Grado en ingeniería informática de Gestión y Sistemas de Información

Algoritmos de búsqueda de rutas con puntos intermedios y navegación indoor.

Un trabajo presentado en cumplimiento de los requisitos del trabajo de fin de grado de ingeniería informática de gestión y sistemas de la información por

Francisco Ruiz de Alegría Íñiguez de Heredia

2023

Grado en ingeniería informática de Gestión y Sistemas de Información

Algoritmos de búsqueda de rutas con puntos intermedios y navegación indoor.

Un trabajo presentado en cumplimiento de los requisitos del trabajo de fin de grado de ingeniería informática de gestión y sistemas de la información por Francisco Ruiz de Alegría Íñiguez de Heredia bajo la supervesión de Dr. Miguel Larrañaga Olagaray

2023

Abstract

En este trabajo se aborda el problema de la búsqueda de rutas entre dos puntos, con la posibilidad de definir puntos de paso intermedios. El problema de búsqueda de rutas es uno de los grandes retos informáticos y matemáticos, ya que se trata de un problema con una complejidad NP-Completo. Los planos sobre los que se buscarán las rutas se transformarán en grafos, por lo que se ha realizado un estudio sobre la teoría de grafos y se han identificado los elementos que resultan útiles para que los algoritmos de búsqueda sean más eficientes. Para la búsqueda de rutas con puntos intermedios se han estudiado en profundidad tres algoritmos: (1) El algoritmo de Dijkstra que ofrece una solución óptima a base de examinar todas las posibles soluciones. (2) El algoritmo *Ant Colony Optimization (ACO)*, que es una solución totalmente heurística y que permite obtener soluciones adecuadas en un tiempo de cómputo razonable. (3) En último lugar se implementará el algoritmo de ramificación y acotamiento, *Branch and Bound*, el cual a través de una función heurística devuelve una ruta óptima. La solución propuesta se ha implementado e integrado en dos aplicaciones reales, por un lado una aplicación web y por otro una aplicación móvil.

Índice general

Índice de figuras

Índice de tablas

Listados

Índice de algoritmos

Siglas

MM**AS** $MAX - MIN$ Ant System.

ACO Ant Colony Optimization.

AS Ant System.

GA genetic algorithm.

TS Tabu Search.

TSP Travelling Salesman Problem.

1 Introducción

En el ámbito de la algoritmia computacional, la búsqueda de rutas óptimas, es uno de los temas más estudiados. Un ejemplo de ello es el problema del vendedor viajero, *Travelling Salesman Problem*[1], en ingles, el cual responde a la siguiente pregunta, "dada una lista de ciudades y las distancias entre cada par de ciudades, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad de origen?". Esta pregunta plantea un problema NP-Completo[2] en optimización combinatoria, importante en informática teórica e investigación de operaciones.

En el caso de búsqueda de rutas de un punto a otro sin puntos intermedios, la mejor opción es el uso del algoritmo de Dijkstra[3], el cual busca el camino mas corto entre un punto de origen a un punto de destino. Para ello, recorre todos los caminos posibles entre el origen y el destino.

Cuando se trata de buscar una ruta, con paradas intermedias, las cuales hay que ordenar para optimizar la ruta, el algoritmo de Dijkstra es ineficiente a partir de un determinado número de paradas intermedias, ya que el algoritmo tiene un orden de complejidad factorial, y en tiempo de computo la solución no es eficiente.

Para el caso en el que algoritmo de Dijkstra deja de ser eficiente, se ha hecho un estudio de diferentes soluciones, y se ha optado por el algoritmo heurístico ACO[4], el cual está basado en el comportamiento real que tienen las hormigas para comunicarse entre ellas a través de las feromonas para guiarles en busca de alimentos o de regreso

[1] https://en.wikipedia.org/wiki/Travelling_salesman_problem
[2] https://es.wikipedia.org/wiki/NP-hard
[3] https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra
[4] https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

al hormiguero. La solución ofrecida por ACO no tiene por que ser la mejor solución al problema, pero si que ofrece una solución óptima en un tiempo razonable.

En último caso se ha estudiado una solución híbrida, combinando varios algoritmo. Por un lado el algoritmo de Dijkstra, que creará un subgrafo partiendo del grafo original. Por otro lado un algoritmo para calcular la ruta. Los algoritmos para el cálculo de ruta utilizarán el subgrafo generado para ordenar las paradas intermedias e intentar conseguir la menor ruta. Las soluciones que se han empleado para el calculo de la ruta serán la de ramificación y acotamiento, *Branch and bound*[5],y la de ACO, también se ha estudiado las ventajas e inconvenientes que ofrecen cada una de ellas.

Tanto el algoritmo de Dijkstra como el algoritmo ACO operan sobre grafos, por lo que la teoria de grafos va a tener gran importancia en este proyecto, donde cada mapa tendrá asociado un grafo, en el cual habrá diferentes tipos de nodos, y aristas ponderadas, las cuales podrán ser dirigidas o no dirigidas.

Los datos que se necesitan para el desarrollo del trabajo se han sacado de dos tipos de bases de datos. En una base de datos relacional, se tendrá la información del mapa y de los grafos, y en otra base de datos no relacional se recogerán las ubicaciones de algunos objetos en tiempo real, como puede ser la localización de partida, o determinados puntos de destino, los cuales no son estáticos, sino dinámicos.

En último lugar, se ha integrado el programa realizado de búsqueda de rutas, con un frontend desarrollado en angularJS y en un backend desarrollado en Groovy, donde se puede visualizar los resultados, además se ha desarrollado una aplicación móvil implementada con Ionic, donde se puede seguir la ruta calculada, en modo navegación.

[5] https://en.wikipedia.org/wiki/Branch_and_bound

2 Planificación

Tener una buena planificación es importante para llevar a buen puerto el proyecto, por lo que se va a definir y programar las tareas a realizar, e identificar y minimizar los posibles riesgos que se puedan dar en el transcurso del trabajo.

2.1. Distribución del proyecto

El proyecto se va a dividir en tareas principales y estás a su vez en subtareas, las cuales se van a programar como se puede observar en la Figura 2.2. Tanto las tareas como la programación pueden estar sujetas a cambios durante el transcurso del proyecto. La estructura del desglose de las tareas se puede ver en la Figura 2.1, además se incluye una breve descripción de cada tarea.

- **T1 Gestión del proyecto**
  - **T1.1** Planificación: Dividir y programar el trabajo.
  - **T1.2** Riesgos para su correcto desarrollo: Evaluar los riesgos que pueden suceder para minimizarlos
  - **T1.3** Seguimiento: Revisiones del trabajo e identificar los problemas que van surgiendo.
- **T2 Investigación**
  - **T2.1** Algoritmos de rutas mínimas: Investigar los tipos de algoritmos existentes para el cálculo de rutas mínimas

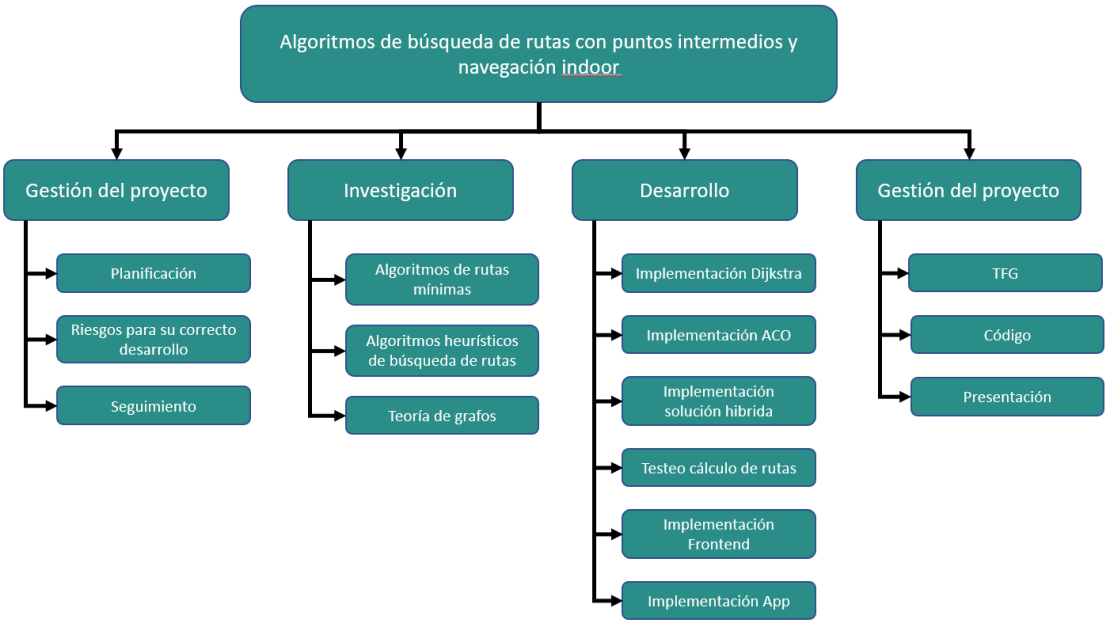


Figura 2.1 – Diagrama de desglose del proyecto

- **T2.2** Algoritmos heurísticos de búsqueda de rutas: Investigar los tipos de algoritmos heurísticos existentes para el cálculo de rutas.
- **T2.3** Teoría de grafos: Investigar como poder trabajar con diferentes tipos de grafos.

- **T3 Desarrollo**
  - **T3.1** Implementación Dijkstra: Implementar el algoritmo Dijkstra en Python.
  - **T3.2** Implementación ACO Implementar el algoritmo ACO en Python
  - **3.3** Implementación de la solución híbrida.
  - **T3.4** Testeo de cálculo de rutas: Comprobar el resultado de las rutas obtenidas
  - **T3.5** Implementación FrontEnd: Implementar la visualización web de las rutas trabajando con AngularJS.
  - **T3.6** Implementación App: Desarrollar la navegación en una aplicación móvil con Ionic.

- **Documentación**

- **T4.1** TFG: Redacción de este documento.
- **T4.2** Código: Documentar el código fuente utilizado.
- **T4.3** Presentación: Preparar la presentación de este proyecto.

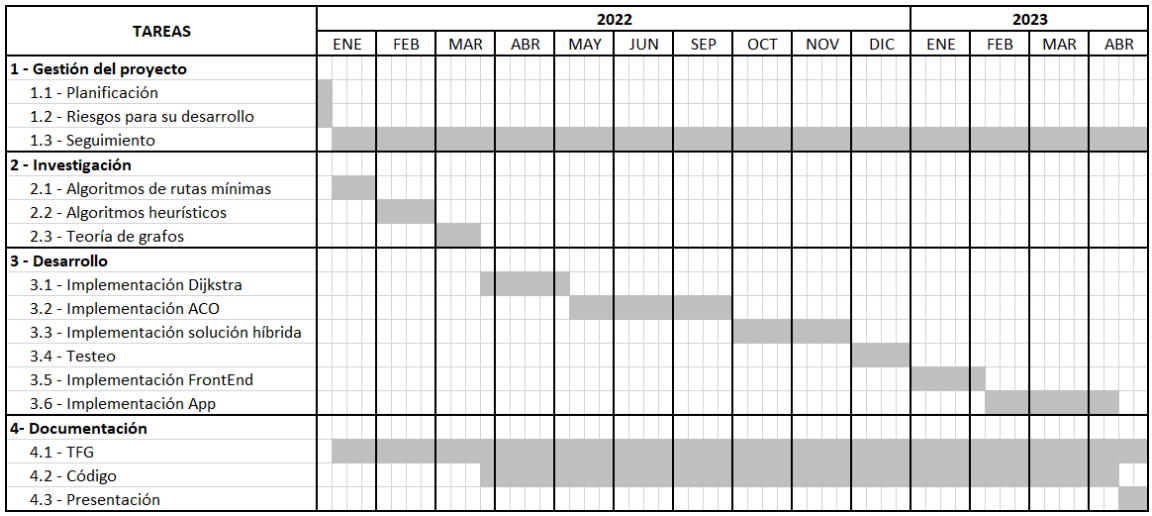


Figura 2.2 – Diagrama de Gantt, muestra el periodo de tiempo para cada una de las tareas identificadas

2.2. Gestión de riesgos

Después de analizar el proyecto, se han identificado los riesgos que se describen a continuación, y si es posible se propone un plan de contingencia.

- **R1:** El trabajo está desarrollado por una única persona, por lo que si cae enferma el proyecto se demorará hasta su recuperación, pare este riesgo no hay solución posible, ya que al autor del proyecto no se le puede sustituir.
- **R2:** ACO es un algoritmo heurístico, por lo que el resultado que pueda dar puede que resulte ser no satisfactorio para los objetivos que tiene la empresa, ya que se pide un resultado, que aunque no sea el mejor, sea óptimo. Además los trabajos estudiados sobre dicho algoritmo están aplicados en el problema del TSP el cual utiliza un grafo diferente al que se va a usar en este proyecto. Para minimizar este problema, se intentarán buscar soluciones alternativas a los problemas que surjan.

- **R3:** El algoritmo ACO ejecutado con grafos grandes requiere mucho tiempo de computación, esto es un problema ya que uno de los requisitos que requiere este proyecto, es que la búsqueda de rutas se realice un periodo relativamente corto. Existe un riesgo de que el tiempo de ejecución del algoritmo sobrepase lo tolerable, en caso de que ocurra, tendré el desafío de reducir el tiempo sin que se vea alterado el algoritmo, buscando herramientas alternativas.
- **R4:** El desarrollo de este proyecto se va a realizar mientras curso el segundo cuatrimestre de tercero y el primer cuatrimestre de cuarto, por lo que una carga excesiva de trabajo puede retrasar el proyecto.

2.3. Seguimiento y evaluación

Durante el transcurso del proyecto, se han tenido que ir adaptando pequeños cambios al mismo, los cuales no estaban previstos al inicio del proyecto.

Uno de los mayores problemas encontrados, es el recogido en el R2, ya que una vez desarrollado el algoritmo, en los testeos posteriores, se comprobó que el tiempo que tardaba en arrojar un resultado era excesivo, por lo que se tuvieron que buscar alternativas y realizar nuevos estudios para intentar llegar al objetivo.

Otro de los problemas que se encontraron, es que los grafos que se utilizan para este proyecto provocaban que el algoritmo se atasque en algunos casos, este caso se identificó en R2. Para poder solventar este problema, se tuvo que adaptar ligeramente el algoritmo ACO.

El último problema que surgió fue el señalado en el riesgo R3. Ya que el algoritmo ACO no se adaptaba bien a los grafos grandes, para ello se pensó en una solución híbrida juntado los algoritmos de Dijkstra y ACO. También se estudió el algoritmo de ramificación y acotamiento, para observar como se adaptaba a esta solución híbrida.

3

Estado del Arte

Los algoritmos de búsqueda de rutas utilizan normalmente grafos ponderados para representar los mapas y los posibles caminos entre los posibles puntos del mapa. Los grafos se componen de nodos y aristas. Cada nodo representa un punto en el mapa, y las aristas son el camino marcado entre cada par de nodos. Los algoritmos de búsqueda de rutas exploran rutas desde un nodo de partida hasta un nodo de destino, atravesando las aristas y otros nodos. Los pesos de las aristas pueden incidir la distancia entre los dos puntos del mapa representados por los nodos, o el tiempo necesario para trasladarse de un punto a otro. Esta información se utiliza para determinar el camino más adecuado.

La búsqueda de rutas es uno de los problemas más estudiados en el campo de la investigación operativa. En muchas ocasiones se da la situación de que se necesita la mejor ruta para realizar determinada actividad. Teniendo en cuenta que los grafos son el medio utilizado para representar la información, se definirán las estructuras de datos mas habituales para codificar los grafos, así como algunos teoremas y técnicas necesarias para comprobar que los caminos que se quieren obtener existen o son viables. Dada la complejidad de la tarea, y la necesidad de que la solución se ejecute en "tiempos razonables" se presenta la teoría necesaria para medir la complejidad de las soluciones estudiadas. A continuación, se muestran las principales técnicas para abordar problemas parecidos al abordado en este proyecto, para finalizar con un apartado de conclusiones donde se presenta la propuesta que se implementará en este proyecto.

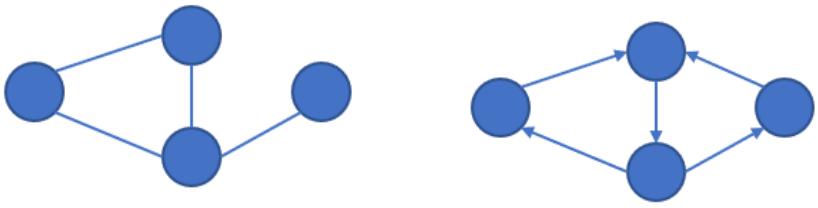
3.1. Grafos

Teniendo en cuenta que los grafos constituyen el mecanismo fundamental para la representación de mapas y el cálculo de rutas, en este capitulo se definirán los grafos desde el punto de vista matemático y se presentarán teoremas y algoritmos, así como las formas de representar los grafos y los teoremas fundamentales que proporcionan herramientas para el cálculo de caminos entre dos nodos del grafo.

Definición de grafo: Un grafo es un par $G = (V, A)$ donde V es un conjunto de nodos (también conocidos como vértices) y A es el conjunto de aristas o arcos entre pares de nodos (Cirre, 2004, p 126). Dos vértices o nodos a y b son adyacentes si están unidos por una arista, es decir, si $\{a, b\} \in A$.

En este proyecto se va a trabajar con grafos simples y conexos, los cuales pueden ser o no dirigidos, ver Figura 3.1. Las características de dichos grafos son las siguientes:

- El número de nodos es finito.
- No hay aristas paralelas.
- No hay bucles, es decir no existe ninguna arista que comience y termine en el mismo nodo.
- No pueden existir nodos aislados.
- Si es un grafo dirigido la arista indica de que nodo a que nodo se puede ir. En un grafo dirigido se puede ir de cualquiera de los dos nodos que une la arista al otro nodo.



(a) Grafo no dirigido **(b)** Grafo dirigido

Figura 3.1 – Grafos simples y conexos

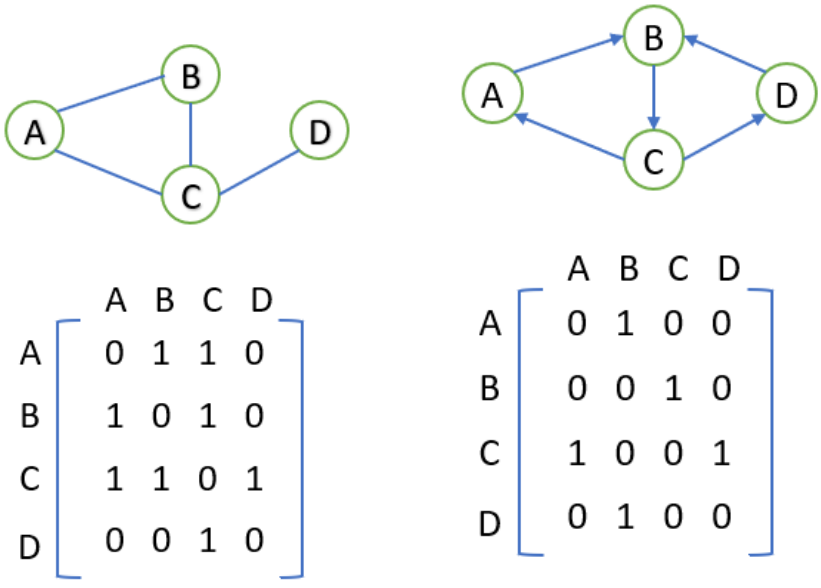
El grado de un nodo en un grafo no dirigido es el número de aristas incidentes en él (Veerarajan et al., 2008, p 367). Si el grado de un nodo es uno, éste se llama nodo colgante.

3.1.1. Representación matricial de grafos

Cuando se realizan algoritmos sobre grafos se utilizan representaciones matriciales. A continuación se muestran las distintas alternativas para representar los grafos.

Definición de matriz de adyacencia: Sea $G = (V, E)$ con $|V| = n$ y $V = v_1, v_2, \ldots, v_n$. La matriz de adyacencia de G, respecto a los vértices anteriores, es una matriz binaria nxn, A, cuyo elemento i, j vale 1, representando a verdadero, cuando v_i es adyacente a v_j y 0, representando a falso, cuando no lo es. Es decir, si $A = [a_{ij}]$ es la matriz de adyacencia, entonces

$$a_{ij} = \begin{cases} 1 & \text{si } \{v_i, v_j\} \text{ es un lado de G} \\ 0 & \text{en otro caso} \end{cases}$$



(a) Matriz de adyacencia en grafo no dirigido

(b) Matriz de adyacencia en grafo dirigido

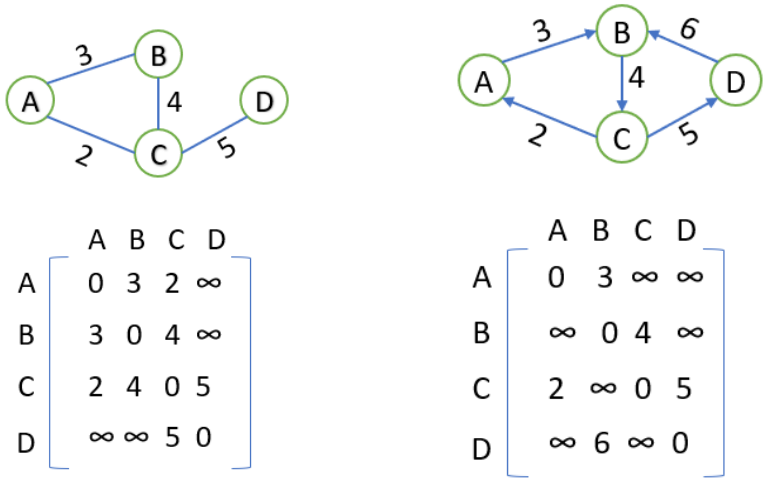
Figura 3.2 – Matrices de adyacencia en grafos

La Figura 3.2 muestra como se representa un grafo simple y un grafo dirigido. Si el grafo es simple (no dirigido), $a_{ij} = a_{ji}$, su matriz de adyacencia es simétrica, (García Merayo, 2005, p 384), en cambio si es un digrafo (dirigido), entonces $a_{ij} \neq a_{ji}$.

Por otra parte, al no tener bucles, aristas que tienen como origen y destino el mismo nodo, los grafos con los que se van a trabajar, aristas que tienen como origen y destino el mismo nodo, será $a_{ii} = 0, \forall\ i = 1, 2, \ldots, n$.

Los grafos cuyas aristas tienen información o peso, se les llama grafos ponderados. La información o peso de las aristas puede ser la distancia entre dos puntos, el tiempo de trayecto, el costo en dinero... En los grafos ponderados se puede sacar la matriz de pesos (Veerarajan et al., 2008, p 406). El valor de $A_{i,j}$ de la matriz de pesos será el peso que tenga la arista en caso de tenerla, y infinito en caso de no tenerla. Cuando se quiere ir de un nodo a ese mismo nodo, el coste es cero, ya que ya te encuentras en ese nodo. Se puede observar un ejemplo en la Figura 3.3.

$$a_{ij} = \begin{cases} peso\ de\ la\ arista & \text{si } \{v_i, v_j\} \text{ es un lado de G} \\ \infty & \text{en otro caso} \end{cases}$$



(a) Matriz de pesos en grafo no dirigido

(b) Matriz de pesos en grafo dirigido

Figura 3.3 – Matrices de pesos en grafos

3.1.2. Conectividad en grafos

Definición: Sea $G = (V, E)$ un grafo, se dice que G es *conexo* si existe un camino elemental entre cualquier par de nodos, (García Merayo, 2005, p 351). En

caso contrario, recibe el nombre de *grafo no conexo* o *disjunto*. Un digrafo o grafo dirigido es conexo si su grafo subyacente también lo es.

