}
50      else{ it_all=all_cities.begin();
51           origin_.push_back(*it_all);} // whole TSP
52
53
54      // Put in "intermediate" (vector) intermediate cities from list "all_cities"
55      it_all=all_cities.begin(); std::advance (it_all,1);
56      for(int i=1;i<all_cities.size()-1;i++){
57        intermediate.push_back(*it_all);
58        std::advance (it_all,1); }
59
60      // Put in "destiny_" cities from "destiny"
61      it_all=all_cities.begin(); std::advance(it_all, all_cities.size()-1);
62      destiny_.push_back(*it_all);
63
64      Graph little_branch(origin_,intermediate,destiny_, iterations,mtx,sub_branch_id,
            threshold);
65
66      // ADDING FIRST PATH TO THE LITTLE BRANCHES' TSP (if needed) AND SHOWING RESULTS
67    if(sub_branch_id!=99){
68       it_all=all_cities.begin(); origin_2.push_back(*it_all);
69       intermediate.clear(); threshold=2;
70       Graph first_path(origin_2,intermediate,origin_, iterations,mtx,sub_branch_id,
             threshold);
71
72       // Join graphs data
73       output_little_branches(first_path,little_branch,mtx); // SEE RESULTS
74    }
75    else if (sub_branch_id==99){ return_cities_ids=output_whole_graph(little_branch,
          mtx,threshold); } // SEE RESULTS
76
77    return return_cities_ids;
78
79     }
```

Código A.49 Clase WholeGraph: *add-aux-cities*.

```
1
2      void WholeGraph::add_aux_cities(std::vector<City> cities,std::vector<int>
           ids_to_include){
```

```
3
4       for(int i=0;i<cities.size();i++){
5        for(int j=0;j<ids_to_include.size();j++){
6         if(cities[i].id==ids_to_include[j]){
7           aux_cities.push_back(cities[i]);}}}
8     }
```

Código A.50 Clase WholeGraph: *add-res-cities*.

```
1
2      void WholeGraph::add_res_cities(std::vector<City> cities,std::vector<int>
           ids_to_include){
3
4       for(int i=0;i<cities.size();i++){
5        for(int j=0;j<ids_to_include.size();j++){
6         if(cities[i].id==ids_to_include[j]){
7           res_cities.push_back(cities[i]);}}}
8     }
```

Código A.51 Clase Mtsp: *solve-only-res*.

```
1
2  void Mtsp::solve_only_res(std::vector<std::vector<float>> origin,
3                    std::vector<std::vector<float>> cities,
4                    std::vector<std::vector<float>> destiny,
5                    int sub_branch_id, int iterations,
6                    std::vector<std::vector<int>> res_ids,std::mutex &mtx){
7
8    // Declare variables
9    std::vector<WholeGraph> tsp;
10   std::vector<std::vector<int>> return_cities_ids;
11   tsp.resize(destiny.size());
12   return_cities_ids.resize(destiny.size());
13
14   // Fill in a vector of cities the vector of floats with "cities" coordinates
15   std::vector<City> c; City ct;
16   for(int i=0;i<cities.size();i++){
17     ct.fill(i+2,cities[i]);
18     c.push_back(ct); }
19
20   // Be aware of repeated ids. Haven't considered that error yet
21   for(int i=0;i<destiny.size();i++){
22   tsp[i].add_res_cities(c,res_ids[i]); }
23
24   // Last argument for cities to travel (0 for all cities). See code for details (
         argumrnt "number_cities_traveled")
25   //tsp1.main_algorithm(origin,destiny[0],cities.size(),sub_branch_id,iterations,mtx
         ,0);
26   //tsp2.main_algorithm(origin,destiny[0],cities.size(),sub_branch_id,iterations,mtx
         ,0);
27
28   for(int i=0;i<destiny.size();i++){
29     return_cities_ids[i]=tsp[i].main_algorithm(origin,destiny[i],cities.size(),
           sub_branch_id,iterations,mtx,0); }
30
31 };
```

Código A.52 Clase Mtsp: *solve-only-aux*.