En un grafo puede haber más de una trayectoria entre dos nodos cualesquiera. El número de trayectorias entre dos nodos cualesquiera de un grafo puede encontrarse analíticamente utilizando la matriz de adyacencia de dicho grafo, para ello hay que aplicar el siguiente teorema.

Teorema: Si A es la matriz de adyacencia de un grafo G, entonces el número de trayectorias diferentes de longitud r de v_i y v_i es igual a la entrada $(i-j)$-ésima de A^r (Veerarajan et al., 2008, p 391).

Por ejemplo, el número de caminos compuestos por 3 aristas entre todo par de nodos del grafo mostrado en la Figura 3.2a se puede obtener de la siguiente manera.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}^3 = \begin{pmatrix} 2 & 3 & 4 & \mathbf{1} \\ 3 & 2 & 4 & 1 \\ 4 & 4 & 2 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix}$$

En el resultado se puede observar que según el valor de $R_{a,d}$, solo existe un camino de tres pasos entre el nodo A y el nodo D, por lo que el camino está compuesto por tres aristas y 4 nodos.

Uno de los requisitos que tiene este proyecto es que el grafo tiene que ser conexo. Desde cualquier nodo se tiene que poder llegar a cualquier otro nodo. Para verificar que se tiene un grafo conexo se puede utilizar el teorema anteriormente descrito. Se puede llegar a la conclusión de que si se calculan todas las trayectorias posibles hasta una longitud del número total de nodos del grafo menos 1, y las sumamos entre sí. Se calcula la longitud del número total de nodos menos 1, ya que la longitud viene dada por el número de aristas. Entre un par de nodos el camino mínimo lo compone solo una arista. Entre n nodos el camino mínimo será $n-1$ aristas. Si la matriz resultante tiene ausencia de valores nulos exceptuando la diagonal principal, se puede decir que el grafo es fuertemente conexo. Para quitar los valores nulos de la diagonal principal se puede sumar la matriz identidad, y tendríamos como resultado la siguiente Fórmula 3.1.

$$T = I + A + A^2 + \cdots + A^{n-1} \tag{3.1}$$

Donde I es la matriz identidad, A la matriz de adyacencia y n el número total de nodos en el grafo. Si todos los valores de T son diferentes a 0, el grafo es fuertemente conexo, en caso contrario será un grafo no conexo.

3.1.3. Ejemplo de representación de un mapa con un grafo.

Veamos un ejemplo sencillo en que se utiliza un grafo para representar un mapa y los caminos existentes entre las distintas ciudades del mapa. En el mapa hay cuatro ciudades, Vitoria, Bilbao, Pamplona y San Sebastian, que representan los nodos. También hay cuatro carreteras de doble sentido que representan aristas bidireccionales. El mapa se puede ver en la Figura 3.4.

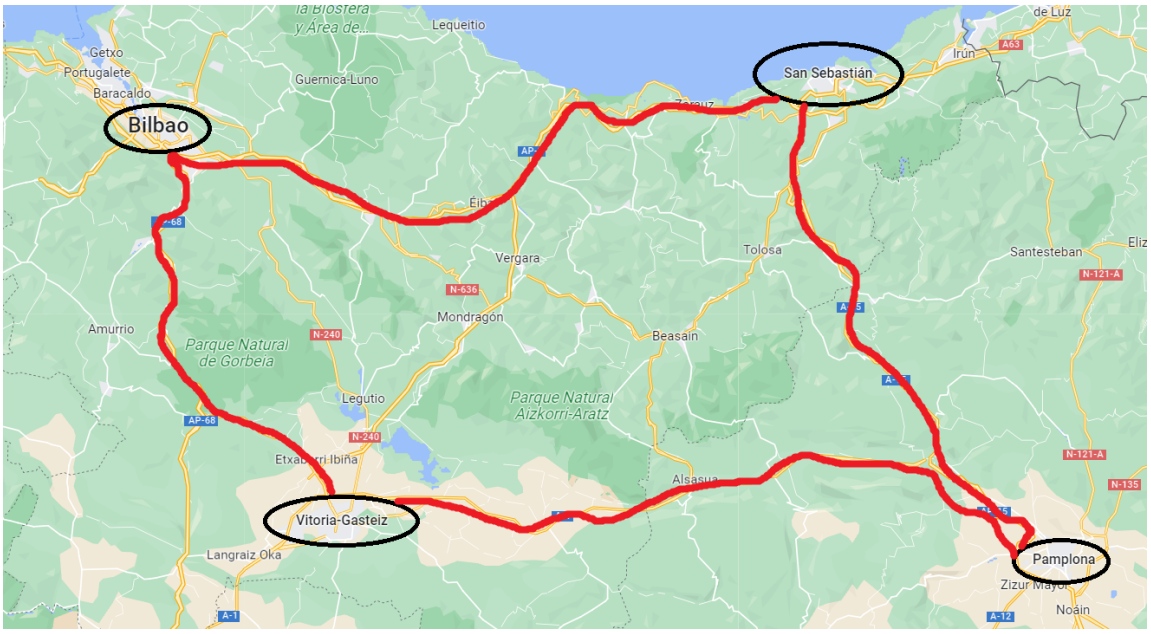


Figura 3.4 – Ejemplo de un grafo en un mapa[1]

Tanto en la matriz de adyacencia como en la matriz de pesos cada fila y cada columna representan una ciudad. En el ejemplo mostrado, la fila y columna cero representan a Vitoria, la uno a Bilbao, la dos a Pamplona y la tres a San Sebastian.

La matriz de adyacencia del grafo resultante estaría representada en la Matriz 3.2. En dicha matriz se puede observar si existe carretera entre dos ciudades. Por ejemplo, la fila cero que representa a Vitoria, indica que existe camino directo hacia Bilbao y Pamplona, en cambio no se podría ir a San Sebastian de manera directa, habría que estudiar si se podría llegar pasando por otras ciudades.

[1] https://www.google.es/maps

$$M_{Adyacencia} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \tag{3.2}$$

La matriz de pesos del grafo estaría representada en la Matriz 3.3. En esta matriz se puede observar el peso que tiene cada arista. En este ejemplo el peso representa la distancia entre dos ciudades. Por ejemplo, en la fila dos se puede observar que para ir de Pamplona a Vitoria, sin pasar por ninguna otra ciudad, hay que recorrer una distancia de 94.8 kilómetros. Para ir de Pamplona a San Sebastian habría que recorrer 83.1 kilometros. Por último para ir de Pamplona a Bilbao, no habría camino camino directo.

$$M_{Pesos} = \begin{pmatrix} 0 & 64{,}5 & 94{,}8 & 0 \\ 64{,}5 & 0 & 0 & 101{,}4 \\ 94{,}8 & 0 & 0 & 83{,}1 \\ 0 & 101{,}4 & 83{,}1 & 0 \end{pmatrix} \tag{3.3}$$

En último lugar, para saber si el grafo del mapa es conexo, aplicamos la Fórmula 3.1.

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}^2 + \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}^3 = \begin{pmatrix} 3 & 5 & 5 & 2 \\ 5 & 3 & 2 & 5 \\ 5 & 2 & 3 & 5 \\ 2 & 5 & 5 & 3 \end{pmatrix}$$

Como se puede observar en los cálculos realizados, la matriz T no tiene ningún valor nulo, por lo que el grafo es conexo.

3.2. Complejidad de los algoritmos.

Para determinar la eficiencia de un algoritmo hay que calcular el tiempo de CPU requerido al implementarlo sobre una computadora específica. Para calcular dicho eficiencia vamos a introducir dos definiciones. Sea $T(n)$ el Tiempo de Ejecución para el peor caso del algoritmo.

- Si existe una constante positiva k tal que

$$T(n) == (n^k) \tag{3.4}$$

Entonces se dice que el algoritmo es de hecho un **Algoritmo de Tiempo Polinomial**.

- Por otro lado, si para toda constante positiva k se tiene que

$$T(n) = \Omega(n^k) \tag{3.5}$$

Entonces se dice que el algoritmo es de **Tiempo Superpolinomial**.

Los algoritmos de tiempo superpolinomial tienen unas funciones de tiempo de ejecución con un crecimiento explosivo, aún con tamaños de entrada pequeños. Para ver esa diferencia, se pueden poner dos ejemplos, un algoritmo de tiempo polinomial y otro algoritmo de tiempo superpolinomial.

- **Ejemplo de algoritmo de tiempo polinomial:** Se tiene un algoritmo A que en el peor de los casos tiene un tiempo de ejecución $T_A(n) = n \log_2 n$, considerando el tamaño de la entrada de $n = 45$, la salida llevaría alrededor de 0.0002 segundos.

- **Ejemplo de algoritmo superpolinomial:** Se tiene un algoritmo B, el cual resuelve el mismo problema que el algoritmo A, usando la misma computadora, pero su tiempo de ejecución es de $T_A(n) = 2^n$ para el peor de los casos, teniendo la misma entrada que en el ejemplo anterior, la salida llevaría unos 407 días.

Un algoritmo de tiempo polinomial es llamado también Algoritmo Eficiente o Razonable, (Pérez Aguila, 2012, p. 28). En cambio, un algoritmo de tiempo superpolinomial es llamado Algoritmo Ineficiente.

3.2.1. Teoría de los problemas NP-completo

En primer lugar se va a definir que es P y NP en la teoría de la complejidad.

- **NP:** (Polinomial no-determinista) Es el conjunto de problemas que se puede comprobar en un tiempo razonable si una respuesta al problema es correcta o no. No se sabe si estos problemas se pueden resolver en orden polinomial. Es una de las incognitas de las matemáticas.

- **P:** (Polinomial) Conjunto de problemas en los que podemos encontrar una respuesta al problema en un tiempo razonable. Este tipo de problemas se pueden resolver en un orden polinomial.

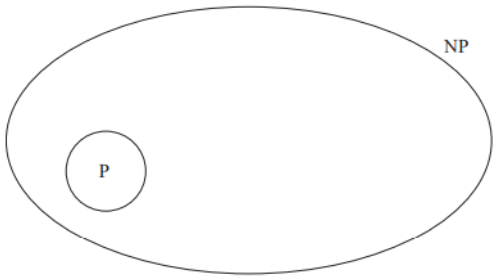


Figura 3.5 – Conjunto de problemas NP

También se puede decir que todos los problemas en P están en NP, pero no al revés, Figura 3.5.

Los problemas NP-completos son un extenso tipo de problemas que en apariencia carecen de algoritmos de tiempo polinomial que los resuelvan. El conjunto de estos problemas es muy grande, he incluye problemas como el que nos ocupa, el del agente viajero, tal como se muestra en la Figura 3.6. Todos estos problemas poseen una característica común: hasta ahora, en el peor de los casos, ningún problema NP-completo puede resolverse con un algoritmo polinomial, hasta la fecha. El mejor algoritmo para resolverlos tiene una complejidad exponencial. Por la definición de la teoría de problemas NP-completos, (Lee et al., 2014, p 321), se cumple lo siguiente: Si un problema NP-completo puede resolverse en tiempo polinomial, entonces todos los problemas NP pueden resolverse en tiempo polinomial. O bien, si cualquier problema NP-completo puede resolverse en tiempo polinomial, entonces NP=P.

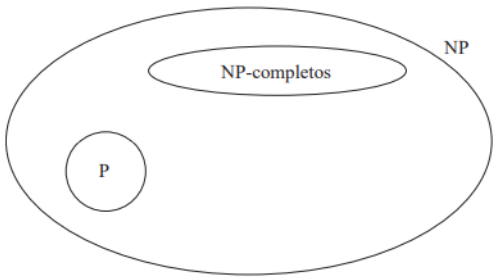


Figura 3.6 – Conjunto de problemas NP-completos

3.3. TSP. Problema del agente viajero.

El origen del problema del TSP (ver Figura 3.7) no se conoce con certeza. Ya a principios del siglo 19 apareció un libro sobre un comerciante alemán. Este libro era una guía para viajantes de 1832, el cual menciona el problema e incluye ejemplos de viajes a través de Alemania y Suiza, pero no contiene un tratamiento matemático del mismo. Poco a poco, el problema del TSP fue cogiendo relevancia entre la comunidad científica, hasta hacerlo uno de los problemas más interesantes y más estudiados.

Figura 3.7 – German Salesman Book, del año 1832, foto obtenida de (Cook, 2012)

El problema del agente viajero se puede explicar con un suceso que empezó con una campaña publicitaria de Procter & Gamble. La compañía lanzaba un concurso con un premio de $10.000 para calcular la ruta más corta visitando 33 ciudades siendo el punto de partida y el destino la misma ciudad. Este anuncio causo un gran debate entre los matemáticos de la época en la primavera de 1962. La campaña publicitaria decía lo siguiente.

> Imagina que Toody y MUldoon quieren conducir por todo le país y visitar cada uno de los 33 lugares representados por puntos en el mapa del concurso, y desean viajar por la ruta más corta posible. Se debe planificar una ruta que resulte la de menor kilometraje, comenzando en Chicago y acabando en el mismo punto.

Para resolver este problema por fuerza bruta, el número de posibles caminos sería 33!, pero hay que tener en cuenta dos puntos. Primero, se sale y se acaba en la misma ciudad, la cual ya está definida, está característica reduce el calculo a "32!". Segundo, la distancia entre la ciudad A y la ciudad B es la misma que entre B y A, lo que nos deja el calculo en 32!/2. La cantidad de posibles soluciones es de:

131.565.418.466.846.765.083.609.006.080.000.000

Para comprobar todas y cada una de las posibles soluciones posibles, llevaría una cantidad ingente de tiempo que haría que la solución no sea factible. Si el problema

Figura 3.8 – Imagen de la campaña publicitaria de Procter & Gamble, foto obtenida de (Cook, 2012)

se generaliza, el crecimiento del mismo sería exponencial, lo que lo haría inasumible poder abordarlo, (Cook, 2012, p 1).

	n = 10	n = 25	n = 50	n = 100
n^3	0.000001 segundos	0.000002 segundos	0.0001 segundos	0.001 segundos
2^n	0.000001 segundos	0.03 segundos	13 días	40 billones años

Tabla 3.1 – Tiempo de ejecución para una computadora de 10^9 operaciones por segundo.

El problema del TSP, extrapolado a la teoría de grafos pide hallar el circuito hamiltoniano de menor coste. El nombre de *circuito hamiltoniano* se debe al matemático irlandés *William Rowan Hamilton*, que estudió este tipo de circuitos en grafos. La definición formal de un circuito hamiltoniano es la siguiente.

Definición: Sea $G = (V, E)$ un grafo con $|V| \geq 3$, sin nodos aislados. Recibe el nombre de camino de Hamilton en G, todo camino elemental de G que contiene todos sus nodos. Se dice que G posee un ciclo de Hamilton, si existe un ciclo que contenga todos los nodos de V. Entonces G es hamiltoniano.

No existen condiciones necesarias y suficientes para encontrar caminos o ciclos hamiltonianos en un grafo, (García Merayo, 2005, p 402), por lo que le problema de concluir si un grafo es o no hamiltoniano puede llegar a ser complicado, sobre todo si se trata de grafos grandes.

Los circuitos hamiltonianos en grafos grandes no son sencillos de encontrar, pero en el TSP no solo se pide un circuito hamiltoniano, sino que tiene que ser el de menor coste. El problema del TSP está catalogado como un problema np-completo.

3.3.1. Soluciones al TSP

Alguna de las soluciones posibles del TSP son:

- Soluciones de fuerza bruta. Es decir, hacer una lista de todas las posibles soluciones al problema y calcular el coste de cada una de ellas, y por comparación, buscar la solución más óptima.

- Métodos exactos. También llamados algoritmos óptimos, se intenta descartar familias enteras de posibles soluciones, tratando así de reducir el tiempo de búsqueda de la solución. Uno de los que más se usa para resolver el TSP es Ramificación y Acotamiento.

- Métodos heurísticos. Son métodos que obtienen buenas soluciones en tiempos de computo relativamente cortos, aunque no garantizan la mejor solución. Algunas de la soluciones heurísticas son:

  - Algoritmos genéticos, genetic algorithm (GA), consiste en encontrar un individuo rutaçuya combinación de genes (caga gen es una variable, es decir una ciudad), de una solución óptima. El orden de complejidad de estos algoritmos es de $O(n^2)$.
  - Redes neuronales, simulan las conexiones entre nodos (lugares por visitar), y cada recorrido por las diferentes neuronas genera una solución con todas las ciudades visitadas solo una vez. Con las redes neuronales se puede llegar a alcanzar un orden de complejidad de $O(n^2 \log n)$.
  - Colonia de hormigas, Ant Colony Optimization (ACO), las hormigas generan el camino entre dos puntos, para TSP el punto de inicio y el de fin serán el mismo, de esta manera, las hormigas deben recorrer todas las ciudades en un circuito, sin repetir ciudad. Al igual que en los algoritmos genéticos el orden de complejidad es $O(n^2)$.
  - Búsqueda Tabú, Tabu Search (TS), consiste en buscar el vecino próximo cuyos costos de traslado del nodo actual al siguiente, sea el de nenor costo en cuanto al uso de recursos. Su orden de complejidad es de $O(n^2)$.

Esta información ha sido obtenida de (Fuentes Penna, 2014).

3.4. Algoritmos para el cálculo de mejores rutas entre dos puntos del mapa

El algoritmo de *Dijkstra* es uno de los más usados con grafos para encontrar el camino más corto de un nodo a todos los demás, (Jordán Lluch et al., 2022, p 75). Este algoritmo solo es válido para un grafo con una matriz de pesos con todos los valores positivos. Existen otros algoritmos como el de *Bellman-Ford* o el de *Floyd-Warshall* que la matriz de pesos puede tener tanto valores positivos como negativos. Estos dos últimos no se van a estudiar ya que las matrices de pesos que nos van a ocupar en este trabajo van a tener como valor las distancias entre nodos, y estas no pueden ser negativas.

Cuando se aplica el algoritmo a un nodo u_0, se obtiene una matriz de distancias $L = (l_ij)_{1*n}$, donde n es el número de nodos. El valor $l_ij = l(v_j)$, es la longitud del camino mas corto desde el nodo u_0 al nodo v_j. También se crea una matriz de nodos,

$A = (a_{1j})_{1*n}$, en la que $a_{1j} = a(v_j) =$ al nodo anterior al v_j en el camino más corto de u_0 a v_j.

Para la ejecución del algoritmo, se dan los siguientes supuestos.

- Sea $G = (V, E)$ en un grafo ponderado positivo con matriz de pesos W, donde v son los nodos y E las aristas.
- Sea u_0 el nodo de origen.
- Sea $l(u)$ una etiqueta asignada a u que almacena los pesos de los caminos de u_0 a u, proporcionando al final del algoritmo el peso del camino más corto entre estos dos nodos.
- Sea $a(u) =$ nodo padre de y en la ruta más corta de u_0 a u
- T es el conjunto que va a contener los nodos para los que el camino mínimo se considera fijado y $\bar{T}$ al conjunto de los nodos etiquetados temporalmente

Ahora se definen los pasos para ejecutar el algoritmo.

- Paso 1. Inicialización.
  $l(u_0) := 0$, $l(u) := \infty \; \forall u \neq u_0$, $T = \{u_0\}$, $\bar{T} = \{u_0\}$, $s := u_0$, $k := 1, a(u_0) := u_0$.
- Paso 2. Actualización de etiquetas. Para cada u de $T(s) \cap \bar{T}$ se debe hacer.
  1. Si $l(s) + w(s, u) < l(u)$, entonces $a(u) := s$.
  2. $l(u) := \text{mín}\{l(u), l(s) + w(s, u)\}$.
- Paso 3. Etiqueta fija.
  1. Se localiza un nodo u' tal que $l(u') := \text{mín}_{u \in \bar{T}}\, l(u)$
  2. $T := T \cup \{u'\}$, $\bar{T} := \bar{T} - \{u'\}$, $s := u'$, $k := k + 1$.
- Paso 4. Test de finalización.
  - Si solo se quiere el camino más corto de u_0 a u_i, si $s \neq u_i$, hay que volver al paso 2, y, en caso contrario, fin.
  - Si se quiere el camino más corto de u_0 a cualquier u_i, si $k < |V|$, se debe repetir el paso 2, y, en caso contrario, fin.

A continuación se muestra como se utiliza el algoritmo de Dijkstra para obtener la mejor ruta entre dos puntos sobre un grafo ponderado y dirigido mostrado en la Figura 3.9. El grafo consta de 6 nodos y 15 aristas $G(6, 15)$. Se van a calcular los caminos más cortos desde el nodo C al resto de nodos.

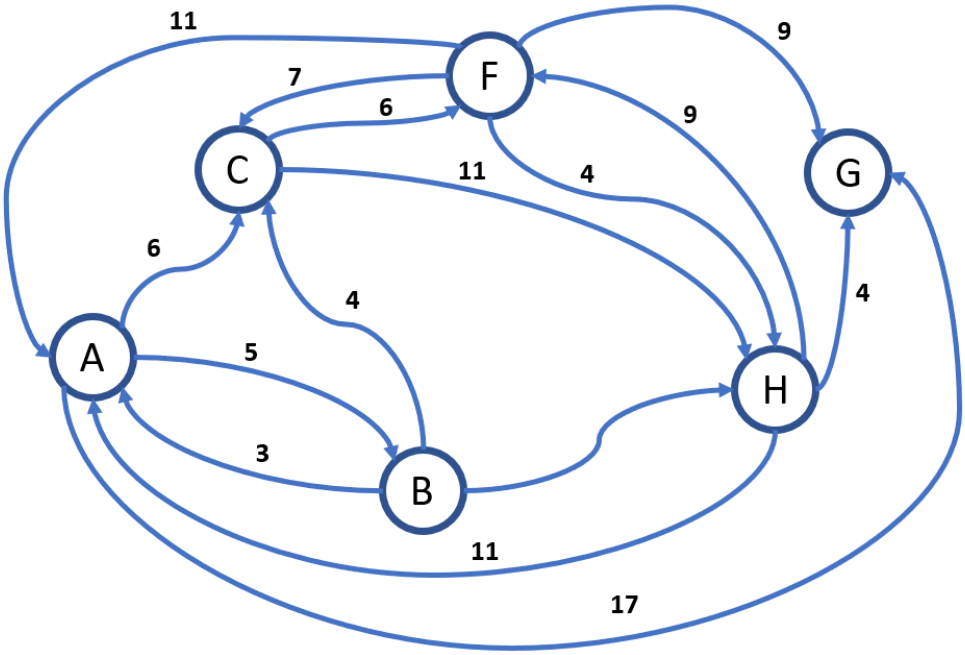


Figura 3.9 – Grafo dirigido y ponderado.

Lo primero que se debe hacer es sacar la matriz de pesos como se ha explicado en la Sección 3.1.1.

$$A = \begin{pmatrix} 0 & 5 & 6 & \infty & 17 & \infty \\ 3 & 0 & 4 & \infty & \infty & 7 \\ \infty & \infty & 0 & 6 & \infty & 11 \\ 11 & \infty & 7 & 0 & 9 & 4 \\ \infty & \infty & \infty & \infty & 0 & \infty \\ 11 & \infty & \infty & 9 & 4 & 0 \end{pmatrix} \tag{3.6}$$

Con la matriz de pesos se puede ver claramente que el grafo no es conexo. Desde el nodo G no se puede ir a ningún lado.

El siguiente paso es ejecutar el algoritmo de Dijkstra, se puede ver en la Tabla 3.2.