```
1
2  void Mtsp::solve_only_aux(std::vector<std::vector<float>> origin,
3                    std::vector<std::vector<float>> cities,
4                    std::vector<std::vector<float>> destiny,
5                    int sub_branch_id,int iterations,
6                    std::vector<int> aux_ids,std::mutex &mtx){
7
8    // Declare variables
9    std::vector<int> cities_to_travel;
10   std::vector<std::vector<int>> ids_assigned;
11   int counter=0,cities_remaining;
12   std::vector<std::vector<int>> return_cities_ids,previous_cities_ids;
13   std::list<int> aux_ids_list; std::list<int>::iterator it_list;
14   std::vector<int> aux_ids2; // I'll use this variable when getting better results
15   std::vector<float>::iterator it_min, it_max, it_dis;
16   std::vector<float> previous_distances;
17    float max_value=1000000000;int end=0;
18   int z_min, z_max;
19 std::vector<std::vector<int>> res_ids; // ONLY NEEDED AS INPUT ARGUMENT FOR "
       solve_tsp" (I DONT USE ANY VALUE)
20
21   // Put in "aux_ids2" the "aux_ids" data
22   for(int i=0;i<aux_ids.size();i++){
23     aux_ids2.push_back(aux_ids[i]);}
24
25
26   // Fill in a vector of cities the vector of floats with "cities" coordinates
27   std::vector<City> c; City ct;
28   for(int i=0;i<cities.size();i++){
29     ct.fill(i+2,cities[i]);
30     c.push_back(ct); }
31
32   // Assign NUMBER OF CITIES to each traveler
33   for(int i=0;i<destiny.size();i++){
34   cities_to_travel.push_back(aux_ids.size()/destiny.size());
35   counter=counter+((aux_ids.size())/(destiny.size())); }
36
37   // If there are remaining NUMBER OF CITIES, give to the rest
38     cities_remaining=(aux_ids.size()-counter);
39   if(cities_remaining>0){
40     for(int i=0;i<destiny.size();i++){
41     cities_to_travel[i]=cities_to_travel[i]+1;
42     cities_remaining--;
43     if(cities_remaining==0){break;} }}
44
45
46
47 while(end==0){
48  // Reset vectors that store data
49     return_cities_ids.clear();
50     min_distances.clear();
51     previous_distances.clear();
52     return_cities_ids.resize(destiny.size());
53
54  // Solving TSPs
55     solve_tsp(origin,cities,destiny,sub_branch_id,iterations,
56              aux_ids,res_ids,mtx,c,return_cities_ids,cities_to_travel,aux_ids_list);
57
58    // Put in "aux_ids" the "aux_ids2" data (RESET AUX_CITIES)
59    for(int i=0;i<aux_ids2.size();i++){
```

```
60      aux_ids.push_back(aux_ids2[i]);}
61
62      // Put in "previous_distances" the "min_distances" data
63      for(int i=0;i<min_distances.size();i++){
64        previous_distances.push_back(min_distances[i]);}
65        it_dis=std::max_element(previous_distances.begin(), previous_distances.end());
66        std::cout << " max_value: " << max_value << std::endl;
67        std::cout << " *it_dis: " << *it_dis << std::endl<< std::endl;
68        if(max_value>*it_dis){max_value=*it_dis;
69
70          // Put in "previous_cities_ids" the "return_cities_ids" data (ONLY WHEN
              GOING ON)
71          previous_cities_ids.clear();
72          previous_cities_ids.resize(destiny.size());
73          for(int i=0;i<return_cities_ids.size();i++){
74            previous_cities_ids.push_back(return_cities_ids[i]);}
75
76          std::cout << "(GO ON) MAX ID VALUE ("<< it_dis-previous_distances.begin()<<")
                : "<< max_value << std::endl;
77
78          // Improvements TSP
79             it_min=std::min_element(min_distances.begin(), min_distances.end());
80             it_max=std::max_element(min_distances.begin(), min_distances.end());
81             z_min=(it_min-min_distances.begin());
82             z_max=(it_max-min_distances.begin());
83             cities_to_travel[z_min]++;
84             cities_to_travel[z_max]--;
85
86      } // IF this condition gets satisfied, go on with iterations in "solve_only_aux"
87      else{end=1;
88           std::cout << "(FINISHED) MAX ID VALUE: "<< max_value << std::endl;}
89
90
91
92  } // END WHILE
93
94  std::cout << std::endl<< std::endl<< std::endl<< std::endl;
95  std::cout << "previous_cities_ids DATA: "<< std::endl;
96
97  for(int i=0;i<previous_cities_ids.size();i++){
98    for(int j=0;j<previous_cities_ids[i].size();j++){
99    std::cout << previous_cities_ids[i][j] << std::endl;}std::cout << std::endl;}
100
101 };
```

Código A.53 Clase Mtsp: *solve-aux-and-res*.