Iteración	Nodo visitado	A	B	C	F	G	H	Camino	Peso
0	–	$(\infty,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$	(C)	0
1	C	$(\infty,-)_1$	$(\infty,-)_1$	–	$(6,C)_1$	$(\infty,-)_1$	$(11,C)_1$	(C,F)	6
2	F	$(17,F)_2$	$(\infty,-)_2$	–	–	$(15,F)_2$	$(10,F)_2$	(C,F,H)	10
3	H	$(17,F)_3$	$(\infty,-)_3$	–	–	$(14,H)_3$	–	(C,F,H,G)	14
4	G	$(17,F)_4$	$(\infty,-)_4$	–	–	–	–	(C,F,A)	17
5	A	–	$(22,A)_5$	–	–	–	–	(C,F,A,B)	22
6	B	–	–	–	–	–	–		

Tabla 3.2 – Ejecución matemática del algoritmo de Dijkstra.

Después de la ejecución del algoritmo de Dijkstra, se han obtenido todos los caminos mas cortos desde el nodo C hasta el resto de los nodos, los resultados se pueden ver en la Tabla 3.3.

Origen	Destino	Ruta	Distancia
C	F	$C-F$	6
	H	$C-F-H$	10
	G	$C-F-H-G$	14
	A	$C-F-A$	17
	B	$C-F-A-B$	22

Tabla 3.3 – Resultados del algoritmo de Dijkstra.

3.5. Algoritmos de optimización

Los algoritmos que existen para buscar buenos resultados son muchos, y hay toda una ciencia detrás de ello. Existen algoritmos exactos, estos algoritmos tiene un coste alto, para reducir el tiempo de computo surgieron los algoritmos heurísticos.

Cuando el problema que se tiene entre manos requiere que se garantice encontrar la mejor respuesta posible, entonces hay que recurrir a algoritmos exactos o de búsqueda exhaustiva. Un ejemplo claro sería si se pide encontrar los número primos entre A Y B, siendo A y B dos números enteros positivos, no quedaría otra solución que mirar cada número comprendido entre A y B y ver si es primo.

Para problemas del tipo NP, no se conocen algoritmos exactos con tiempos de convergencia en tiempo polinómico. Por lo que aunque exista un algoritmo exacto que encuentre la solución, tardaría tanto tiempo en encontrarla que lo hace completamente inviable. Debido a este motivo es necesaria la utilización de algoritmos aproximados o heurísticos que permitan obtener una solución de calidad en un tiem-

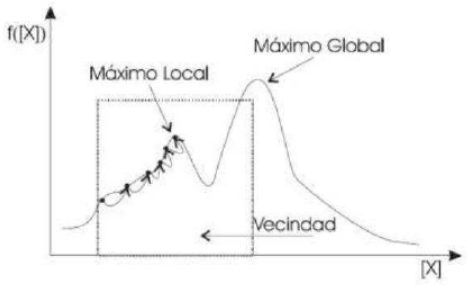


Figura 3.10 – Encasillamiento de algoritmos heurísticos en óptimos locales, imagen obtenida de (Vidal, 2013)

po razonable. El término heurística proviene del vocablo griego *heuriskein*, que puede traducirse como encontrar, descubrir o hallar.

El principal problema de los algoritmos heurísticos es el de quedarse atrapados en óptimos locales, (Vidal, 2013). Por ejemplo como se muestra en la Figura 3.10, para la vecindad dada el algoritmo heurístico basado en un método de búsqueda local se quedaría atrapado en un máximo local sin poder alcanzar el máximo global.

Por dicho problema, los algoritmos heurísticos evolucionaron para evitar en la medida de lo posible quedar atrapados en óptimos locales. Estos algoritmos incorporaron la memoria de los caminos recorridos, para penalizar o premiar las rutas.

A continuación se van a explicar algunos de los métodos más conocidos de resolución de problemas de optimización.

3.5.1. Búsqueda exhaustiva

Consiste, tal y como indica su nombre, en evaluar todas las soluciones posibles hasta encontrar la mejor solución global. Por lo que si no se conoce el resultado de la solución global, no queda otro remedio que examinar todo el espacio de búsqueda. Si el espacio de búsqueda es muy grande se necesita mucho tiempo computacional para analizar cada una de las soluciones.

Los algoritmos exhaustivos son interesantes para dar solución a algunos problemas por su simplicidad, ya que su único requisito es generar sistemáticamente cada solución posible. También existen métodos para agilizar el algoritmo, como por ejemplo las técnicas de *backtracking* o retroceso, y *branch and bound* o ramificación y acotamiento, (Vidal, 2013).

Para examinar el espacio de búsqueda se transforma el grafo en un grafo de tipo árbol. El árbol es un grafo conexo sin ningún circuito, (Veerarajan et al., 2008, p 415).

3.5.2. Búsqueda local

La búsqueda local es la base de muchos métodos usados en problemas de optimización. Una búsqueda local es un proceso, que dada una solución inicial , selecciona iterativamente una solución de su entorno para continuar la búsqueda, (Vidal, 2013). Es decir, empieza con una solución inicial y busca en la vecindad una solución mejor, en caso de encontrarla, reemplaza la solución, y continua el proceso de manera iterativa hasta que ya no se pueda mejorar.

El diseño de la vecindad es muy importante para que el algoritmo de buenas soluciones. La vecindad son el conjunto de posibles soluciones que se contemplan en cada punto.

La búsqueda local es un método determinista y sin memoria, donde dada una misma entrada, devuelve una misma salida.

La principal ventaja es que encuentra soluciones de manera muy rápida, y su principal inconveniente es que se puede quedar atrapado en óptimos locales.

3.5.3. Búsqueda tabú

La búsqueda tabú utiliza las técnicas de la búsqueda local, pero añadiendo a la memoria las soluciones que han llevado a la última solución antes de decidir cuál es la siguiente, (Vidal, 2013). Este algoritmo fue desarrollado por Glover en 1986, (Glover, 1986).

El esquema general es que a partir de una solución inicial, se determina una vecina que la mejore hasta llegar a una solución aceptable. Pero en este algoritmo entra en juego el movimiento tabú, el cual impide que dos movimientos se repitan en un periodo de tiempo corto. Para que esto se pueda hacer, hay que registrar los movimientos que han modificado las soluciones anteriores. De ahí el término de memoria a corto plazo.

3.5.4. Recocido simulado

El algoritmo de recocido simulado, *simulated annealing* en inglés, se basa en principios de la termodinámica y el proceso de recocido del acero, (Vidal, 2013). El acero a temperaturas muy elevadas se convierte en una amalgama de líquido en el que las partículas se configuran aleatoriamente. El acero en estado sólido se caracteriza por tener una configuración concreta de mínima energía (mínimo global), para llegar a ese estado hay que enfriarlo lentamente, ya que un enfriado brusco provocaría una configuración distinta (mínimo local).

El recocido simulado introduce una variable de control T a la que se le llama variable de temperatura, que al avanzar en las iteraciones permite empeorar la función objetivo con cierta probabilidad, para poder escapar de este modo de un mínimo local y poder alcanzar el mínimo global.

3.5.5. Algoritmos genéticos

Los algoritmos genéticos se basan en el comportamiento de la genética y de la selección natural, (Vidal, 2013). Trabaja simultáneamente con dos conjuntos de soluciones factibles, cada solución es denominada *individuo*, estos pertenecen a una población y se cruzan entre si. También los individuos pueden llegar a mutar sus genes. Este algoritmo fue introducido por *Holland* en 1975 (Holland, 1992).

Estos algoritmos tiene una métrica llamada función de aptitud, que permite evaluar cuantitativamente cada solución candidata (individuo). Las primeras individuos se generan de manera aleatoria, por lo que tendrán, con mucha probabilidad, una eficiencia mínima con respecto a la resolución del problema. A estos individuos se les permite reproducirse, para ello dos individuos se cruzan entre si, creando un descendiente. El descendiente tendrá genes de los individuos padres, y luego con una probabilidad determinada, podrán mutar algún gen o no, generando nuevas soluciones. Si las soluciones nuevas mejoran las anteriores se conservan, en caso contrario se eliminan (selección natural). Se puede ver la secuencia en la Figura 3.11.

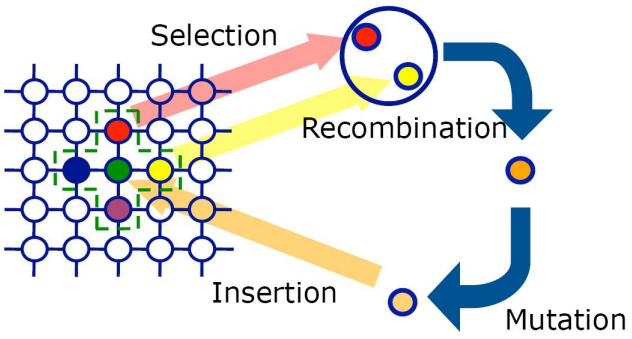


Figura 3.11 – Secuencias del algoritmo genético [2]

[2] http://educagratis.cl/moodle/course/view.php?id=255

3.6. Colonia de hormigas

El algoritmo Ant Colony Optimization (ACO) es una heurística para resolver problemas de optimización combinatoria. Esta basado en el comportamiento que tienen algunas especies de hormigas a la hora de ir en busca de alimentos. Las hormigas van soltando feromonas en el suelo para marcar algún camino favorable, de este modo el resto de hormigas de la colonia lo pueden seguir. El algoritmo fue propuesto en 1992 por *Marco Dorigo*, y capto la atención de muchos investigadores, y en estos momentos hay gran cantidad de aplicaciones exitosas, y variedad de extensiones que hacen que el algoritmo se pueda adaptar mejor a los diferentes problemas.

El primer ejemplo del algoritmo es Ant System (AS), que se propuso para resolver el problema del TSP, pero a pesar de unos resultados iniciales alentadores, no pudo competir con otros algoritmos que se usaban para el mismo problema. Aún así, provocó investigaciones sobre variantes del algoritmo, logrando resultados sensacionales y ampliando el tipo de problemas a los que se le puede aplicar. Algunos de los problemas a los que se le puede aplicar ACO son ordenamiento secuencial, TSP probabilístico, secuenciación de ADN, entre otros.

3.6.1. El origen biológico de ACO

En muchas especies de hormigas, la hormiga puede depositar una feromana (sustancia química que las hormigas pueden oler) en el suelo mientras camina. Al depositar feromonas, las hormigas crean un rastro que es usado, por ejemplo, para marcar el camino desde le hormiguero hasta la fuente de comida y vuelta. El resto de hormigas recolectoras pueden sentir el rastro de feromonas y seguirlo hasta la comida descubierta por otras hormigas. Varias especies de hormigas son capaces de determinar el camino mas corto posible que conduce a la comida por el rastro de feromonas.

Deneubourg et al., 1990 Goss et al., 1989 usaron un puente doble, Figura 3.12, que conectaba un hormiguero con una fuente de alimento, con el objetivo de estudiar el rastro de las feromonas y el comportamiento de las hormigas. Realizaron una serie de experimentos en los que fueron variando la longitud de las dos ramas de los puentes. En el experimento en el que un puente era más largo que el otro, las hormigas al principio elegían de manera aleatoria un puente o el otro, pero según iba pasando el tiempo terminaron usando la rama más corta.

El resultado obtenido del experimento se puede explicar de la siguiente manera. Cuando las hormigas comienzan el camino, no existe ningún rastro de feromonas, por lo tanto no tiene preferencias a la hora de elegir un puente. Se puede esperar que la

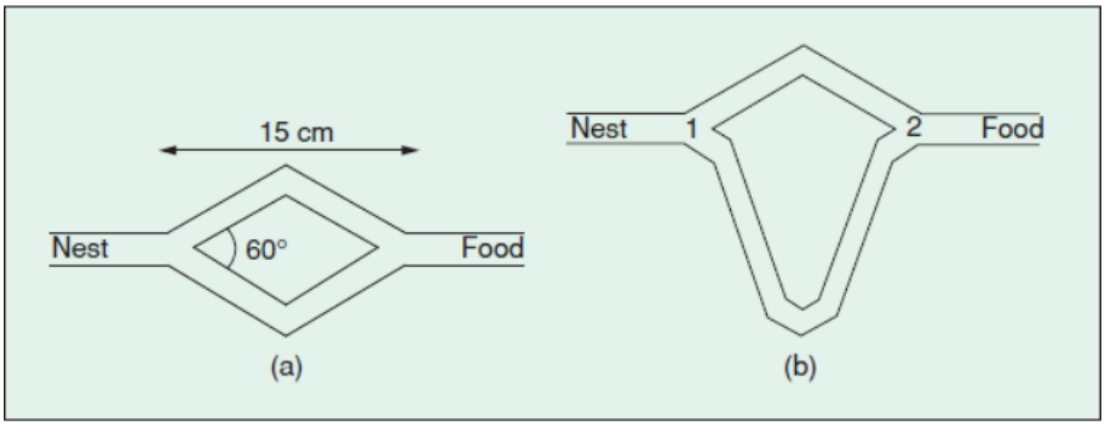


Figura 3.12 – Experimento del puente doble (Dorigo et al., 2006, p. 29). (a) Los dos puentes miden lo mismo. (b) Los puentes tienen diferentes longitudes.

mitad de las hormigas elijan un puente y la otra mitad el otro, aunque en ocasiones, las oscilaciones estocásticas puedan favorecer a uno de los puentes. Sin embargo, como un puente es más corto que el otro, las hormigas que han elegido la ruta corta llegarán antes al hormiguero. Cuando las hormigas vuelvan a salir a buscar comida, al enfrentarse a la decisión de que puente cojer, como el rastro de feromonas del corto es mayor, su decisión se ve sesgada hacía dicho puente. Con este comportamiento, el puente corto, cada vez irá acumulando mayor rastro de feromona, acabando así siendo la opción elegida por la mayoría de las hormigas.

3.6.2. Algoritmos ACO

El algoritmo original de optimización de colonias de hormigas fue AS, el cual se propuso a principios de los noventa, y partir de ahí se propusieron otros alglritmos ACO, en los que todos comparten la misma idea.

Utilizando el problema del TSP, para explicarlo, el algoritmo consiste en crear una cantidad de hormigas artificiales que se van a mover por el grafo del TSP, donde cada nodo es una ciudad y cada arista la conexión entre ciudades. Hay una variable, asociada a cada arista, la cual es llamada feromona, que las hormigas pueden leerla y modificarla.

ACO es un algoritmo iterativo. En cada iteración se sueltan un determinado número de hormigas artificiales. Donde cada una ellas va creando un camino, con la única restricción de no visitar ningún nodo dos veces. En cada paso que da la hormiga

tiene que decidir que nodo visitar, y está decisión está sesgada por la feromona. Al final de cada iteración, se analizan la calidad de los caminos de cada hormiga, y en base a dicha calidad se modifican los valores de las feromonas para influir más en la decisión de las hormigas en iteraciones futuras y se creen nuevas rutas similares a las mejores rutas anteriores.

Una vez explicado el comportamiento general del algoritmo, vamos a ver en detalle el algoritmo original Ant System (AS), y su variante $MAX - MIN$ AS.

3.6.2.1. Ant System (AS)

Ant System es el primer algoritmo ACO propuesto, en el artículo Dorigo et al., 1991. Su principal característica es que en cada iteración, el valor de las feromonas es cambiado por todas las hormigas. La feromona τ_{ij} asociada a la arista que une las ciudades i y j, es actualizada de la siguiente manera:

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \triangle \tau_{ij}^{k} \tag{3.7}$$

Donde ρ es el factor de evaporación de las feromonas, m es el número de hormigas, y $\triangle \tau_{ij}^{k}$ es la cantidad de feromonas depositada en la arista por la hormiga k.

$$\triangle \tau_{ij}^{k} = \begin{cases} Q/L_k & \text{si } (i,j) \text{ ha sido visitado} \\ 0 & \text{en otro caso} \end{cases} \tag{3.8}$$

Donde Q es una constante y L_k es la distancia total de la ruta recorrida por la hormiga k.

La función de probabilidad para que la hormiga k estando en la ciudad i habiendo ya recorrido s^p, vaya a la ciudad j es:

$$p_{ij}^{k} = \begin{cases} \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}} & \text{si } c_{ij} \in N(s^p) \\ 0 & \text{en otro caso} \end{cases} \tag{3.9}$$

Donde $N(s^p)$ es el conjunto de aristas (i,l) donde l es una ciudad no visitada todavía por la hormiga k. Los parámetros α y β controlan la importancia relativa de la feromona versus la información heurística η_{ij}, la cual viene dada por:

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{3.10}$$

donde d_{ij} es la distancia entre las ciudades i y j.

3.6.2.2. $MAX - MIN$ Ant System (MMAS)

El algoritmo MMAS es una mejora hecha sobre el algoritmo AS (Stützle & Hoos, 1996). Los cambios son que sólo las hormigas que han encontrado el mejor camino pueden actualizar los rastros de feromonas, y que dichas feromonas están delimitadas por un máximo y un mínimo.

$$\tau_{ij} \leftarrow [(1-\rho)\cdot\tau_{ij} + \triangle\tau_{ij}^{best}]_{\tau_{min}}^{\tau_{max}} \tag{3.11}$$

Donde τ_{max} es el límite superior y τ_{min} el limite inferior de las feromonas; el operador $[x]_b^a$ es definido como:

$$[x]_b^a = \begin{cases} a & \text{si } x > a \\ b & \text{si } x < a \\ x & \text{en otro caso} \end{cases} \tag{3.12}$$

y $\triangle\tau_{ij}^{best}$ es:

$$\triangle\tau_{ij}^{best} = \begin{cases} 1/L_{best} & \text{si } (i,j) \text{ pertenece a la mejor ruta} \\ 0 & \text{en otro caso} \end{cases} \tag{3.13}$$

Donde L_{best} es la distancia de la mejor ruta encontrada, bien en la iteración actual, bien desde el comienzo del algoritmo o bien una combinación de ambas.

Por otro lado, τ_{max} y τ_{min}, se obtienen de manera empírica y en sintonía con el problema tratado.

Toda la información de esta sección ha sido extraída del libro Gendreau, Potvin et al., 2010 y de los artículos de investigación Dorigo et al., 1991 y Stützle y Hoos, 1996.

3.7. Concluciones sobre el estudio del estado del arte.

El proyecto se basa en encontrar una ruta desde un origen hasta un destino, con la posibilidad de incluir paradas intermedias. En primer lugar se va a probar con un algoritmo de búsqueda exhaustiva, ya que te asegura la mejor solución.

El algoritmo de búsqueda exhaustiva que se va a utilizar es Dijkstra. Este se va utilizar para encontrar la ruta cuando no haya paradas intermedias. Después, se probará con paradas intermedias, adaptando el algoritmo a la nueva situación, y se estudiarán los tiempo dados.

En segundo algoritmo que se va a implementar, como punto de partida, es el de la colonia de hormigas en su variante MMAS. Los motivos de la elección son varios, los cuales se describen a continuación.

- La complejidad de implementación no es demasiado alta.
- La literatura nos dice que, ante el aumento de puntos intermedios en la ruta, el crecimiento del tiempo de computo será lineal.
- La variante MMAS va a proporcionar que la solución no colapse en un mínimo local. Esta variante permitirá escapar del mínimo local, explorando nuevas rutas en busca de mínimo global.

4

Adaptación del grafo para la ejecución de los algoritmos

Cada distribución, tendrá un grafo, el cual estará compuesto por aristas dirigidas, o no dirigidas, que tendrán distribuidos de manera uniforme y cada poca distancia unos *points*, cuya única función es la de poder localizar la arista, Nodos menores, que serán cruces de caminos, y nodos mayores los cuales serán posibles destinos seleccionables para poder navegar hasta ellos. Las aristas tendrán distribuidos de manera uniforme y cada poca distancia unos *points*, que no forman parte del grafo. Los *points* tienen asociados cada uno de ellos unas coordenadas, y se utilizan únicamente para poder localizar la arista.

Por otro lado, las distribuciones tendrán asociadas cada una de ellas unos tags. Los tags son dispositivos de posicionamiento, los cuales en un intervalo corto de tiempo manda su posición. Los tags en los despliegues pueden cumplir dos funciones. Bien la de ser el origen de la ruta, o bien la de ser uno de los puntos intermedios o el destino de la ruta.

Los datos del grafo, están guardados en MySQL, que es una base de datos relacional, y las posiciones de los tags se guardan en MongoDB que es una base de datos no relacional. El código tendrá que comunicarse con ambas base de datos para poder recoger los datos que necesite en cada momento.

Tanto los tags, como el grafo, ya vienen dados del proyecto padre, y se pide lo siguiente. Desde un tag origen, que puede estar en cualquier parte del despliegue, se quiere buscar la ruta hasta un nodo mayor del grafo, o un tag destino, que también puede estar en cualquier lugar del recinto. También se pide poder seleccionar unos nodos mayores y, o, unos tags (paradas intermedias), y buscar la mejor ruta de visita de las paradas intermedias antes de llegar al destino. Dichas paradas no poseen un

orden estricto, por lo que el algoritmo tiene que buscar la mejor manera de ordenar las paradas para que la ruta tenga el menor coste. El nodo origen siempre abrá que añadirlos al grafo original, ya que el origen siempre estará asociado a un tag. Los puntos intermedios y el destino, si son tags abrá que añadirlos al grafo original, en caso contrario, serán nodos mayores que ya están incluidos en el grafo. Un ejemplo del grafo se puede ver en la Figura 4.1.

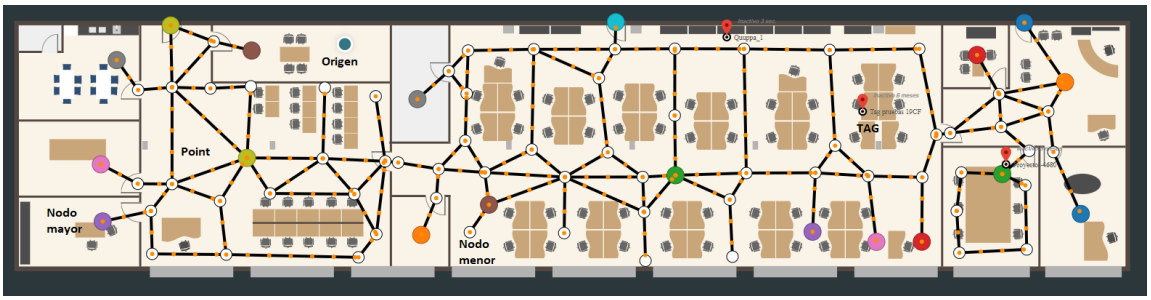


Figura 4.1 – Grafo en una distribución

4.1. Creación de las matrices y diccionarios del grafo

Los datos de los nodos y de las aristas están en una base de datos MySQL. Se tienen que traer esos datos para poder trabajar con ellos. Se construyen dos matrices, y dos diccionarios, ver Figura 4.2.

- **Matriz de pesos con distancias:** Se relacionan los nodos con las distancias que hay entre ellos, siempre y cuando exista una arista que los une. Se tiene en cuenta si la arista es dirigida o simple. Si la arista es simple $d_{ij} = d_{ji}$, en caso de que sea una arista dirigida $d_{ij} \neq d_{ji}$.

- **Matriz de pesos con feromonas:** Se relacionan los nodos con el nivel de feromonas que hay entre ellos, siempre y cuando exista una arista entre ellos. Esta matriz se inicializa con todos los valores a 1.

- **Diccionario para obtener la posición en la matriz del nodo:** Este diccionario sirve para obtener con el Id del nodo, la posición que ocupa en la matriz.

- **Diccionario para obtener el Id del nodo de una posición en la matriz:** Con este diccionario se obtiene el nodo que representa una posición en la matriz.

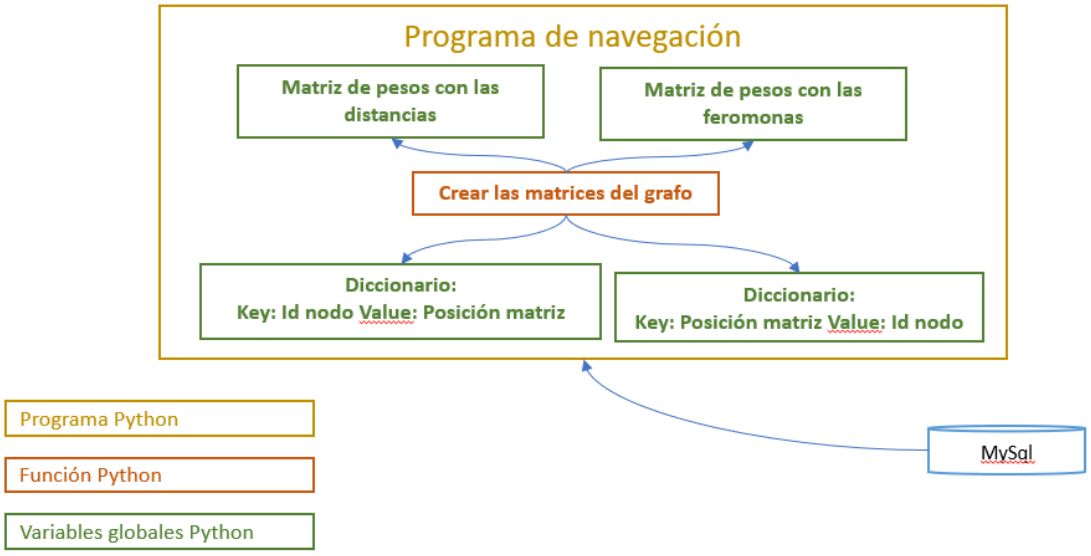


Figura 4.2 – Arquitectura para la creación del grafo

El Algoritmo 4.1 determina como se generan y completan las matrices de pesos y los diccionarios.

Algoritmo 4.1: Crear matrices y diccionarios

Entrada: nodos: lista, aristas: lista, númeroDeNodos: Entero
Salida: idNode_posicionMatriz, posicionMatriz_IDNode, distancias ,feromonas

LÉXICO
idNode_posicionMatriz: Diccionario
posicionMatriz_IDNode: Diccionario
distancias: Matriz de reales
feromonas: Matriz de reales

ALGORITMO

```
// Añadir las posiciones de los nodos a los diccionarios
Para_Cada i Entre 0 Y longitud(nodos) Hacer
    posicionMatriz_IDNode[i] ← nodos[i].id
    idNode_posicionMatriz[nodos[i].id] ← i
// Crear las matrices, una inicializada con ceros y la otra con unos
distancias ← crearMatrizCeros(numFilas = númeroDeNodos, numColumnas =
  númeroDeNodos
feromonas ← 1crearMatrizUnos(numFilas = númeroDeNodos, numColumnas =
  númeroDeNodos
// Añadir las distancias
Para_Cada i Entre 0 Y longitud(aristas) Hacer
    origen ← idNode_posicionMatriz[aristas[i].origen]
    destino ← idNode_posicionMatriz[aristas[i].destino]
    si aristas[i].dirigida == true entonces
        distancia[origen, destino] ← dist
    en otro caso
        distancia[origen, destino] ← dist
        distancia[destino, origen] ← dist
```

A continuación se muestra como se generan las estructuras de datos necesarias para el cálculo de rutas utilizando el algoritmo descrito en el mapa mostrado en la Figura 3.4. Se han representado como nodos las cuatro ciudades, Vitoria, Bilbao, Pamplona y San Sebastian. Se han dibujado unas aristas, que representa la única unión, de manera ficticia, que existe entre ellas.

Los diccionarios y las matrices creados sobre el grafo de la Figura 3.4 quedarían de la siguiente manera:

Listado 4.1 – Diccionarios del grafo

```
idNode_posicionMatriz = {
    'Vitoria': 0,
```

```
    'Bilbao': 1,
    'Pamplona': 2,
    'San Sebastian': 3
}

posicionMatriz_idNode = {
    0: 'Vitoria',
    1: 'Bilbao',
    2: 'Pamplona',
    3: 'San Sebastian'
}
```

Listado 4.2 – Matrices del grafo

```
feromonas = [[1,1,1,1],
             [1,1,1,1],
             [1,1,1,1],
             [1,1,1,1]]

distancias = [[0,64.5,94.8,0],
              [64.5,0,0,101.4],
              [94.8,0,0,83.1],
              [0,101.4,83.1,0]]
```

4.2. Añadir nodos nuevos al grafo

Existe la necesidad de añadir nodos y aristas nuevas al grafo. Esto es debido a que el grafo solo contiene los nodos y las aristas que marcan los caminos principales del mapa. Pero aparte de los nodos que ya contiene el grafo, se puede dar la necesitad de añadir tres tipos de nodos nuevos, los cuales se detallan a continuación.

- **Origen:** El punto de partida puede estar en cualquier lugar del despliegue. Al calcular la ruta, manda las coordenadas de su posición.

- **Tag:** Cuando el destino o una de las paradas es un Tag, este puede estar en cualquier parte del mapa. Se le piden las coordenadas de su localización para añadirlo al mapa.

- **Huecos de picking:** Un hueco de picking en un almacén logístico es el lugar que tiene asignado un artículo para su recogida. Los huecos de picking están definidos en la base de datos como nodos. No están unidos al grafo para aligerarlo. Solo se unen al grafo cuando es necesario ir a un hueco de picking.

Para obtener la posición de los tags, hay que ir a la base de datos de MongoDB. La arquitectura varía un poco respecto a la que se tenía de la función anterior, ver Figura 4.3.

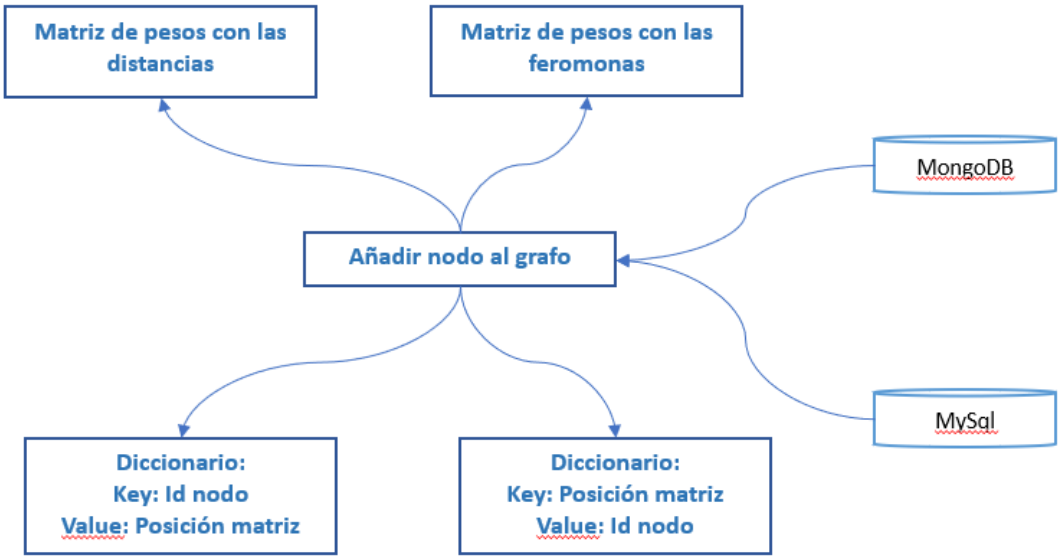


Figura 4.3 – Arquitectura para añadir un nodo

Se va a mostrar de una manera gráfica el proceso de añadir un nodo al grafo. En la Figura 4.4 se puede ver la transformación que hace del grafo cuando el nodo a añadir está próximo a una arista, y en la Figura 4.5 se ve cuando el nodo a añadir está próximo a un nodo del grafo.

El pseudocódigo de añadir un nodo al grafo se muestra en el Algoritmo 4.2.

Algoritmo 4.2: Añadir nodo nuevo al grafo.

Entrada: nodo: objeto
Salida: Las variables globales idNode_posicionMatriz, posicionMatriz_IDNode, distancias ,feromonas son modificadas

LÉXICO

nodosCerca: tupla(nodoA.id: entero) o tupla(nodoA.id: entero,nodoB.id: entero)
puntoArista: tupla(coordX: real,coordY: real,arista.id: entero, esDirigida: booleano)

ALGORITMO

```
// Se añade una fila y una columna en distancias y en feromonas
añadirFilaColumna()
// Se añade el nodo nuevo en los diccionarios
addNodoDicc(nodo)
// Se busca el nodo o la arista más cercano
nodosCerca, puntoArista ← buscarNodoOArista(nodo)
// Si el nodo nuevo está más cerca de una arista
si longitud(nodosCerca)==2 entonces
    // Se borra la arista en distancias
    borrarArista(puntoArista)
    // Se añade el nodo intermedio en los diccionarios
    addNodoDicc(puntoArista)
    // Se añade una fila y una columna en distancias y en feromonas
    añadirFilaColumna()
    // Se añade la arista del nodoInter al nodo nuevo, esta arista
       siempre es bidireccional
    addAristaNoDirigida(nodo, puntoArista)
    // Si la arista es dirigida
    si puntoArista[3] entonces
        // Añadir las dos aristas dirigidas
        addAristaDirigida(nodosCerca[0], puntoArista)
        addAristaDirigida(puntoArista, nodosCerca[1])
    en otro caso
        // Añadir las dos aristas no dirigidas
        addAristaNoDirigida(nodosCerca[0], puntoArista)
        addAristaNoDirigida(puntoArista, nodosCerca[1])
en otro caso
    // Si el nodo nuevo está mas cercano a otro nodo
    addAristaNoDirigida(nodo, nodosCerca[0])
```

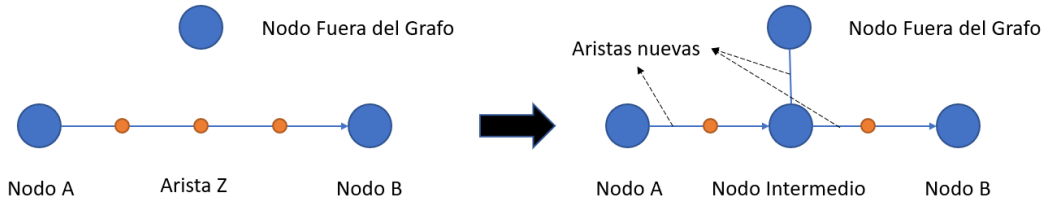


Figura 4.4 – Añadir nodo nuevo desde una arista del grafo

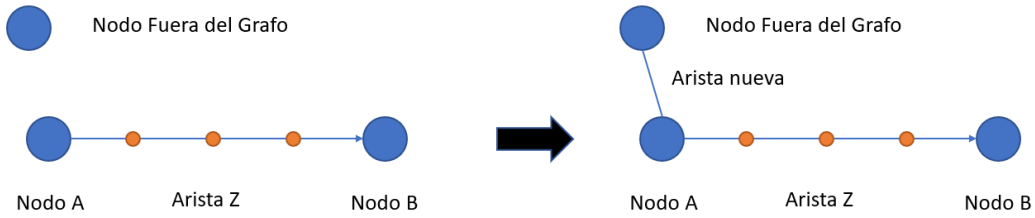


Figura 4.5 – Añadir nodo nuevo desde otro nodo del grafo

4.3. Comprobar conectividad del grafo.

Uno de los requisitos para poder encontrar una ruta en un grafo es que el nodo origen y el nodo destino estén conectados. Es conveniente antes de ejecutar algún algoritmo de búsqueda de rutas comprobar que se puede llegar desde el nodo hasta el destino pasando por los puntos intermedios marcados. Esta comprobación se puede realizar de dos maneras. Bien comprobando que todo el grafo es completamente conexo, es decir que desde cualquier nodo del grafo se puede llegar a cualquier otro. O bien que el nodo origen, el nodo destino, y los nodos de los puntos intermedios están conectados.

Para comprobar la conectividad del grafo se utiliza la propuesta descrita en la Sección 3.1.3. En ella se describe detenidamente como hay que operar la matriz de adyacencia para comprobar la conectividad del grafo. En está sección se presenta el pseudocódigo en el Algoritmo 4.3.

Algoritmo 4.3: Comrobar si el grafo es conexo.

Entrada: matrizAdyacencia: enteros, numeroNodos: entero, son variables globales
Salida: La salida será un booleano, con *true* para grafo conexo y *false* para grafo no conexo

LÉXICO
matrizConexa: matriz de enteros
matrizIdentidad: matriz de enteros
copyMatrizAdyacencia: matriz de enteros

ALGORITMO

```
// Se crea la matriz identidad del orden de la matriz dada
matrizIdentidad ← crearMatrizIdentidad(numeroNodos)
// Se aplica la Fórmula 3.1
matrizConexa ← matrizIdentidad + matrizAdyacencia
// Se crea copia de la matriz de adyacencia
copyMatrizAdyacencia ← matrizAdyacencia
Para_Cada i Entre 0 Y numeroNodos Hacer
    matrizAdyacencia ← matrizAdyacencia * copyMatrizAdyacencia
    matrizConexa ← matrizConexa + matrizAdyacencia
si ningunValorCero(matrizConexa) entonces
    Devolver true
en otro caso
    Devolver false
```

5

Implementación del algoritmo Dijkstra

En este capítulo se describe como se ha utilizado Dijkstra para determinar la ruta entre dos puntos, tanto sin puntos intermedios como con puntos intermedios. Para comprobar la adecuación de esta propuesta para escenarios reales, en las que los grafos pueden ser de tamaño considerable, se han realizado distintas pruebas en las que se han medido los tiempos de identificación de distintas rutas.

5.1. Calculo de rutas usando Dijkstra sin paradas intermedias.

Para el cálculo de rutas entre dos puntos sin paradas obligatorias con el algoritmo Dijkstra se van a tener que utilizar las matrices y los diccionarios del grafo, detallados en la Sección 4.1. También hay que poder añadir nodos nuevos, uno de los cuales seguro va a ser el origen. El otro nodo que igual hay que añadir es el final, dependiendo de que tipo sea, nodo, tag o hueco picking, como se explicó en la Sección 4.2. Antes de aplicar el algoritmo de Dijkstra, hay que comprobar que existe camino entre el nodo origen y el nodo destino, como se describió en la Sección 4.3.

El Listado 5.1 muestra el pseudocódigo correspondiente al algoritmo Dijkstra, descrito en la Sección 3.4.

Algoritmo 5.1: Dijkstra.

Entrada: origen: tupla(coordX: real, coordY: real, id: String), destino: entero o tupla(coordX: real, coordY: real, id: String)
Salida: El camino entre los dos nodos, distancia entre los dos nodos

LÉXICO

```
global distancias: Matriz de reales
global idNode_posicionMatriz: Diccionario
global posicionMatriz_IDNode: Diccionario
s: array de enteros
caminos: array de diccionarios
```

ALGORITMO

```
// Si existe camino entre el origen y el destino
si existeCamino(origen, destino) entonces
    crearGrafo(origen, destino)
    numeroDeNodos ← longitud(distancias)
    dist ← [∞] * numeroDeNodos
    dist[origen] ← distancias[origen][origen]
     nodosVisitados ← crearVectorConValor(False, numeroDeNodos)
    padre ← [−1] * numeroDeNodos
    Para_Cada i Entre 0 Y numeroDeNodos − 1 Hacer
        minimo ← ∞
        u ← 0
        Para_Cada v Entre 0 Y longitud(nodosVisitados) Hacer
            si NodosVisitados[v] == False and dist[v] ⩽ minimo entonces
                minimo ← dist[v]
                u ← v
            nodosVisitado[u] ← true
            Para_Cada v Entre 0 Y numeroDeNodos Hacer
                si not(nodosVisitados[v]) and distancias[u][v] != 0 Y
                 dist[u] + distancias[u][v] < dist[v] entonces
                    padre[v] ← u
                    dist[v] ← dist[u] + distancias[u][v]
    Para_Cada i Entre 0 Y numeroDeNodos Hacer
        j ← i
        Repetir_Mientras padre[j] ≠ −1 Hacer
            s ← añadir(j)
            j ← padre[j]
        s ← añadir(origen)
        caminos[i] ← invertirArray(s)
    Devolver caminos[destino] dist[end]
en otro caso
    Devolver No existe camino
```

5.2. Calculo de rutas usando Dijkstra con paradas intermedias.

Las paradas intermedias son dadas sin orden. Hay que comprobar que orden de paradas es el óptimo para que la ruta sea la de menor distancia entre el origen y el destino. Para conseguir la mejor ruta, hay que comprobar todas las permutaciones del conjunto de las paradas.

Una permutación, según Veerarajan et al. (2008), es un arreglo ordenado de r elementos de un conjunto que contiene n elementos distintos, su denotación matemática es $P(n, r)$. La fórmula para saber el número de permutaciones que contiene un conjunto n, seleccionados de r en r, es.

$$P(n, r) = n(n-1)(n-2)\dots(n-r+1) = \frac{n!}{(n-r)!} \tag{5.1}$$

En este proyecto hay que seleccionar todas las paradas por lo que el total de las permutaciones sería de $P(n, n) = n!$ y el orden de computación de $O(n!)$. Según crecen el número de paradas el tiempo de computación crece de manera factorial.

En la Tabla 5.1 se muestra como crece el número de permutaciones, según crece el número de paradas.

Numero de paradas	1	2	3	4	5	6	7	8	9
Numero de permutaciones	1	2	6	24	120	720	5.040	40.320	362.880

Tabla 5.1 – Crecimiento de las permutaciones según el número de paradas.

Para abordar este problema, se crea un array con todas las permutaciones posibles del conjunto de las paradas. En un segundo paso se calcula la ruta con dijkstra entre el origen, cada parada de la permutación y el destino, esto se repite por cada permutación que contenga el array. Una vez calculados todos los posibles caminos se mira cual es el más corto y se selecciona.

Un ejemplo ilustrativo usando el mapa de la Figura 3.4, sería calcular la ruta saliendo de Vitoria, pasando por Bilbao y Pamplona, y acabando en San Sebastian. La búsqueda de la ruta se realizaría como se muestra en la Figura 5.1.

El pseudocódigo para el cálculo de la ruta con paradas es el descrito en el Algoritmo 5.2.

Algoritmo 5.2: Dijkstra con paradas intermedias.

Entrada: origen: tupla(coordX: real, coordY: real, id: String), destino: entero o tupla(coordX: real, coordY: real, id: String), puntosIntermedios: lista de enteros
Salida: elMejorCamino: array de reales

LÉXICO

global distancias: Matriz de reales
global idNode_posicionMatriz: Diccionario
global posicionMatriz_IDNode: Diccionario
permutaciones: Array de listas de enteros
camino: Array de enteros
caminos: Diccionario de caminos

ALGORITMO

$crearGrafo(origen, destino)$
$permutaciones \leftarrow buscarPermutaciones(puntosIntermedios)$
Para_Cada *permutación* DE *permutaciones* **Hacer**
  $camino \leftarrow limpiarCamino()$
  $dist \leftarrow 0$
  Para_Cada i Entre 0 Y $longitud(permutación)$ **Hacer**
    si $i == 0$ **entonces**
      $parteDelCamino, parteDistancia \leftarrow dijkstra(origen, permutacion[i])$
      $camino \leftarrow añadirCamino(parteDelCamino)$
      $dist \leftarrow parteDistancia$
    en otro caso
      $parteDelCamino,$
       $parteDistancia \leftarrow dijkstra(permutación[i-1], permutación[i])$
      $camino \leftarrow añadirCamino(parteDelCamino[1:]$
      $dist \leftarrow dist + parteDistancia$
  $parteDelCamino, parteDistancia \leftarrow dijkstra(permutación[i-1], destino)$
  $camino \leftarrow añadirCamino(parteDelCamino[1:]$
  $dist \leftarrow dist + parteDistancia$
  $camino \leftarrow añadirDistancia(dist)$
  $caminos \leftarrow añadirCamino(camino)$
Devolver $elMejorCamino \leftarrow obtenerCaminoMasCorto(caminos)$

5.3. Resultados usando Dijkstra.

Se van a realizar una serie de pruebas sobre el grafo que se mostró en la Figura 4.1. El cálculo de la ruta se va a mostrar de manera visual, sobre el mapa, y también se va a obtener el tiempo de computo.

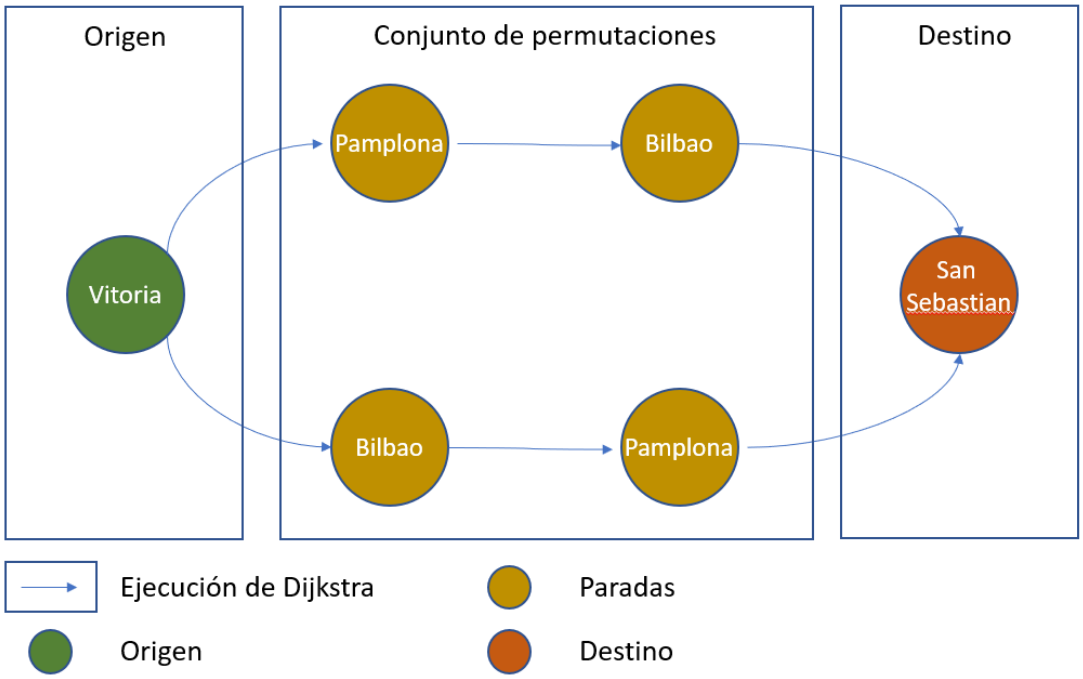


Figura 5.1 – Calculo de ruta con paradas intermedias usando Dijkstra

En primer lugar, se probará una ruta sin ningún punto intermedio. El destino será la cafetería, nodo 55. El origen tendrá las coordenadas (1530.0035,140.6469), lo que generará la tupla que se pasará como parámetro (150.0035,140.6469,origen). El resultado gráfico se puede ver en la Figura 5.2

Se puede comprobar que el camino obtenido es mínimo y la distancia del mismo es de 53.37 metros. El tiempo de computo ha sido de 561 milisegundos segundos.

En la segunda prueba se va a incorporar una parada intermedia. La parada intermedia será el CPD, nodo 62. Se seguirán usando el mismo origen y destino. El resultado de la ruta se puede observar en la Figura 5.3

El resultado genera una ruta de 64.51 metros y un tiempo de computo de 567 milisegundos.

En la tercera prueba, se incorpora una nueva parada. La nueva parada va a ser el "DS Labs", nodo 93. El resto de parámetros serán los mismos que en la prueba anterior. Resultado en la Figura 5.4.