```
1
2   void Mtsp::solve_aux_and_res(std::vector<std::vector<float>> origin,
3                    std::vector<std::vector<float>> cities,
4                    std::vector<std::vector<float>> destiny,
5                    int sub_branch_id,int iterations,
6                    std::vector<std::vector<int>> res_ids,
7                    std::vector<int> aux_ids,std::mutex &mtx){
8
9     // Declare variables
10    std::vector<int> cities_to_travel;
11    std::vector<std::vector<int>> ids_assigned;
12    int counter=0,cities_remaining;
```

```
13    std::vector<std::vector<int>> return_cities_ids,previous_cities_ids;
14    std::list<int> aux_ids_list; std::list<int>::iterator it_list;
15    std::vector<int> aux_ids2; // I'll use this variable when getting better results
16    std::vector<float>::iterator it_min, it_max, it_dis;
17    std::vector<float> previous_distances;
18     float max_value=1000000000;int end=0;
19    int z_min, z_max;
20
21    // Put in "aux_ids2" the "aux_ids" data
22    for(int i=0;i<aux_ids.size();i++){
23      aux_ids2.push_back(aux_ids[i]);}
24
25
26    // Fill in a vector of cities the vector of floats with "cities" coordinates
27    std::vector<City> c; City ct;
28    for(int i=0;i<cities.size();i++){
29      ct.fill(i+2,cities[i]);
30      c.push_back(ct); }
31
32    // Assign NUMBER OF CITIES to each traveler
33    for(int i=0;i<destiny.size();i++){
34    cities_to_travel.push_back(aux_ids.size()/destiny.size());
35    counter=counter+((aux_ids.size())/(destiny.size())); }
36
37    // If there are remaining NUMBER OF CITIES, give to the rest
38      cities_remaining=(aux_ids.size()-counter);
39    if(cities_remaining>0){
40      for(int i=0;i<destiny.size();i++){
41      cities_to_travel[i]=cities_to_travel[i]+1;
42      cities_remaining--;
43      if(cities_remaining==0){break;} }}
44
45
46
47   while(end==0){
48   // Reset vectors that store data
49     return_cities_ids.clear();
50     min_distances.clear();
51     previous_distances.clear();
52     return_cities_ids.resize(destiny.size());
53
54   // Solving TSPs
55     solve_tsp(origin,cities,destiny,sub_branch_id,iterations,
56              aux_ids,res_ids,mtx,c,return_cities_ids,cities_to_travel,aux_ids_list);
57
58    // Put in "aux_ids" the "aux_ids2" data (RESET AUX_CITIES)
59    for(int i=0;i<aux_ids2.size();i++){
60      aux_ids.push_back(aux_ids2[i]);}
61
62      // Put in "previous_distances" the "min_distances" data
63      for(int i=0;i<min_distances.size();i++){
64        previous_distances.push_back(min_distances[i]);}
65        it_dis=std::max_element(previous_distances.begin(), previous_distances.end());
66        std::cout << " max_value: " << max_value << std::endl;
67        std::cout << " *it_dis: " << *it_dis << std::endl<< std::endl;
68        if(max_value>*it_dis){max_value=*it_dis;
69
70          // Put in "previous_cities_ids" the "return_cities_ids" data (ONLY WHEN
              GOING ON)
71          previous_cities_ids.clear();
72          previous_cities_ids.resize(destiny.size());
73          for(int i=0;i<return_cities_ids.size();i++){
```

```
74            previous_cities_ids.push_back(return_cities_ids[i]);}
75
76          std::cout << "(GO ON) MAX ID VALUE ("<< it_dis-previous_distances.begin()<<")
                : "<< max_value << std::endl;
77
78          // Improvements TSP
79             it_min=std::min_element(min_distances.begin(), min_distances.end());
80             it_max=std::max_element(min_distances.begin(), min_distances.end());
81             z_min=(it_min-min_distances.begin());
82             z_max=(it_max-min_distances.begin());
83             cities_to_travel[z_min]++;
84             cities_to_travel[z_max]--;
85
86       } // IF this condition gets satisfied, go on with iterations in "solve_only_aux"
87       else{end=1;
88           std::cout << "(FINISHED) MAX ID VALUE: "<< max_value << std::endl;}
89
90
91
92  } // END WHILE
93
94  std::cout << std::endl<< std::endl<< std::endl<< std::endl;
95  std::cout << "previous_cities_ids DATA: "<< std::endl;
96
97  for(int i=0;i<previous_cities_ids.size();i++){
98    for(int j=0;j<previous_cities_ids[i].size();j++){
99    std::cout << previous_cities_ids[i][j] << std::endl;}std::cout << std::endl;}
100
101 };
```

Código A.54 Clase Mtsp: *solve-tsp*.