La ruta obtenida tiene una distancia de 71.93 metros, y un tiempo de computo de 582 milisegundos.

Figura 5.2 – Ruta obtenida con el algoritmo de Dijkstra sin puntos intermedios.

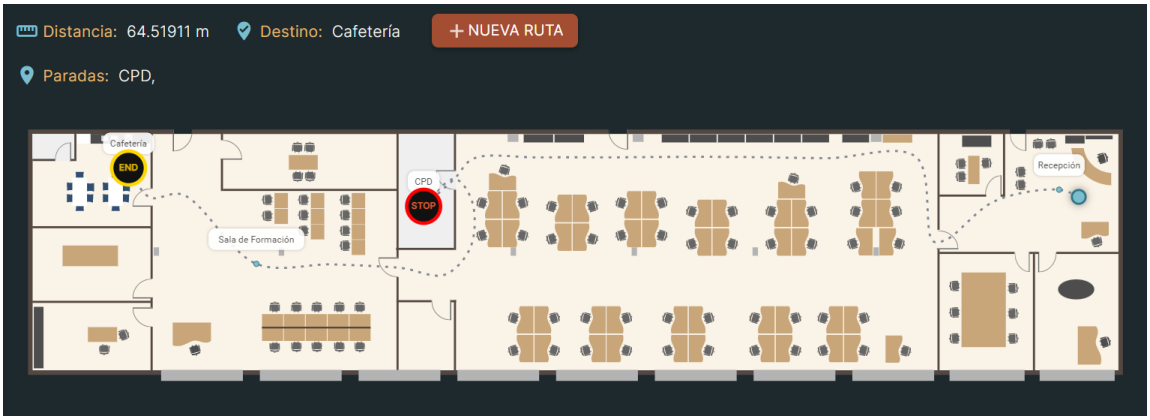


Figura 5.3 – Ruta obtenida con el algoritmo de Dijkstra con un punto intermedio.

Cuarta prueba, se incorpora una nueva parada. La nueva parada va a ser la sala de reuniones 2, nodo 57. El resto de parámetros serán los mismos que en la prueba anterior. Resultado en la Figura 5.5.

La ruta obtenida tiene una distancia de 84 metros, y un tiempo de computo de 624 milisegundos.

Quinta prueba, se incorpora una nueva parada. Ya vamos por cuatro paradas intermedias. La nueva parada va a ser la imprenta, nodo 74. El resto de parámetros serán los mismos que en la prueba anterior. Resultado en la Figura 5.6.

La ruta obtenida tiene una distancia de 88.30 metros, y un tiempo de computo de 749 milisegundos.

Figura 5.4 – Ruta obtenida con el algoritmo de Dijkstra con dos puntos intermedios.

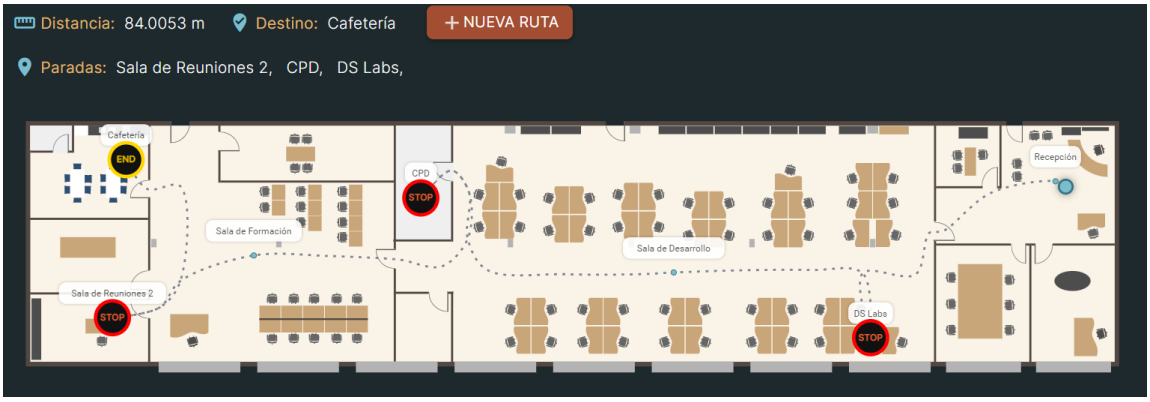


Figura 5.5 – Ruta obtenida con el algoritmo de Dijkstra con tres puntos intermedios.

Cinco paradas intermedias. La nueva parada va a ser el almacén, nodo 64. El resto de parámetros serán los mismos que en la prueba anterior. Resultado en la Figura 5.7.

La ruta obtenida tiene una distancia de 98.41 metros, y un tiempo de computo de 1.78 segundos.

Seis paradas intermedias. La nueva parada va a ser el despacho uno, nodo 59. El resto de parámetros serán los mismos que en la prueba anterior. Resultado en la Figura 5.8.

La ruta obtenida tiene una distancia de 110.19 metros, y un tiempo de computo de 9.35 segundos.

Siete paradas intermedias. La nueva parada va a ser el despacho dos, nodo 65. El resto de parámetros serán los mismos que en la prueba anterior. El resultado no se

Figura 5.6 – Ruta obtenida con el algoritmo de Dijkstra con cuatro puntos intermedios.

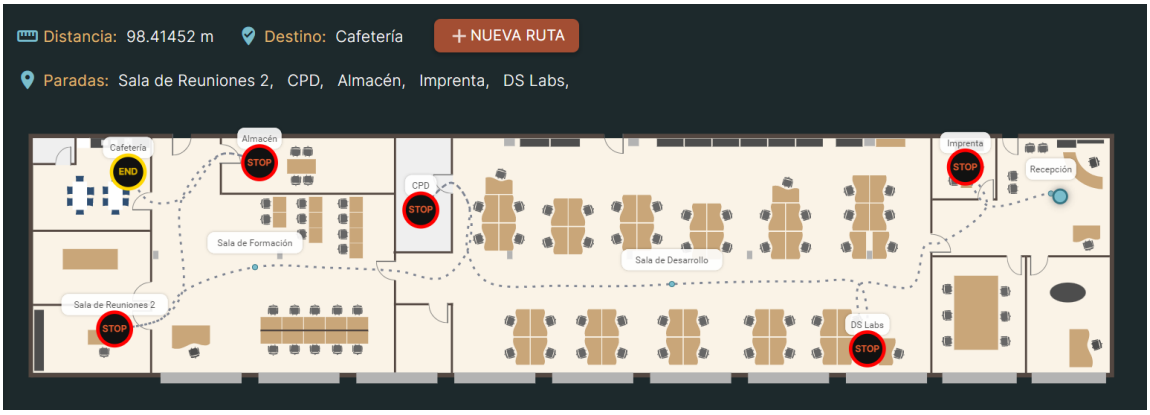


Figura 5.7 – Ruta obtenida con el algoritmo de Dijkstra con cinco puntos intermedios.

puede ver, ya que en la web da un *time-out*, pero se ha podido ejecutar el código y sacar los resultados.

La ruta obtenida tiene una distancia de 118.19 metros, y un tiempo de computo de 1 minuto y 8 segundos.

Ocho paradas intermedias. La nueva parada va a ser sala de reuniones 3, nodo 56. El resto de parámetros serán los mismos que en la prueba anterior. El resultado no se puede ver, ya que en la web da un *time-out*, pero se ha podido ejecutar el código y sacar los resultados.

La ruta obtenida tiene una distancia de 125 metros, y un tiempo de computo de 10 minutos y 21 segundos.

Figura 5.8 – Ruta obtenida con el algoritmo de Dijkstra con seis puntos intermedios.

5.4. Conclusiones para el cálculo de rutas con el algoritmo de Dijkstra.

Después de las pruebas realizadas, se puede llegar a las siguientes conclusiones.

Como se puede observar en la gráfica de la Figura 5.9, y en la Tabla 5.2, en los cálculos hasta los 6 puntos intermedios, el incremento del costo es lento. A partir de 6 puntos intermedios se produce un punto de inflexión, el costo empieza a incrementarse a un ritmo muy elevado. Se puede llegar a la conclusión de que el orden de complejidad del algoritmo de Dijkstra con puntos intermedios es factorial, $O(n!)$.

Para el proyecto que se estudia, los cálculos de rutas de hasta 6 puntos intermedios, el algoritmo de Dijkstra es válido. A partir de 6 puntos intermedios el algoritmo deja de ser válido por el alto costo de computo que necesita.

Puntos intermedios	Tiempo de computo
0	561 msg
1	567 msg
2	582 msg
3	624 msg
4	749 msg
5	1.78 sg
6	9.35 sg
7	1 min 8 sg
8	10 min 21 sg

Tabla 5.2 – Tiempos para el cálculo de rutas usando el algoritmo de Dijkstra.

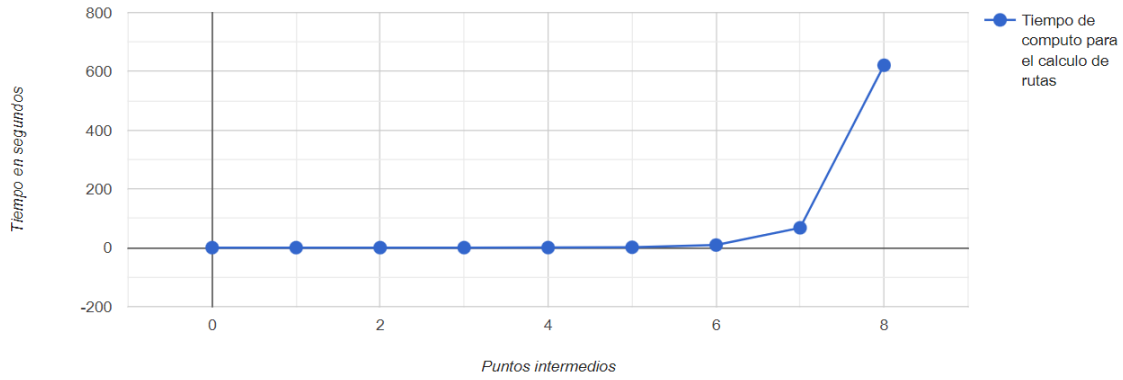


Figura 5.9 – Tiempos de cálculo de rutas con el algoritmo de Dijkstra.

6

Algoritmo Ant Colony Optimization (ACO)

De las conclusiones sacadas del algoritmo de Dijkstra en la Sección 5.4, se necesita buscar una solución alternativa para cuando se desea buscar rutas con más de seis puntos intermedios. Usando Dijkstra, el crecimiento de tiempo de cómputo a medida que se van añadiendo puntos intermedios es factorial. Para lidiar con grafos con un número de nodos elevado es necesario un algoritmo de orden lineal.

Para dar solución a este problema, se ha decido aplicar un algoritmo heurístico. En la Sección 3.5 se mencionaron algunas de las posibles soluciones que hay en dicho campo. Para este trabajo se ha decidido implementar, en un principio, el algoritmo de ACO en su variante de MMAS, por los motivos argumentados en la Sección 3.7.

6.1. Implementación del algoritmo $MAX - MIN$ *Ant System (MMAS)*

Este algoritmo aporta la ventaja de que los caminos no explorados tengan más posibilidades de ser elegidos por las hormigas que en el algoritmo *Ant System (AS)*. El motivo es que se pone un límite mínimo a la cantidad de feromonas de cada arista. También se acota el máximo de las feromonas de cada arista para que la exploración de nuevos caminos pueda ser mayor.

Este algoritmo cuenta con cinco variables que hay que ajustar, para que la ejecución del mismo sea lo más eficiente posible. Dichas variables son:

- **Número de hormigas:** Es el número de hormigas que se sueltan para la búsqueda de rutas en cada iteración.

- **Número de iteraciones:** Es la cantidad de veces que el algoritmo suelta las hormigas para la búsqueda de rutas

- **Factor de evaporación:** La cantidad de feromona que se va en cada iteración.

- **Peso de las feromonas en la elección del siguiente nodo:** A este factor se le llamará α, y determinará la importancia que tienen las feromonas en la elección del siguiente nodo.

- **Peso de la distancia de la arista en la elección del siguiente nodo:** A este factor se le llamará β, y determinará la importancia que tiene la longitud de la arista para elegir el siguiente nodo.

- **Máximo de feromona:** Valor máximo que puede tener una feromona.

- **Mínimo de feromona:** Valor mínimo que puede tener una feromona.

El Listado 6.1 muestra el algoritmo ACO en su variante MMAS.

El grafo que representa el escenario puede ser un grafo no completo, pueden existir vértices entre los que no exista una arista. Esto puede ocasionar que al ir eliminando los nodos ya visitados, la hormiga se encuentre en un nodo desde el que no pueda viajar a ningún otro nodo. Para solucionar este problema, la función de elegir el nodo siguiente identifica esta situación y la comunica. El pseudocódigo de dicha función se detalla en el Algoritmo 6.2.

Algoritmo 6.1: $MAX - MIN$ Ant System (MMAS).

Entrada: origen: entero, destino: entero
Salida: El camino entre los dos nodos

LÉXICO
iteraciones: entero
hormigas: entero
nodosVisitados: array de enteros
rutas: array de nodosVisitados
Global matrizDistancias: matriz de reales

ALGORITMO

```
Para_Cada i Entre 0 Y iteraciones Hacer
    Para_Cada i Entre 0 Y hormigas Hacer
        distancias ← matrizDistancias
        nodosVisitados ← añadir(origen)
        nodoActual ← origen
        Repetir_Mientras nodoActual ≠ destino Y
         haynodosiguiente(nodoAcutal, distancias) Hacer
            nodoSiguiente ← elegirNodo(nodoActual, distancias)
            distancias[, nodoActual] ← 0
            nodosVisitados ← añadir(nodoSiguiente)
            nodoActual ← nodoSiguiente
        si nodoSiguiente ≠ ∞ entonces
            rutas ← añadir(nodosVisitados)
    si longitud(rutas)! = 0 entonces
        actualizarFeromonas(rutas)
mejorRuta ← rutaMasCorta(rutas)
Devolver mejorRuta
```

Algoritmo 6.2: Elegir siguiente nodo.

```
Entrada: nodoActual: entero, distancias: matriz de reales
Salida: nodoSiguiente: entero
LÉXICO
α: real
β: real
probabilidadNodo: array de reales
listaNodos: array de enteros
Global feromonas: matriz de reales
ALGORITMO
    posiblesDestinos ← distancias[nodoActual, ]
    listaFeromonas ← feromonas[nodoActual, ]
    si ningunDestino(posiblesDestinos) entonces
        Devolver ∞
    Para_Cada i Entre 0 y longitud(posiblesDestinos) Hacer
        si posiblesDestinos[i] ≠ 0 entonces
            SumatorioDistanciasFeromonas ← 0
            Para_Cada z Entre 0 y longitud(posiblesDestinos) Hacer
                si posiblesDestinos[z] ≠ 0 entonces
                    sumatorioDistanciasFeromonas =
                      sumatorioDistanciasFeromonas + (listaFeromonas[z]^α *
                      (1/posiblesDestinos[z]^β))
            prob ← (listaFeromonas[i]^α *
              (1/posiblesDestinos[i]^β))/sumatorioDistanciasFeromonas
            probabilidadNodo ← añadir(prob)
            listaNodos ← añadir(i)
    Devolver elegirSegunProbabilidad(probabilidadNodo, listaNodos)
```

Cada vez que se acaba una iteración, hay que actualizar las feromonas. De este modo, las hormigas se ven condicionadas a coger el nodo que tenga más feromonas. Esto provoca que se busquen más caminos alrededor del mejor camino encontrado hasta ese punto. El pseudocódigo de la actualización de las feromonas es el descrito en el Algoritmo 6.3

Algoritmo 6.3: Actualizar las feromonas.

Entrada: listaRutas: array de rutas
Salida: Actualiza la matriz global de feromonas

LÉXICO
máximoFeromona: real
mínimoFeromona: real
factorDeEvaporación: real
Global matrizFeromonas: matriz de reales

ALGORITMO

$mejorCamino \leftarrow buscarMejorCamino(listaRutas)$
$distancia \leftarrow conseguirDistancia(mejorCamino)$
$matrizFeromonas \leftarrow$
$multiplicarTodosLosElementos(matrizFeromonas, (1 - factorDeEvaporación))$
Para_Cada i Entre 1 y $longitud(mejorCamino)$ **Hacer**
$matrizFeromonas[(i-1), i] \leftarrow matrizFeromonas[(i-1), i] + (1/distancia)$
$matrizFeromonas \leftarrow$
$limiteInferiorSuperior(matrizFeromonas, máximoFeromonas, mínimoFeromonas)$

Si el grafo que se está procesando tiene nodos de grado uno, nodos que solo tienen una arista, las hormigas pueden quedar atrapadas en un nodo, ya que no pueden retroceder por el camino transitado, ver Figura 6.1.

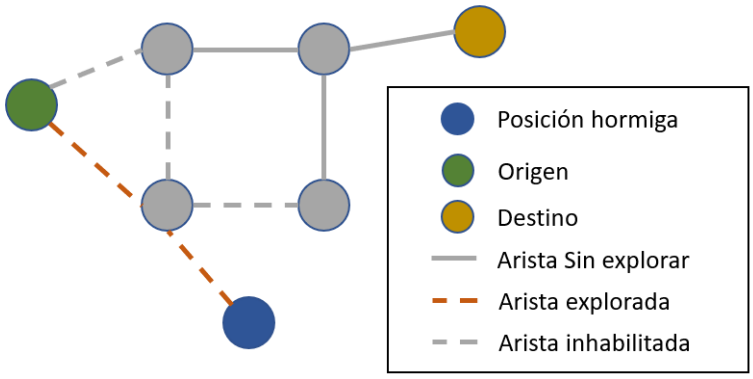


Figura 6.1 – Hormiga atrapada en un nodo de grado uno

Para dar solución a este problema, se ha adaptado el algoritmo de la siguiente manera. La hormiga, antes de viajar al siguiente nodo, va a comprobar si ese nodo es

de grado uno. Si el siguiente nodo tiene grado uno, se verifica si es el nodo destino, en cuyo caso la hormiga se desplazará a dicho nodo. Si el nodo no es el nodo destino, la hormiga descarta el nodo y seleccionará otro nodo al que viajar si es posible.

El algoritmo ACO, para evitar la repetición de nodos en el camino, elimina las aristas que tengan como destino el nodo alcanzado en cada paso. Esto puede ocasionar que, como se puede observar en Figura 6.2, la hormiga también se pueda quedar atrapada. Esto impide que la hormiga siga explorando el grafo en búsqueda de una solución.

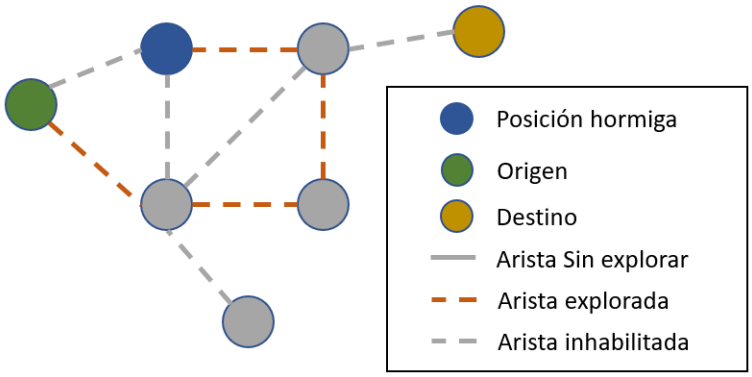


Figura 6.2 – Hormiga atrapada por las aristas inhabilitadas

Para mitigar este problema se decidió no inhabilitar las aristas por las que no haya pasado la hormiga. De esta manera la hormiga puede seguir explorando el grafo y encontrar una solución, como se muestra en la Figura 6.3.

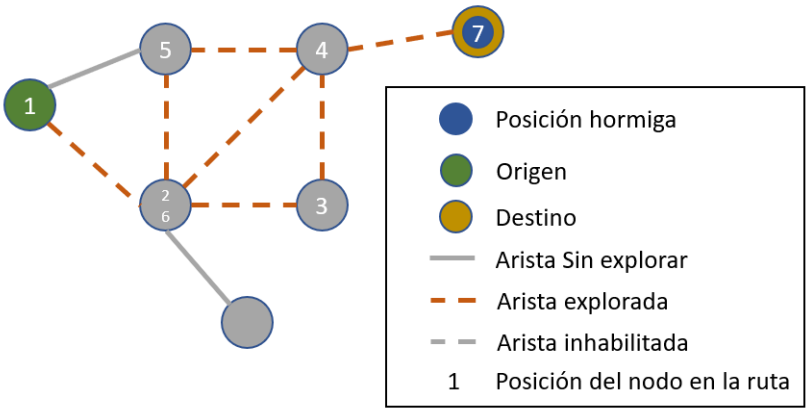


Figura 6.3 – Posible recorrido de una hormiga sin inhabilitar aristas.

Sin embargo, la solución dada al atrapamiento de hormigas por aristas inhabilitadas, genera otro problema. Las rutas encontradas por las hormigas pueden tener bucles, como el que se ha generado en la Figura 6.3.

Para asegurar que las rutas no tienen bucles se ha diseñado una función que revisa la ruta y elimina los bucles.

En algunos grafos pueden existir puentes. Un puente es una arista que conecta un bloque del grafo con otro bloque del grafo. Se muestra un ejemplo de un puente en la imagen de la Figura 6.4, que representa un fragmento de los escenarios en los que se ha utilizado este proyecto.

Los puentes pueden dificultar la exploración de caminos de las hormigas, ya que estas podrían quedar atrapadas en un lado del puente del grafo. Para evitar esta situación, se modifica el grafo, siempre que sea posible, para eliminar el puente. La modificación propuesta es duplicar el nodo y las aristas del puente. La Figura 6.6 muestra el resultado de aplicar la solución propuesta para evitar el puente de la Figura 6.5.

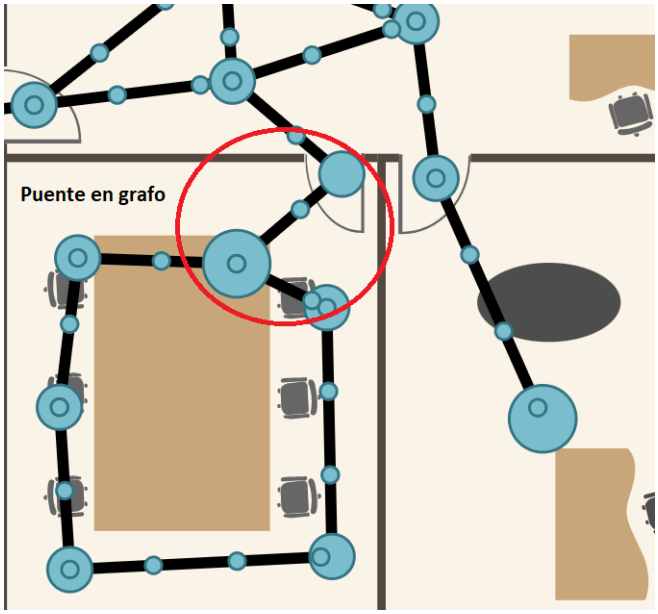


Figura 6.4 – Ejemplo real de puente en un grafo.

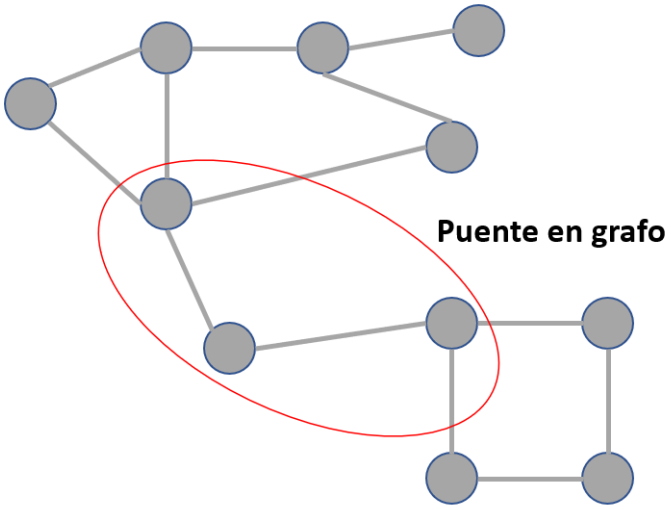


Figura 6.5 – Ejemplo simulado de un puente en un grafo.

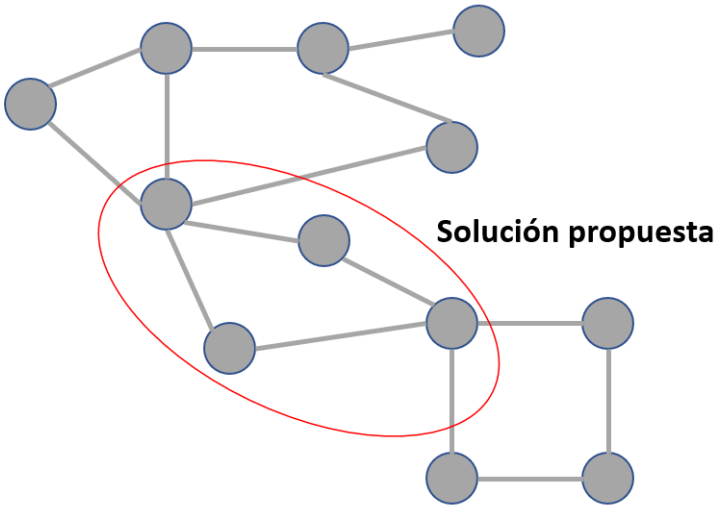


Figura 6.6 – Solución propuesta a los puentes en los grafos.

A pesar de las mejoras introducidas, se pueden dar situaciones como la que se muestra en la Figura 6.7 en las que la hormiga puede quedar atrapada. En esa situación no queda otro remedio que desechar la hormiga.

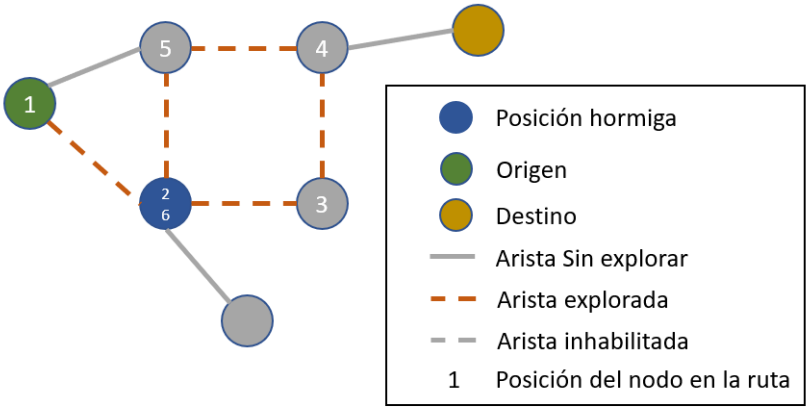


Figura 6.7 – Hormiga atrapada y desechada.

6.2. Algoritmo MMAS modificado para evitar el colapso de algunas hormigas

Para el desarrollo de este algoritmo se ha partido de la implementación realizada por PelayoChoya[1] y se han adaptado a las mejoras antes mencionadas. Se espera que con esta adaptación las hormigas sean más eficientes en la búsqueda de rutas. La modificación se puede ver en detalle en el Algoritmo 6.5.

Como es inevitable que se creen bucles a la hora de buscar rutas, se ha creado una función para eliminarlos, la cual se explica en el pseudocódigo del Algoritmo 6.4.

Algoritmo 6.4: Borrar bucles.

```
Entrada: ruta: array de enteros
Salida: rutaSinBucles: array de enteros
LÉXICO
 rutaSinBucles: array de bucles
ALGORITMO
    rutaSinBucles ← ruta
    Para_Cada i Entre 0 Y longitud(ruta) Hacer
        repeticionNodo ← buscarCoincidencias(ruta, ruta[i])
        repeticionNodo ← revertirArray(repeticionNodo)
        Para_Cada i Entre 0 Y longitud(repeticionNodo) Hacer
            si i ≠ longitud(repeticionNodo) entonces
                rutaSinBucles[repeticionNodo[i + 1] : repeticionesNodo[i]]
    Devolver rutaSinBucles
```

[1] https://github.com/PelayoChoya/ACO_path_planning

Algoritmo 6.5: Modificación de MMAS para optimizar la búsqueda de rutas.

Entrada: origen: entero, destino: entero
Salida: El camino entre los dos nodos

LÉXICO
iteraciones: entero
hormigas: entero
nodosVisitados: array de enteros
rutas: array de nodosVisitados
Global matrizDistancias: matriz de reales

ALGORITMO

```
Para_Cada i Entre 0 Y iteraciones Hacer
    Para_Cada i Entre 0 Y hormigas Hacer
        distancias ← matrizDistancias
        nodosVisitados ← añadir(origen)
        nodoActual ← origen
        // El bucle siguiente se repite en la adaptación con paradas, se
           le hará mención en el Algoritmo 6.6
        Repetir_Mientras nodoActual ≠ destino Y
         hayNodoSiguiente(nodoActual, distancias) Hacer
            nodoSiguiente ← elegirNodo(nodoActual, distancias)
            // El condicional siguiente se repite en la adaptación con
               paradas, se le hará mención en el Algoritmo 6.6
            si nodoSiguiente ≠ destino entonces
                distancias[nodoActual, nodoSiguiente] = 0
                distancias[nodoSiguiente, nodoActual] = 0
                destinosNodoSiguiente = distancias[nodoSiguiente, ]
                Repetir_Mientras ningunDestino(destinosNodoSiguiente) Y
                 hayNodoSiguiente(nodoActual, distancias) Hacer
                    nodoSiguiente ← elegirNodo(nodoActual, distancias)
                    distancias[nodoActual, nodoSiguiente] = 0
                    distancias[nodoSiguiente, nodoActual] = 0
                    destinosNodoSiguiente = distancias[nodoSiguiente, ]
            nodosVisitados ← añadir(nodoSiguiente)
            nodoActual ← nodoSiguiente
        si nodoSiguiente ≠ ∞ entonces
            nodosVisitados = quitarBucles(nodosVisitados)
            rutas ← añadir(nodosVisitados)
    si longitud(rutas) ≠ 0 entonces
        actualizarFeromonas(rutas)
mejorRuta ← rutaMasCorta(rutas)
Devolver mejorRuta
```

Para determinar la configuración más adecuada del algoritmo se han realizado varias pruebas sobre el mismo grafo que se probó en la sección anterior. Se muestran dos pruebas realizadas en Figura 6.8 y Figura 6.9.

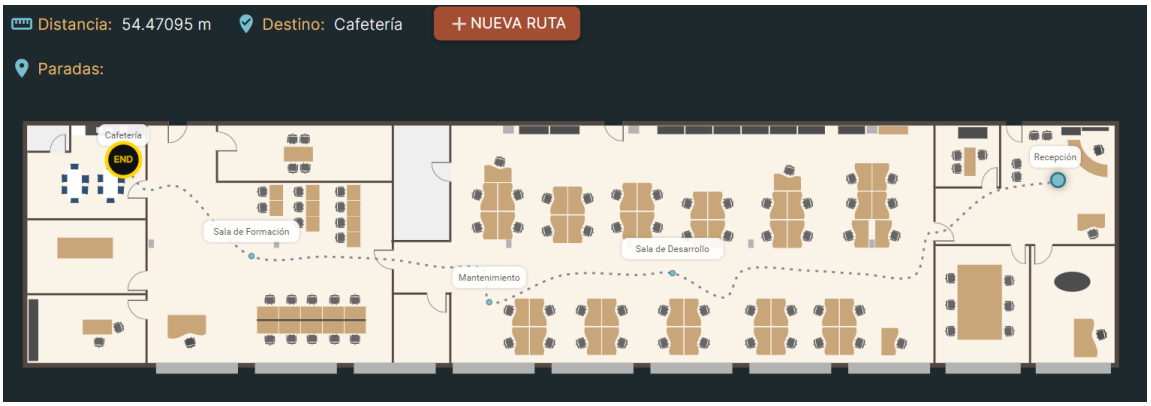


Figura 6.8 – Ejemplo de MMAS sin puntos intermedios.

Figura 6.9 – Segundo ejemplo de MMAS sin puntos intermedios.

Después de varias pruebas, los parámetros que mejor se han adaptado al grafo son los siguientes:

- **Iteraciones:** Es variable según los puntos intermedios.
- **Hormigas:** Es variable según los puntos intermedios.
- **Valor máximo de la feromona:** 0,8.

- **Valor mínimo de la feromona:** 0,3.
- **Factor de evaporación:** 0.2.
- α**:** 1.
- β**:** 3.

Se ha dado más peso a la hora de elegir nodo al factor de la evaporación que al factor de la distancia. El valor mínimo de las feromonas no se ha fijado muy bajo, para que las feromonas no premiadas con la mejor ruta tengan algo más de relevancia a la hora de la elección del siguiente nodo. De esta manera se provoca buscar nuevas rutas.

Como se puede ver en los resultados que muestran Figura 6.8 y Figura 6.9, las rutas no son las mejores. Sin embargo, las rutas encontradas son aceptables. Existe poca variación de distancia entre las dos pruebas.

6.3. Algoritmo MMAS adaptado para paradas intermedias

El algoritmo MMAS modificado para búsqueda de rutas sin paradas intermedias ha tenido un comportamiento aceptable, como se ha podido ver en la Sección 6.2. Ahora es el momento de incluir las paradas intermedias y ver como se comporta el algoritmo.

Para la adaptación de paradas intermedias, se va a variar el comportamiento de las hormigas. La hormiga va ha salir del origen en busca de una parada intermedia, cualquiera de ellas. Cuando se encuentra la parada se limpia el recorrido de bucles, y la hormiga sigue en busca de la siguiente parada intermedia. Este proceso se repite hasta que no queden paradas intermedias, que es el momento en que la hormiga empieza a buscar el destino. En cada proceso de búsqueda de un punto por la hormiga, la matriz de distancias se tiene que resetear para que no tenga aristas inhabilitadas.

El pseudocódigo del algoritmo definido se puede ver en el Algoritmo 6.6.

Algoritmo 6.6: Adaptación de $MMAS$ para incluir paradas intermedias.

```
Entrada: origen: entero, destino: entero, paradas: array de enteros
Salida: El camino entre los dos nodos
LÉXICO
iteraciones: entero
hormigas: entero
nodosVisitados: array de enteros
rutaParcial: array de enteros
rutas: array de nodosVisitados
rutaSinBucles: array de enteros
Global matrizDistancias: matriz de reales
ALGORITMO
    Para_Cada i Entre 0 Y iteraciones Hacer
        Para_Cada i Entre 0 Y hormigas Hacer
            distancias ← matrizDistancias
            nodosVisitados ← [origen]
            nodoActual ← origen
            rutaSinBucles ← [origen]
            Repetir_Mientras longitud(paradas) ≠ 0 Hacer
                Repetir_Mientras noEstaEnParadas(nodoActual, paradas) Y
                 hayNodoSiguiente(nodoActual, distancias) Hacer
                    nodoSiguiente ← elegirNodo(nodoActual, distancias)
                    si noEstaEnParadas(nodoActual, paradas) entonces
                        // Condicional que se indica en el Algoritmo 6.5
                    nodosVisitados ← añadir(nodoSiguiente)
                    nodoActual ← nodoSiguiente
                distancias ← matrizDistancias
                rutaParical ← quitarBucles(nodosVisitados)[1 :]
                rutaSinBucles ← añadir(rutaParcial)
                actualizacionParcialFeromonas(rutaSinBucles)
                nodosVisitados ← [nodoActual]
                paradas ← eliminarParada(nodoActual)
            distancias ← matrizDistancias
            si hayNodoSiguiente(nodoActual, distancias) entonces
                Repetir_Mientras  nodoActual ≠ destino Hacer
                    // Bucle que se indica en el Algoritmo 6.5
                si hayNodoSiguiente(nodoActual, distancias) entonces
                    rutaParical ← quitarBucles(nodosVisitados)[1 :]
                    rutaSinBucles ← añadir(rutaParcial)
                    rutas ← añadir(rutaSinBucles)
        si longitud(rutas) ≠ 0 entonces
            actualizarFeromonas(rutas)
    mejorRuta ← rutaMasCorta(rutas)
    Devolver mejorRuta
```

Se ha añadido otra variante al algoritmo $MMAS$. Cada vez que una hormiga encuentra un nodo intermedio, se hace una actualización parcial de las feromonas. Esta actualización penaliza en menor medida los nodos no visitados. Es decir el factor de evaporación de las feromonas es mucho menor. Con esto se quiere conseguir que las siguientes hormigas se vean sesgadas en la elección de los nodos a favor de la ruta parcial encontrada. El depósito de feromonas también tiene una modificación. Cuando una hormiga encuentra un camino parcial, el premio que reciben los nodos afectados es mucho menor que cuando se actualiza con el mejor camino encontrado en una iteración. Las nuevas variables para la función de actualización parcial de feromonas son:

- Evaporación parcial de feromonas, $\rho_{parcial}$.
- Premio a la ruta parcial encontrada, $\triangle\tau_{ij}^{k}$.

$$\triangle\tau_{ij}^{k} = \begin{cases} \rho_{parcial} & \text{si } (i,j) \text{ está en la ruta parcial} \\ 0 & \text{en otro caso} \end{cases} \tag{6.1}$$

Al testear el algoritmo con 7 puntos intermedios, se puede observar en la Figura 6.10 que se encuentra una ruta más o menos aceptable, pero al dibujarla sobre el mapa se observa que la ruta hace eses que desde los puntos seleccionados en la ruta hasta cualquier otro punto seleccionado en la ruta. Es decir los puntos intermedios los ordena correctamente para seleccionar la mejor ruta, pero la ruta encontrada entre ellos pierde calidad.

Figura 6.10 – Ruta encontrada por $MMAS$ con siete puntos intermedios.

7

Solución hibrida

La última solución propuesta consiste en utilizar la solución de Dijkstra, explicada en el Capítulo 5 para buscar la mejor ruta entre todos los puntos implicados en el grafo, y crear un subgrafo. En este proyecto, los puntos implicados en el grafo son el origen, el destino y los puntos intermedios por los que hay que pasar sí o sí.Una vez creado el subgrafo se le han aplicado dos algoritmos diferentes para la búsqueda de la ruta, el algoritmo de ramificación y acotamiento, *(Branch and Bound)*, y el algoritmo ACOexplicado en el Capítulo 6. El algoritmo de ramificación y acotamiento para rutas con más de dos puntos intermedios arroja una solución óptima con un coste computacional bastante mayor al de ACO. Se han estudiado las ventajas y desventajas que ofrecen los algoritmos de ramificación y acotamiento y ACO ejecutados sobre el subgrafo generado.

En el Capítulo 6, cuando se ejecutaba el algoritmo ACO se generaba una matriz de pesos del grafo. La matriz generada era del orden del número de nodos del grafo por el número de nodos del grafo. Otra característica del grafo original, es que no era un grafo completo. Un grafo completo es cuando existe camino entre cada par de nodos.

Lo que se va a realizar, es ejecutar el algoritmo de Dijkstra desde cada punto implicado, hasta cada punto implicado. De esta manera se obtienen todos los caminos óptimos y sus distancias, entre todos los puntos implicados en el proceso de búsqueda de ruta. Una vez obtenidos los datos, en realizad lo que se obtiene es un nuevo subgrafo de $G(v, e)$.

$$G(v,e) \Rightarrow \begin{cases} v = \text{puntos implicados en la búsqueda.} \\ e = \frac{v(v-1)}{2} \end{cases} \tag{7.1}$$

Las ventajas de la creación del subgrafo son.

- Reducir la matriz de pesos con la que trabaja el algoritmo ACO. La matriz de pesos pasa a tener un orden del número de nodos del grafo por el número de nodos del grafo, a tener un orden de número de puntos implicados en la búsqueda por el número de puntos implicados en la búsqueda.

- El grafo se convierte en un grafo completo. El original era un grafo simple. Ahora existe camino entre cada par de nodos, lo que simplifica mucho el algoritmo ACO.

- Se ha obtenido el camino óptimo entre cada par de puntos. La heurística de ACO influye únicamente en el orden de selección de los puntos intermedios y no en la ruta entre ellos.

- Da la posibilidad de aplicar sobre el subgrafo el algoritmo de ramificación y acotamiento.

En definitiva. Se ha transformado en problema original, en un problema TSP puro. Con todas las ventajas que ello genera.

Ejemplo: En la Figura 7.1, se muestra un grafo de orden $G(276, 389)$. Este grafo generaría una matriz de pesos de $276X276$. Si queremos ir de un origen a un destino pasando por 8 puntos intermedios como se muestra en la Figura 7.2, ejecutamos el algoritmo de Dijkstra entre cada par de nodos implicados. Con las distancias obtenidas se crea un nuevo grafo, como el que se muestra en la Figura 7.3. Este nuevo grafo pasa a ser de orden $G(8, 28)$, que generaría una matriz de pesos de $8X8$. Ahora el trabajo para ACO con este subgrafo y su matriz de pesos correspondiente, no tiene nada que ver con el grafo original y su matriz de pesos.

En el segundo paso habría dos opciones, o bien ejecutar el algoritmo de ramificación y acotamiento, o bien ejecutar el algoritmo ACO. Ambos algoritmos tendrían como punto de partida el origen, como punto final el destino, y tendrían que ordenar los puntos intermedios, de tal modo que el resultado obtenido sería el de menor coste.

El tercer paso y último consistiría en añadir el camino óptimo entre cada par de nodos del resultado. Estos caminos óptimos los hemos generado en el primer paso al ejecutar el algoritmo de Dijkstra.

La solución tiene una complejidad computacional de $O(n^2)$ por la ejecución del algoritmo de Dijkstra mas la complejidad del algoritmo ACO o el de ramificación y acotamiento, los cuales ambos tienen una complejidad diferente. La de ACO es de $O(n^2)$ y la de ramificación y acotamiento es de $O(n^2 2^n)$.

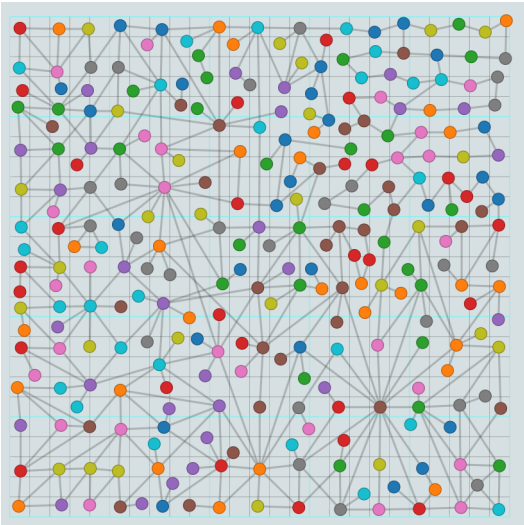


Figura 7.1 – Grafo con una cantidad de nodos alta.

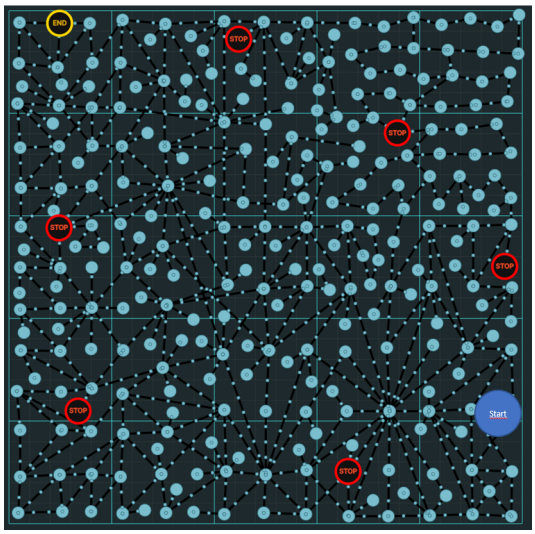


Figura 7.2 – Puntos implicados en la búsqueda de la ruta.

7.1. Ramificación y acotamiento

Este algoritmo se basa en transformar el subgrafo generado en un árbol. La raíz del árbol sería el nodo de punto de partida. A partir de dicho nodo se van desplegando ramas con las diferentes rutas posibles. A cada rama se le asigna un valor, el cual va

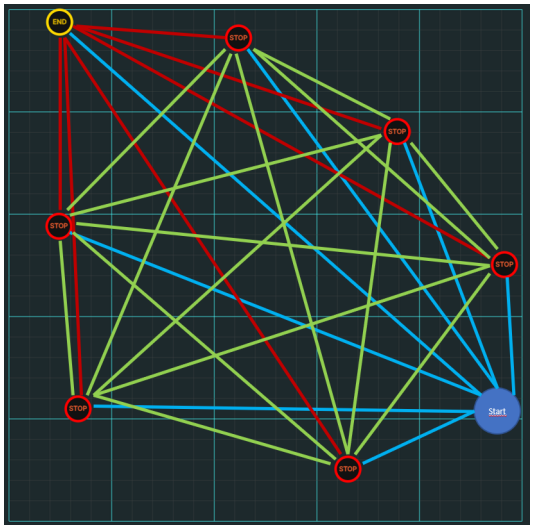


Figura 7.3 – Subgrafo generado.

a ser la distancia mínima que se va ha obtener en la ruta si se continua desplegando esa rama.

Cada rama acaba en una hoja, la hoja es representada por el nodo destino, es decir es el final del árbol, el cual ha recorrido todos los demás nodos. Cuando se llega a la hoja se obtiene la distancia de esa ruta.

Si las ramas que quedan por desplegar como valor una distancia mínima de la ruta mayor al valor de la mejor ruta encontrada hasta el momento, entonces esas ramas se podan y no se exploran.

Una vez exploradas todas las ramas del árbol que no se han podado se puede obtener la ruta mínima del grafo pasando por todos los puntos intermedios. El coste del peor caso, es decir, que no se pueda podar ninguna rama, del algoritmo de ramificación y acotamiento será de $O(n^2 2^n)$. Se realiza un ejemplo para entender mejor el algoritmo.