```
1
2   void Mtsp::solve_tsp(std::vector<std::vector<float>> &origin,
3                std::vector<std::vector<float>> &cities,
4                std::vector<std::vector<float>> &destiny,
5                int sub_branch_id,int iterations,
6                std::vector<int> &aux_ids,
7                std::vector<std::vector<int>> &res_ids,
8                std::mutex &mtx,
9                std::vector<City> &c,
10               std::vector<std::vector<int>> &return_cities_ids,
11               std::vector<int> &cities_to_travel,
12               std::list<int> &aux_ids_list){
13
14    std::list<int>::iterator it_list;
15    std::vector<WholeGraph> tsp;
16    tsp.resize(destiny.size());
17
18
19
20    // Solving TSPs
21    for(int z=0;z<destiny.size();z++){
22
23       // z traveler considers all AUX cities
24        tsp[z].add_aux_cities(c,aux_ids);
25
26        // ADDING RES CITIES TO TRAVELERS (DIRECTLY)
27        if(res_ids.size()>0){
```

```
28        tsp[z].add_res_cities(c,res_ids[z]); }
29
30        // z traveler
31        if(res_ids.size()>0){ // AUX AND RES CITIES
32
33        return_cities_ids[z]=tsp[z].main_algorithm(origin,destiny[z],
34                            cities.size(),sub_branch_id,iterations,mtx,cities_to_travel[z
                                ]+res_ids[z].size());
35        }
36        else{ // ONLY AUX CITIES
37          return_cities_ids[z]=tsp[z].main_algorithm(origin,destiny[z],
38                              cities.size(),sub_branch_id,iterations,mtx,cities_to_travel
                                  [z]);}
39
40
41      // Remove "return_cities_ids[z]" from "aux_ids"
42        // Copy IDs to a list (in order to remove later)
43        for(int i=0;i<aux_ids.size();i++){
44          aux_ids_list.push_back(aux_ids[i]);}
45        // Remove from list ids stored in "return_cities_ids[0]"
46        for(int i=0;i<return_cities_ids[z].size();i++){
47          aux_ids_list.remove(return_cities_ids[z][i]);}
48        // Put in vector values from list again
49        it_list=aux_ids_list.begin();
50        aux_ids.clear();
51        for(int i=0;i<aux_ids_list.size();i++){
52          aux_ids.push_back(*it_list);
53          std::advance(it_list, 1);}
54
55       // See "min_distance" output
56          min_distances.push_back(tsp[z].min_distance);
57       //  std::cout << " MIN DISTANCE "<< z <<": " << min_distances[z] << std::endl;
58
59      }
60
61    }
```

Simulación en *Gazebo*

Código A.55 Ciudades especificadas para *Gazebo*..

```
1
2   // For MTSP
3       // -------------------------------------------------------------
4       // Defining vectors with CITIES, ORIGIN and DESTINY
5       std::vector<std::vector<float>> origin, destiny;
6       std::vector<std::vector<float>> cities;
7       std::vector<std::vector<int>> def_track_points;
8
9       origin.push_back({0,1.1}); // Origin City 1
10
11      // 4 Intermediate Cities
12      cities.push_back({1,2.3});    // 2
13      cities.push_back({3,2});      // 3
14      cities.push_back({2,-8.4});   // 4
15      cities.push_back({7,-9});     // 5
16      cities.push_back({-2,9});     // 6
17      cities.push_back({-7,2});     // 7
```

```
18      cities.push_back({-7,9});     // 8
19
20      destiny.push_back({5,5}); // Destiny City 1
21
22      // -----------------------------------------------------
23
24      Mtsp mtsp;
25      std::vector<std::vector<int>> res_ids ={{2,3,4,5}} ;
26      mtsp.solve_only_res(origin,cities,destiny,WHOLE_GRAPH,10,res_ids,mtx);
27      def_track_points=mtsp.return_cit_ids();
```

Código A.56 Asignación de coordenadas de las ciudades.

```
1
2       // Loop for adding MTSP track points
3       for(int i=0;i<def_track_points[0].size()+2;i++){
4       grvc::ual::Waypoint waypoint;
5       waypoint.header.frame_id = "map";
6       if(i==0){ // ORIGIN
7       waypoint.pose.position.x = origin[0][0];
8       waypoint.pose.position.y = origin[0][1];
9       }
10      else if(i==(def_track_points[0].size()+1)){ // DESTINY
11      waypoint.pose.position.x = destiny[0][0];
12      waypoint.pose.position.y = destiny[0][1];
13      }
14      else if(i>0 && i<(def_track_points[0].size()+1)){ // CITIES
15      waypoint.pose.position.x = cities[def_track_points[0][i-1]-2][0];
16      waypoint.pose.position.y = cities[def_track_points[0][i-1]-2][1];
17      }
18      waypoint.pose.position.z = flight_level; //flight_level;
19      waypoint.pose.orientation.x = 0;
20      waypoint.pose.orientation.y = 0;
21      waypoint.pose.orientation.z = 0;
22      waypoint.pose.orientation.w = 1;
23      path.push_back(waypoint);}
```

Código A.57 Sección de código encargada del movimiento del UAV.

```
1
2       std::cout << "Blocking version of goToWaypoint" << std::endl;
3       for (auto p : path) {
4           std::cout << "Waypoint: " << p.pose.position.x << ", " << \
5               p.pose.position.y << ", " << p.pose.position.z << ", frame_id: " << p.
                    header.frame_id << std::endl;
6           ual.goToWaypoint(p);
7           std::cout << "Arrived!" << std::endl;
8       }
9
10      // Land
11      ual.land();
```

Índice de Figuras

Índice de Códigos

Bibliografía

[1] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on evolutionary computation*, vol. 1, no. 1, pp. 53–66, 1997.

[2] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, 1965.

[3] C. C. Murray and A. G. Chu, "The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery," *Transportation Research Part C: Emerging Technologies*, vol. 54, pp. 86–109, 2015.

[4] E. Semsch, M. Jakob, D. Pavlicek, and M. Pechoucek, "Autonomous uav surveillance in complex urban environments," in *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology-Volume 02*. IEEE Computer Society, 2009, pp. 82–85.

[5] C.-W. Lim, S. Park, C.-K. Ryoo, K. Choi, and J.-H. Cho, "A path planning algorithm for surveillance uavs with timing mission constrains," in *ICCAS 2010*. IEEE, 2010, pp. 2371–2375.

[6] J. Isaacs and J. Hespanha, "Dubins traveling salesman problem with neighborhoods: A graph-based approach," *Algorithms*, vol. 6, no. 1, pp. 84–99, 2013.

[7] J. Faigl and P. Váňa, "Unsupervised learning for surveillance planning with team of aerial vehicles," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 4340–4347.

[8] K. Obermeyer, "Path planning for a uav performing reconnaissance of static ground targets in terrain," in *AIAA Guidance, Navigation, and Control Conference*, 2009, p. 5888.

[9] X. Zhang, J. Chen, B. Xin, and Z. Peng, "A memetic algorithm for path planning of curvature-constrained uavs performing surveillance of multiple ground targets," *Chinese Journal of Aeronautics*, vol. 27, no. 3, pp. 622–633, 2014.

[10] C. Lim, C. Ryoo, K. Choi, and J.-H. Cho, "Path generation algorithm for intelligence, surveillance and reconnaissance of an uav," in *Proceedings of SICE Annual Conference 2010*. IEEE, 2010, pp. 1274–1277.

[11] S. Rathinam and R. Sengupta, "Lower and upper bounds for a multiple depot uav routing problem," in *Proceedings of the 45th IEEE Conference on Decision and Control*. IEEE, 2006, pp. 5287–5292.

[12] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[13] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski, "An integrated semantics for uml class, object and state diagrams based on graph transformation," in *International Conference on Integrated Formal Methods*. Springer, 2002, pp. 11–28.

[14] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *ACM Sigplan Notices*, vol. 21, no. 11. ACM, 1986, pp. 38–45.

[15] P. R. S. J. C. Fran Real, Arturo Torres-Gonzalez and A. Ollero, “Ual: an abstraction layer for unmanned vehicles,” in *2nd International Symposium on Aerial Robotics (ISAR)*, 2018.