7.1.1. Ejemplo de ramificación y acotamiento

El ejemplo que se va a utilizar para ilustrar este algoritmo parte del subgrafo que se muestra en la Figura 7.4. Es un subgrafo dirigido y completo, lo que significa que desde cualquier nodo se puede viajar hasta cualquier otro nodo. La matriz reducida respecto al grafo original que genera este subgrafo es la siguiente.

$$A_{reducida} = \begin{pmatrix} 0 & 14 & 4 & 10 & 1 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix} \tag{7.2}$$

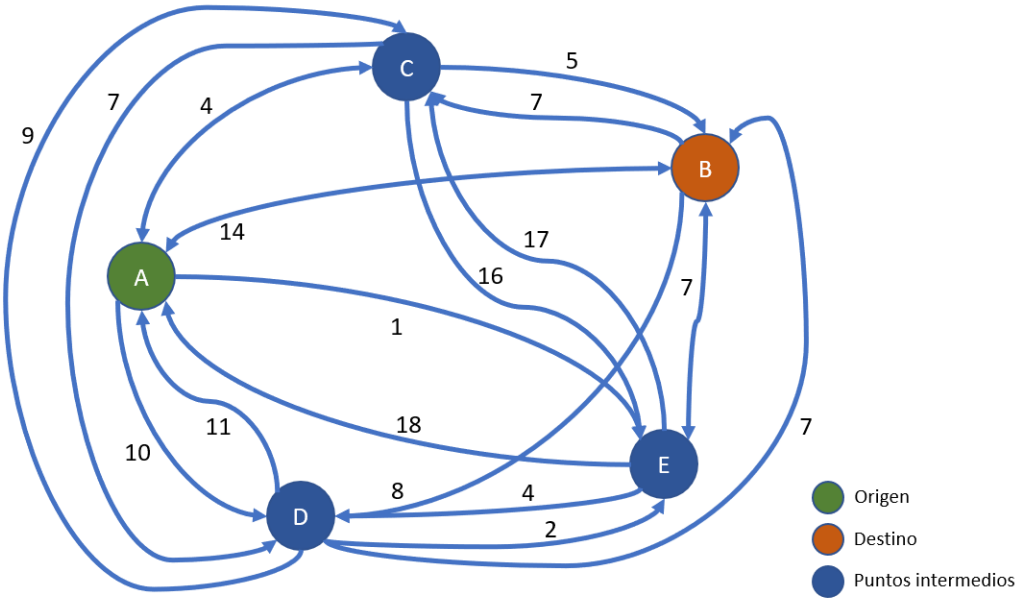


Figura 7.4 – Subgrafo utilizado en el ejemplo del algoritmo de ramificación y acotamiento.

Lo primero que se realiza es crear la raíz del árbol y añadir el coste mínimo que va a tener la ruta desde la raíz. Para el cálculo de dicho coste se cogen los valores mínimos cada fila de la matriz sin incluir la fila del nodo destino, ya que desde ese nodo no se va a viajar a ningún lado. En los nodos intermedios tampoco se puede escoger el valor de la columna del nodo origen ya que no se va a volver en ningún caso al origen. Dicho esto, los valores que se escogerían en la matriz para el camino mínimo desde la raíz serían los siguientes.

$$A_{reducida} = \begin{pmatrix} 0 & 14 & 4 & 10 & \textcircled{1} \\ 14 & 0 & 7 & 8 & 7 \\ 4 & \textcircled{5} & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & \textcircled{2} \\ 18 & 7 & 17 & \textcircled{4} & 0 \end{pmatrix} \tag{7.3}$$

Cualquier camino que salga desde la raíz (origen), tiene que tener un coste mínimo de 12.

El segundo paso es explorar todos los caminos a los que se pueden llegar desde la raíz. Desde la raíz no se puede llegar al destino, ya que antes hay que pasar por todos los demás nodos. También hay que calcular el coste mínimo que tiene cada rama.

Por ejemplo si vamos de la raíz al punto intermedio C, el coste mínimo será la arista de A a C más el valor mínimo de los demás puntos intermedios sin contar el camino ya recorrido, en este caso los nodos A y C. Por lo tanto los valores seleccionados en la matriz reducida serán los siguientes.

$$A_{reducida} = \begin{pmatrix} 0 & 14 & \textcircled{4} & 10 & 1 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & \textcircled{7} & 16 \\ 11 & 7 & 9 & 0 & \textcircled{2} \\ 18 & 7 & 17 & \textcircled{4} & 0 \end{pmatrix} \tag{7.4}$$

La nueva rama que ha generado el árbol que va desde el nodo origen hasta el punto intermedio C generará una ruta como mínimo con un valor de 17. Si realizamos la misma operación con todos los puntos intermedios a los que puedes viajar desde la raíz quedaría un árbol como el mostrado en la Figura 7.5.

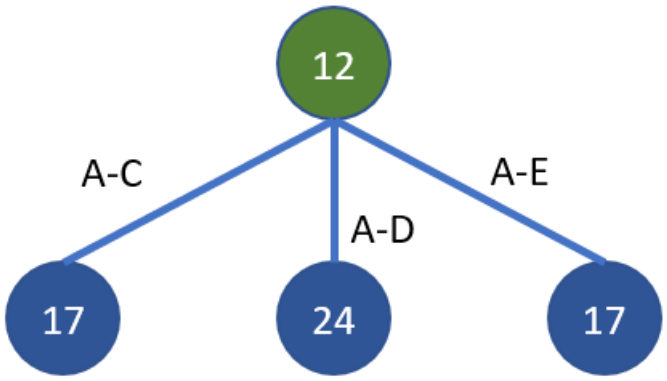


Figura 7.5 – Primer nivel del árbol.

Después de haber explorado el primer nivel del árbol se escoge la rama que más probabilidades de éxito tenga, en este caso, existen dos ramas con la misma probabilidad de éxito, por lo que se escoge una de manera indistinta. Para el ejemplo se va a explorar primero la rama que va de A a C.

De la rama que va de A a C, a su vez van a surgir dos nuevas ramas, las del camino $A - C - D$ y las del camino $A - C - E$, como para nuestro problema es condición indispensable que la ruta pase por todos los puntos intermedios y acabe en el destino seleccionado, desde este nivel ya se puede llegar a las hojas de las nuevas ramas generadas. Las rutas exploradas serían $A-C-D-E-B$ y $A-C-E-D-B$, las cuales tendrían un coste de 20 y 31 respectivamente. Por el momento la hoja que tiene un menor coste es la del camino $A - C - D - E - B$ con un coste de 20. Se muestra el árbol explorado hasta el momento en la Figura 7.6. También se puede deducir que la rama que va de A a D tiene un coste mínimo de 24 y al encontrar una hoja con un coste menor, dicha rama se puede podar.

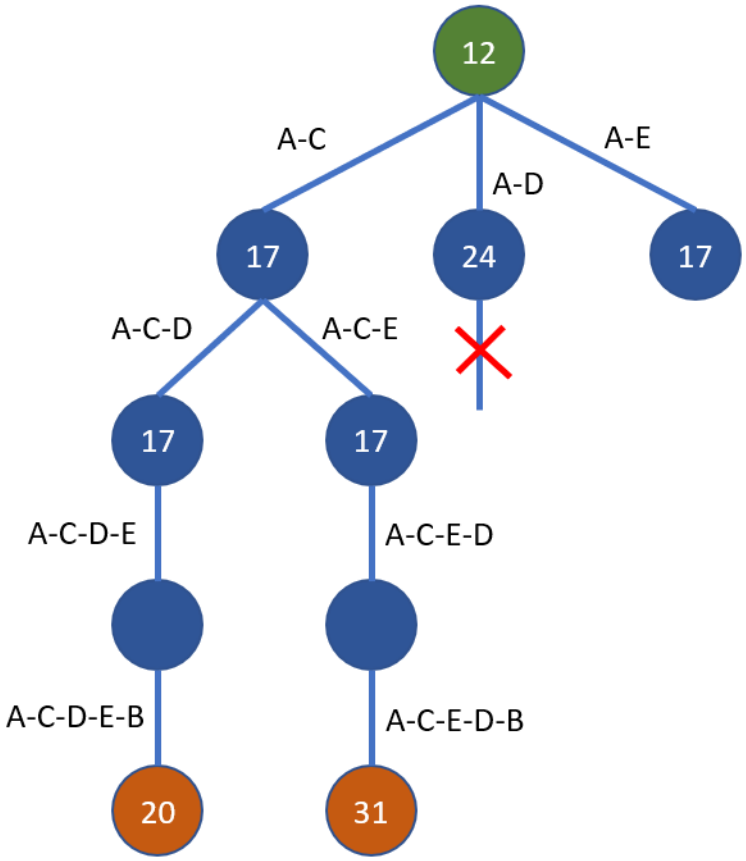


Figura 7.6 – Exploración del segundo nivel de una de las ramas.

Si no existiera ninguna rama sin podar el algoritmo habría acabado y ya se habría encontrado el camino óptimo. En este caso todavía queda una rama que tiene probabilidades de éxito por lo tanto hay que explorarla. El resultado de la exploración se muestra en la Figura 7.7. En esta nueva exploración se ha obtenido una nueva hoja con un valor mínimo de 19 de la ruta. La ruta de esta nueva hoja pasa a ser en ese momento la mejor ruta hasta el momento, que a la postre va a ser la definitiva ya que se ha terminado de explorar el árbol puesto que no quedan más ramas por verificar.

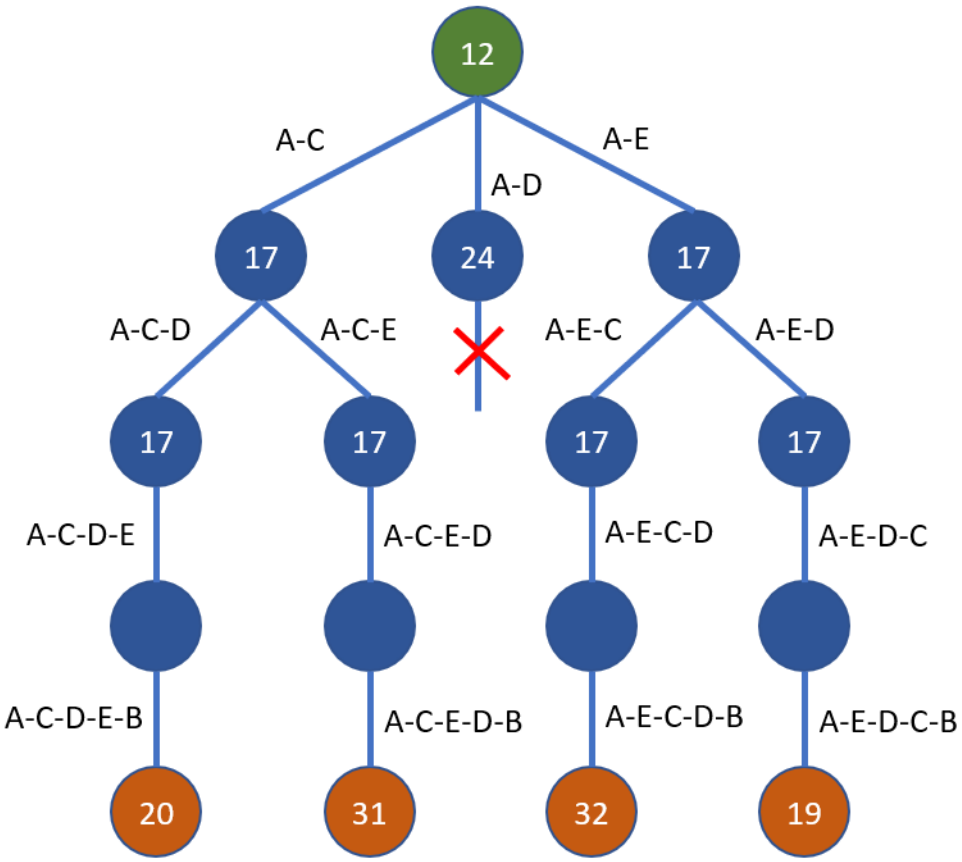


Figura 7.7 – Exploración de la segunda rama con probabilidades de éxito.

7.1.2. Pseudocódigo de ramificación y acotamiento

El algoritmo de ramificación y acotamiento para este proyecto hace uso de la clase nodo. La clase nodo, como se puede ver en la Figura 7.8 tiene 3 atributos, el de nivel, que marca en que nivel del árbol está ese nodo, el de camino, donde se guarda el recorrido del árbol hasta llegar al nodo, y el del límite, que marca el coste que como mínimo va a tener la ruta desde por la rama que está el nodo.

El Algoritmo 7.2, muestra el psuedocódigo del método de ramificación y acotamiento.

Algoritmo 7.1: Ramificación y acotamiento.

Entrada: origen: matrizDePesos: Matriz de enteros)
Salida: El camino entre los dos nodos, distancia entre los dos nodos

LÉXICO
longitudCamino: entero
rutaOptima: Array de enteros
distanciaOptima: real
distanciaMinima: real
PQ: cola que prioriza el nodo con el limite más bajo
NodoHijo: Nodo
NodoPadre: Nodo

ALGORITMO

```
longitudCamino ← dimension(matrizDePesos)
distanciaMinima ← ∞
nodoPadre ← Nodo(Nivel ← 0, camino ← |0|)
distanciaOptima ← 0
nodoPadre.limite ← hallarLimite(matrizDePesos, nodoPadre)
PQ ← añadir(nodoPadre)
nodoHijo ← Nodo()
Repetir_Mientras NoEstaVacio(PQ) Hacer
    NodoPadre ← sacarNodo(PQ)
    si nodoPadre.limite < distanciaMinima entonces
        nodoHijo.nivel ← nodoPadre.nivel + 1
        Para_Cada i ENTRE nodosSinExplorar(nodoPadre.camino) Hacer
            nodoHijo.camino ← nodoPadre.camino
            nodoHijo.camino ← añadir(i)
            si nodoHijo.nivel == longitudCamino − 3 entonces
                nodoHijo.camino ← añadirNodoFaltante()
                nodoHijo.camino ← añadirDestino()
                l ← calcularLongitud(matrizDePesos, nodoHijo)
                si l < distanciaMinima entonces
                    distanciaMinima ← l
                    distanciaOptima ← l
                    rutaOptima ← nodoHijo.camino
            en otro caso
                nodoHijo.limite ← hallarLimite(matrizDePesos, nodoHijo)
                si nodoHijo.limite < distanciaMinima entonces
                    PQ ← añadir(nodoHijo)
            nodoHijo ← Nodo(nivel ← nodoHijo.nivel)
Devolver rutaOptima, distanciaOptima
```

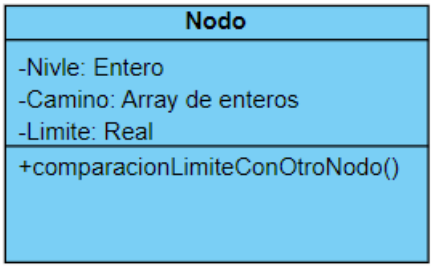


Figura 7.8 – Descripción de la clase Nodo.

La función heurística de la que hace uso el algoritmo de ramificación y acotamiento se define en el pseudocódigo del Algoritmo 7.2.

Algoritmo 7.2: Ramificación y acotamiento.

Entrada: origen: matrizDePesos: Matriz de enteros, nodo: Nodo)
Salida: caminoMinimo: real
LÉXICO
camino: Array de enteros
limite: real
longitudCamino: entero
nodosFaltantes: Array de enteros
ALGORITMO
$camino \leftarrow nodo.camino$
$longitudCamino \leftarrow dimension(matrizDePesos)$
$nodosFaltantes \leftarrow añadirNodosFaltantes(camino)$
$limite \leftarrow añadirLongitudCaminoRecorrido(camino)$
$limite \leftarrow limite + distanicaMinimadesdeNodo(camino| - 1|)$; **Para_Cada** i ENTRE $nodosFaltantes$ **Hacer**
  $limite \leftarrow limite + distanicaMinimadesdeNodo(i)$;
Devolver $limite$

7.2. Uso de la solución híbrida

El uso que se propone para esta solución es la de siempre y cuando la ruta a buscar tenga un numero mayor o igual a dos paradas, usar el algoritmo de Dijkstra para la creación del subgrafo. Una vez creado el subgrafo, ejecutar el algoritmo de ramificación y acotamiento siempre y cuando el coste computacional del mismo

sea aceptable. El coste computacional del método de ramificación y acotamiento dependera de dos factores, por un lado el número de paradas intermedias que se deseen hacer y por otro lado la tipografía del grafo.

Si el coste computacional del metodo de ramificación y acotamiento es inasumible, se ejecutará el algoritmo MMAS sobre el subgrafo. Las ventajas que dará el subgrafo generado al algoritmo MMAS son varias.

- Todas las hormigas tendrán éxito en la busqueda de rutas.
- Los puntos del grafo a explorar por las hormigas son mucho menos.
- Al ser la matriz reducida más pequeña, las hormigas que se necesitan soltar y las iteraciones que se necesitan realizar son menos que cuando se ejecuta MMAS sobre la matriz original del grafo.
- La ruta que se devuelve entre dos puntos implicados en la ruta, es decir, entre el origen y un punto intermedio, el destino y un punto intermedio o entre dos puentos intermedios es la óptima.

Para testear el código de la solución híbrida se incluyen dieciocho paradas intermedias. La ruta devuelta la ha realizado en menos de 15 segundos, empleando el algoritmo de ramificación y acotamiento. La ruta que se ha obtenido es la óptima como se puede observar en la Figura 7.9.

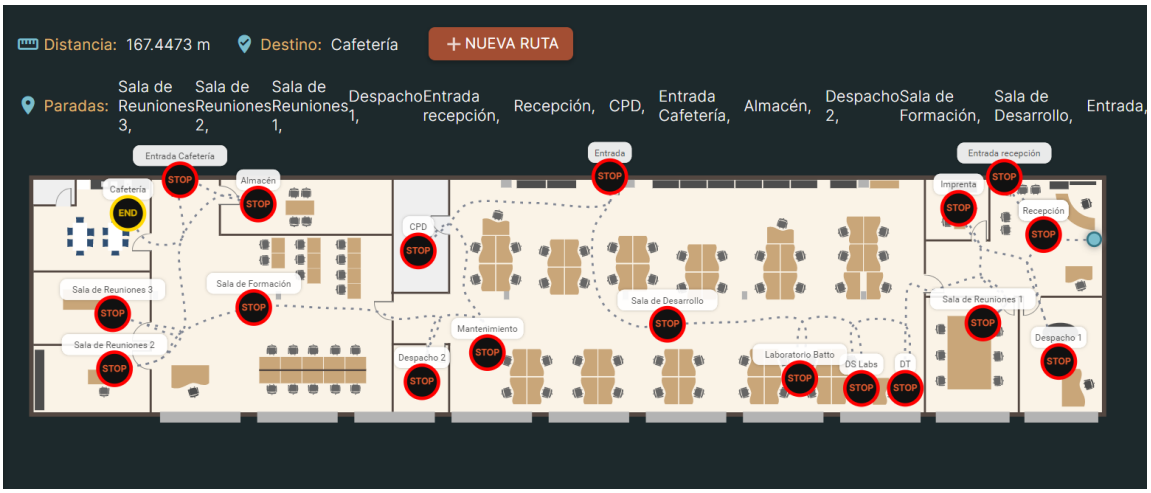


Figura 7.9 – Ruta encontrada por la solución híbrida con dieciocho puntos intermedios.

8

Pruebas de los diferentes algoritmos.

Se van a realizar varias pruebas sobre tres grafos bastante diferentes. El primer grafo, es el mostrado en la Figura 4.1, con alguna variante. Es un grafo no dirigido, con bastantes nodos de grado 1. Los puentes que se han detectado se han modificado empleando la solución mostrada en la Figura 6.6. Este grafo es propenso a que las hormigas colapsen en nodos sin salidas, ya que los nodos son de grados bajos, y tiene tendencia a que las hormigas recorran un ciclo y después se queden sin opciones. El grafo sin puentes cuenta con 89 nodos y 125 aristas $G(89, 125)$, se puede observar en la Figura 8.1.

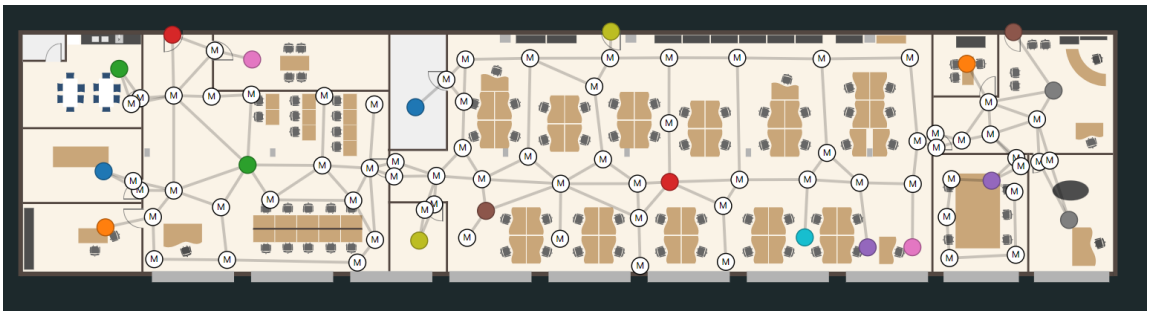


Figura 8.1 – Grafo no dirigido con varios nodos de grado uno.

El segundo grafo, Figura 8.2, que se utiliza para las pruebas tiene una topología bien distinta. Es un grafo mixto, es decir, tiene aristas dirigidas y aristas no dirigidas. No contiene ningún nodo de grado uno. No existen puentes en el grafo. La media del grado de los nodos es mayor que en el grafo anterior, sin ser muy alta tampoco. El grafo está representado por 93 nodos 180 aristas, $G(93, 180)$.

La principal característica del último grafo a probar, Figura 8.3, es su densidad de nodos y aristas. Es un grafo con más del doble de nodos que los otros dos grafos. También triplica en aristas a los grafos anteriores. Además de eso, es un grafo no dirigido, y con nodos, de media, con mayor grado de todas las pruebas realizadas. El grafo cuenta con 276 nodos y 389 aristas, $G(276, 389)$.

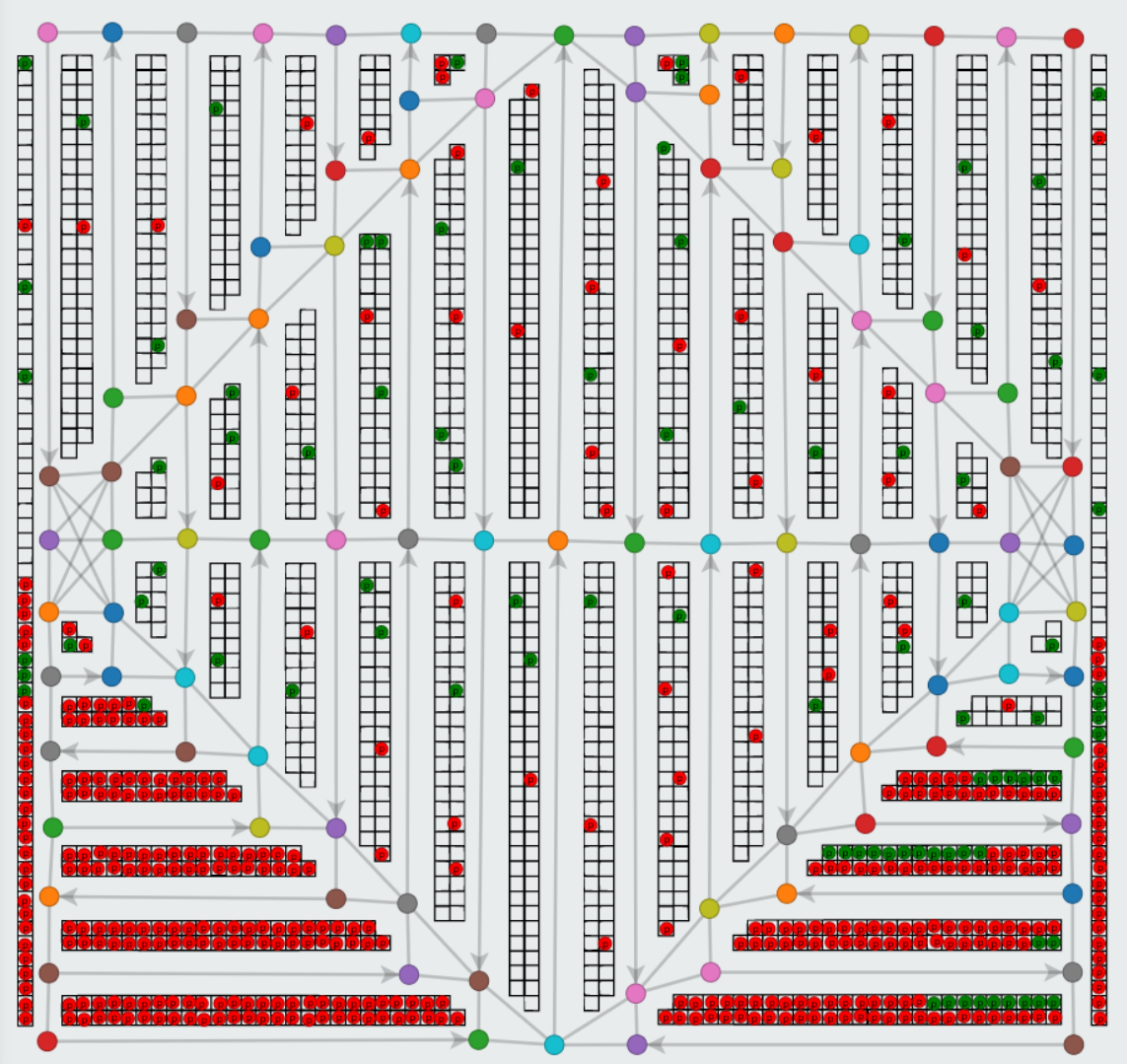


Figura 8.2 – Grafo mixto con aristas dirigidas y aristas no dirigidas.

Las pruebas que se van a realizar van a ir desde el cálculo de la ruta directa desde un origen a un destino, hasta el cálculo de la ruta con 25 paradas intermedias. En la

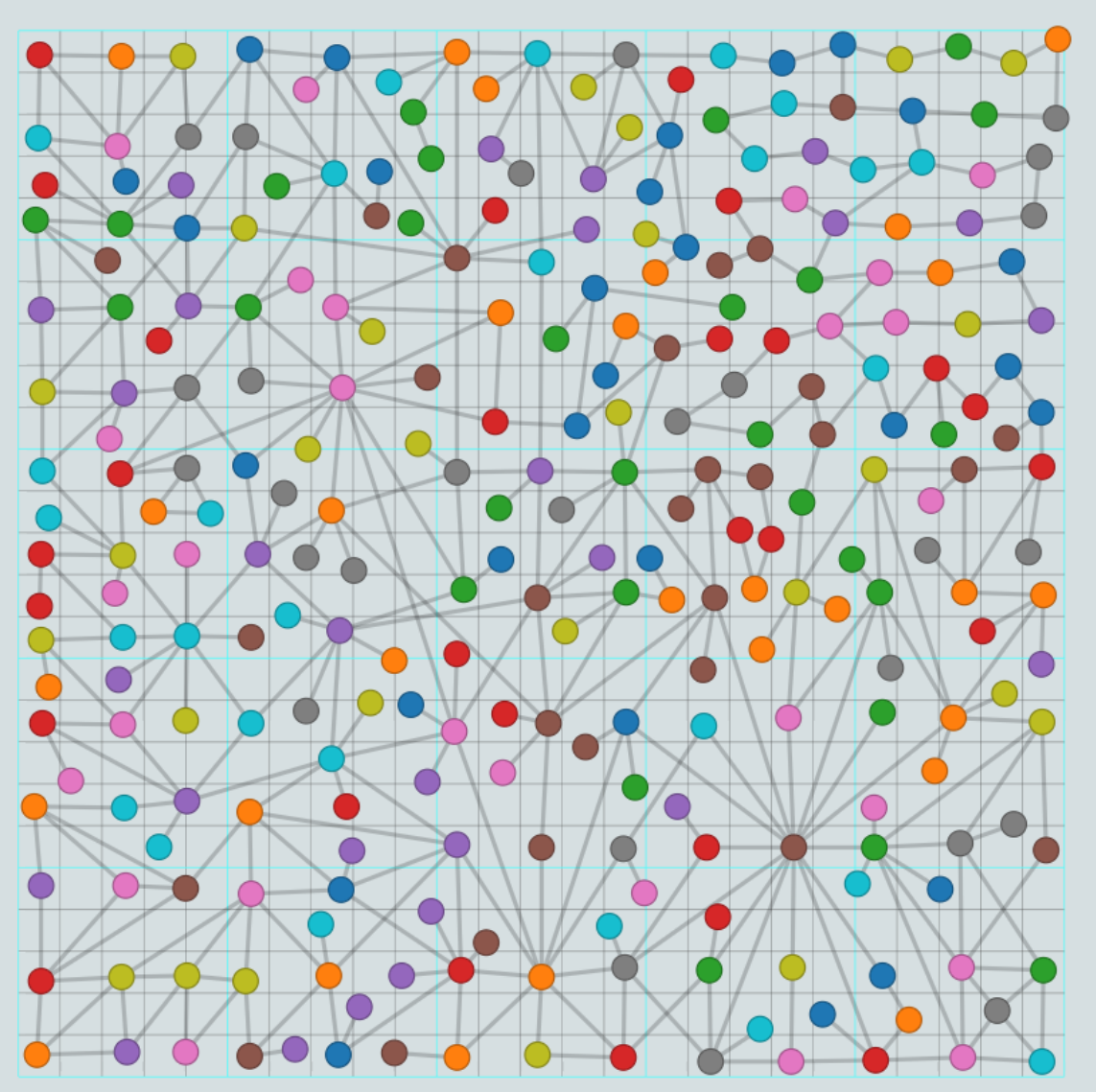


Figura 8.3 – Grafo denso con gran cantidad de nodos y de aristas.

Figura 8.4 se muestra la gráfica de crecimiento de las permutaciones cada vez que se añade una parada.

En las pruebas se van a ejecutar los siguientes algoritmos para poder comparar las rutas obtenidas. En primer lugar se ejecuta el algoritmo de Dijkstra, seguido se ejecuta el algoritmo ACO con cien hormigas en cada una de las cien iteraciones (ACO

Figura 8.4 – Gráfica de crecimiento de las posibles soluciones según el número de puntos intermedios.

100X100), en tercer lugar, se vuelve a probar ACO pero esta vez con mil hormigas en cada una de las cien iteraciones (ACO 100X1000). Una vez realizadas estas tres pruebas se pasa a evaluar la solución híbrida, con el algoritmo de Dijkstra junto con el de ramificación y acotamiento (D+R) en primer lugar, seguido de Dijkstra más ACO (D+ACO) en segundo lugar

8.1. Pruebas sobre un grafo no dirigido

Se realizan las pruebas en el grafo no dirigido, donde se va ha ejecutar el algoritmo de Dijkstra, el algoritmo de MMAS sobre la matriz del grafo original primero con 10 iteraciones y con 10 hormigas, luego con 100 iteraciones y 100 hormigas. También se va a testear el algoritmo de ramificación y acotamiento sobre la matriz reducida y el algoritmo MMAS sobre la matriz reducida también. Los resultados sobre las

pruebas del grafo no dirigido se pueden ver en la Tabla 8.1. El origen, el destino y los nodos intermedios en caso de tenerlos serán los mismos. Se intentará escoger tanto origen, como destino, como puntos intermedios más alejados entre ellos.

GRAFO NO DIRIGIDO										
P	**Dijkstra**	**ACO 100*100**			**ACO 100*1000**			**D+R**	**D+ACO**	
	T	T	σ	C	T	σ	C	T	T	σ
0	0.12”	24”	7 %	74	3’ 48”	1 %	674	–	–	–
1	0.13”	25”	10 %	56	4’ 9”	0 %	502	–	–	–
2	0.15”	29”	13 %	30	4’ 47”	0 %	253	0.15”	0.29”	0 %
3	0.18”	30”	21 %	16	4’ 59”	6 %	135	0.16”	0.41”	0 %
4	0.37”	33”	14 %	6	5’ 39”	13 %	66	0.15”	0.68”	0 %
5	1.58”	36”	74 %	2	5’ 52”	10 %	29	0.15”	1”	0 %
6	10”	36”	76 %	1	6’ 12”	55 %	15	0.19”	1”	0.3 %
7	1’ 20”	40”	78 %	4	6’ 22”	27 %	13	0.16”	2”	0 %
8	12’ 5”	42”	77 %	1	7’ 6”	58 %	10	0.17”	4”	14 %
9	2h 9’	X	X	0	8’ 14”	198 %	1	0.2”	6”	12 %
10					8’ 56”	75 %	5	0.23”	8”	15 %
11					9’ 20”	52 %	10	0.31”	12”	29 %
12					10’ 9”	64 %	6	1”	17”	18 %
14					10’ 42”	55 %	4	1.3”	23”	37 %
15					11’ 3”	70 %	4	0.5”	31”	40 %
16					10’ 55”	97 %	4	1.2”	40”	49 %
17					X	X	0	3.5”	53”	61 %
18								4”	1’ 9”	68 %
19								13”	4’ 49”	67 %

Tabla 8.1 – Pruebas de los algoritmos sobre el grafo no dirigido.

En la gráfica de la Figura 8.5 se puede observar que con pocas paradas intermedias el algoritmo de Dijkstra responde bien, pero hay un punto de inflexión cuando hay 8 paradas intermedias, donde el algoritmo comienza a crecer y se vuelve ineficiente.

El algoritmo de ramificación y acotamiento, tiene un buen comportamiento, ya que la función de poda de las ramas da buenos resultados en este grafo. El tiempo de computo con 18 paradas intermedias, con un conjunto de soluciones posibles mayor a 10^15, no sobrepasa los 13 segundos, lo cual es un resultado muy bueno.

El algoritmo MMAS tiene un comportamiento lineal, donde el tiempo de cómputo depende de las iteraciones y de las hormigas parametrizadas.

Figura 8.5 – Tiempo de cómputo de los algoritmos en un grafo no dirigido.

Por otro lado, se puede observar en la gráfica de la Figura 8.6 que cuando aplicamos el algoritmo MMAS sobre la matriz del grafo original, las hormigas que llegan a destino son pocas y a medida que se van añadiendo puntos intermedios a la ruta son cada vez menos hasta llegar a un punto que ninguna hormiga llega a encontrar una ruta.

En la gráfica de la Figura 8.7, vemos la calidad del camino encontrado por los algoritmos heurísticos MMAS. Se han realizado tres pruebas de este algoritmo, dos realizadas sobre la matriz del grafo original y otra realizada sobre la matriz del grafo reducido. En las pruebas de MMAS sobre el grafo original se han soltado cien hormigas en cien iteraciones en la primera prueba y mil hormigas con cien iteraciones en la segunda prueba. En el algoritmo MMAS sobre el grafo reducido, al tener la seguridad de que todas las hormigas van a tener éxito en la busqueda de una ruta, se han soltado $númeroPuntosIntermedios + 2 * 10$ hormigas y el mismo número de iteraciones.

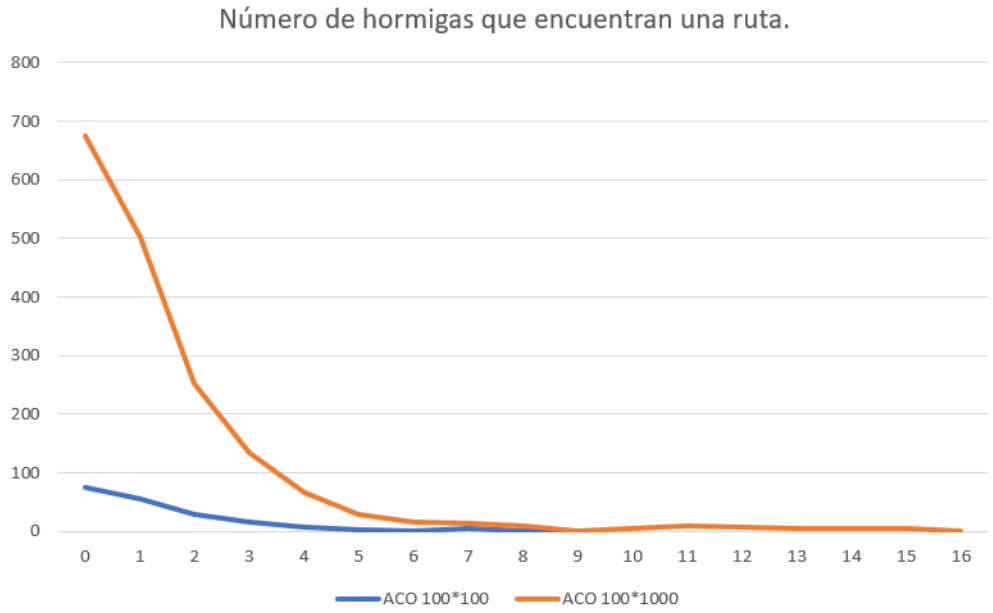


Figura 8.6 – Éxito de las hormigas para encontrar una ruta en el grafo no dirigido sobre la matriz original.

8.2. Conclusiones de las pruebas sobre el grafo no dirigido.

Observando las pruebas realizadas sobre el grafo no dirigido se pueden sacar las siguientes conclusiones.

- El algoritmo de Dijkstra, permutando las paradas intermedias, da buenos resultados hasta las seis paradas intermedias, luego el tiempo de cómputo se dispara de tal modo que hace que el algoritmo resulte ineficiente.

- Con el algoritmo MMAS sobre el grafo original se puede observar que desde un inicio se tiene una gran perdida de hormigas. Según se van añadiendo paradas intermedias, la perdida de hormigas se agudece hasta llegar a un punto en el que el algoritmo no es capaz de hacer llegar una hormiga al destino.

- La calidad del camino encontrada por el algoritmo MMAS se ve afectada por el número de paradas.

Figura 8.7 – Calidad del camino encontrado por el algoritmo MMASsobre el grafo no dirigido.

- Se observa una relación de perdida de calidad del camino encontrado con el número de hormigas que obtienen una solución en los algoritmos heurísticos. El algoritmo MMAS sobre el grafo reducido que no tiene perdida de hormigas es que mejor calidad de ruta ofrece en el resultado entre los algoritmos MMAS.

- El algoritmo de ramificación y acotamiento es el que mejor comportamiento presenta, incluso con el número máximo de paradas intermedias que se puede dar en este grafo. Con el máximo de paradas intermedias (18) solo tarda 13 segundos en ofrecer el resultado óptimo.

8.3. Pruebas sobre un grafo mixto.

Las pruebas realizadas a continuación se van a realizar sobre el grafo mostrado en la Figura 8.2. Este grafo contiene aristas dirigidas y aristas no dirigidas. Las pruebas que se pueden ver en la Tabla 8.2, se han realizado en los mismos términos explicados en la Sección 8.1, pero en este grafo al ser más grande se han utilizado hasta 25 puntos intermedios. Al aumentar los puntos intermedios y las dimensiones del grafo, existen algunos casos en los que no se ha podido sacar la solución óptima,

y tampoco se puede calcular la desviación de las rutas heurísticas, se señala con (??) en la tabla.

GRAFO MIXTO										
P	**Dijkstra**	**ACO 100*100**			**ACO 100*1000**			**D+R**	**D+ACO**	
	T	T	σ	C	T	σ	C	T	T	σ
0	0.15”	24”	0 %	1083	7’ 55”	0 %	11122	–	–	–
1	0.16”	1’ 6”	0 %	959	10’ 18”	0 %	9437	–	–	–
2	0.18”	1’ 8”	5 %	473	11’ 14”	5 %	4733	0.14”	0.41”	0 %
3	0.21”	1’ 15”	19 %	405	12’ 10”	2 %	4261	0.37”	0.59”	0 %
4	0.43”	1’ 18”	20 %	376	12’ 53”	11 %	3824	0.21”	0.64”	0 %
5	1.67”	1’ 22”	32 %	349	13’ 43”	17 %	3520	0.23”	1”	0 %
6	10”	1’ 27”	61 %	227	14’ 30”	15 %	2615	0.26”	2”	0.3 %
7	1’ 26”	1’ 33”	48 %	236	15’ 32”	20 %	2452	0.21”	3”	0 %
8	13’ 3”	1’ 37”	53 %	246	16’ 9”	35 %	2342	0.18”	4”	14 %
9	2h 8’	1’ 44”	54 %	206	17’ 25”	46 %	2005	0.22”	6”	12 %
10		1’ 51”	62 %	115	18’ 22”	50 %	1427	0.41”	9”	5 %
11		1’ 55”	94 %	136	19’ 19”	67 %	1146	0.79”	12”	7 %
12		2’ 6”	85 %	96	22’ 4”	47 %	816	2”	17”	22 %
13		2’ 21”	90 %	82	23’ 29”	61 %	863	36”	23”	29 %
14		2’ 31”	126 %	93	25’ 11”	67 %	851	25”	30”	43 %
15		2’ 35”	122 %	73	26’ 47”	87 %	767	1’ 24”	40”	43 %
16		2’ 40”	129 %	82	27’ 35”	90 %	751	2’ 34”	51”	43 %
17		2’ 51”	120 %	59	29’ 58”	89 %	612	9’ 43”	1’ 7”	58 %
18		2’ 57”	140 %	24	29’ 52”	90 %	294	56’ 41”	1’ 26”	31 %
19		3’	??	16	30’ 17”	??	303		1’ 49”	??
20		3’ 2”	??	34	31’ 7”	??	274		2’ 16”	??
21		3’ 14”	??	26	31’ 35”	??	243		2’ 51”	??
22		3’ 20”	??	29	33’ 14”	??	270		3’ 27”	??
23		3’ 16”	??	23	33’ 58”	??	264		4’ 08”	??
24		3’ 26”	??	33	34’ 19”	??	233		4’ 57”	??
25		3’ 19”	??	23	33’ 58”	??	272		5’ 58”	??

Tabla 8.2 – Pruebas de los algoritmos sobre el grafo mixto.

Los tiempos de computo de los diferentes algoritmos se muestran en la gráfica de la Figura 8.8. El dato más reseñable es que el algoritmo de ramificación y acotamiento a partir del punto intermedio 17 deja de ser eficiente.

En el gráfico de la Figura 8.9 se puede observar que el nivel de éxito de las hormigas para encontrar una ruta utilizando la matriz original es mucho mayor que

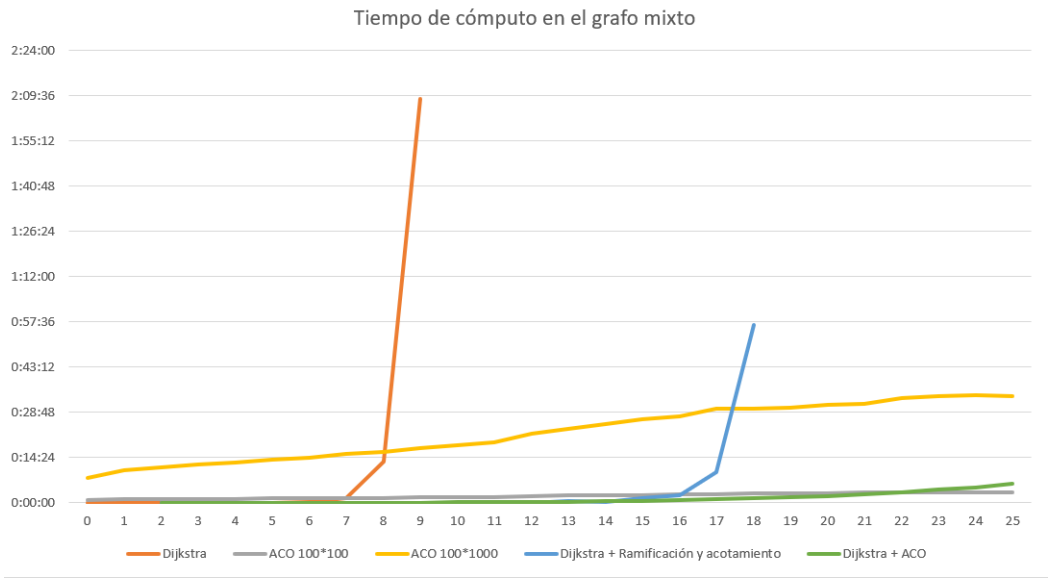


Figura 8.8 – Tiempo de cómputo de los algoritmos en un grafo mixto.

en el grafo no dirigido. Esto se debe a la tipología del grafo. El grafo dirigido no tiene nodos de grado 1, y las hormigas tienen más facilidad para salir de los bucles.

En este grafo las hormigas son capaces de buscar rutas con hasta 25 puntos intermedios, pero a partir de 18 puntos intermedios son muy pocas las que llegan a tener éxito en la búsqueda.

La calidad del camino encontrada por el algoritmo MMAS empeora a medida que se van añadiendo puntos intermedios, al igual que pasaba en el grafo no dirigido. Cuando se ejecuta MMAS sobre la matriz original la calidad de los caminos es sustancialmente peor que cuando se ejecuta sobre la matriz reducida, como bien se puede observar en la gráfica de la Figura 8.10.

8.4. Conclusiones de las pruebas sobre el grafo mixto.

Las conclusiones que se pueden destacar sobre las pruebas realizadas sobre el grafo mixto son las siguientes.

- Los algoritmos que encuentran la solución óptima, como el de Dijkstra o el de ramificación y acotamiento son solo eficientes para un número determinado de

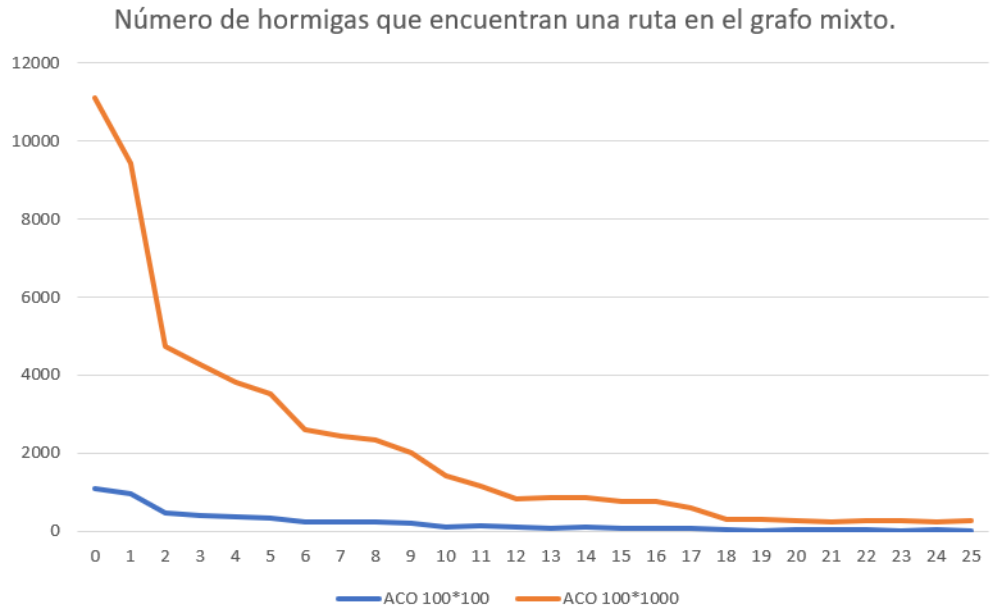


Figura 8.9 – Éxito de las hormigas para encontrar una ruta en el grafo mixto sobre la matriz original.

paradas. Sobrepasado ese número de puntos intermedios el algoritmo se hace excesivamente costoso.

- El algoritmo de ramificación y acotamiento se comporta bastante mejor que el algoritmo de Dijkstra. Puede llegar a obtener la solución óptima en un espacio de tiempo muy reducido, con hasta el doble de puntos intermedios que el algoritmo de Dijkstra.

- El algoritmo MMASse comporta mucho mejor que en grafo no dirigido. Las tres pruebas del algoritmo llegan a obtener una ruta con hasta 25 puntos intermedios. El tiempo de cómputo tiene un crecimiento lineal, el cual está totalmente relacionado con el número de hormigas que se saquen a explorar rutas y las iteraciones de las mismas.

- A partir de 18 puntos intermedios, la única solución viable para encontrar una ruta es usar el algoritmo MMAS.

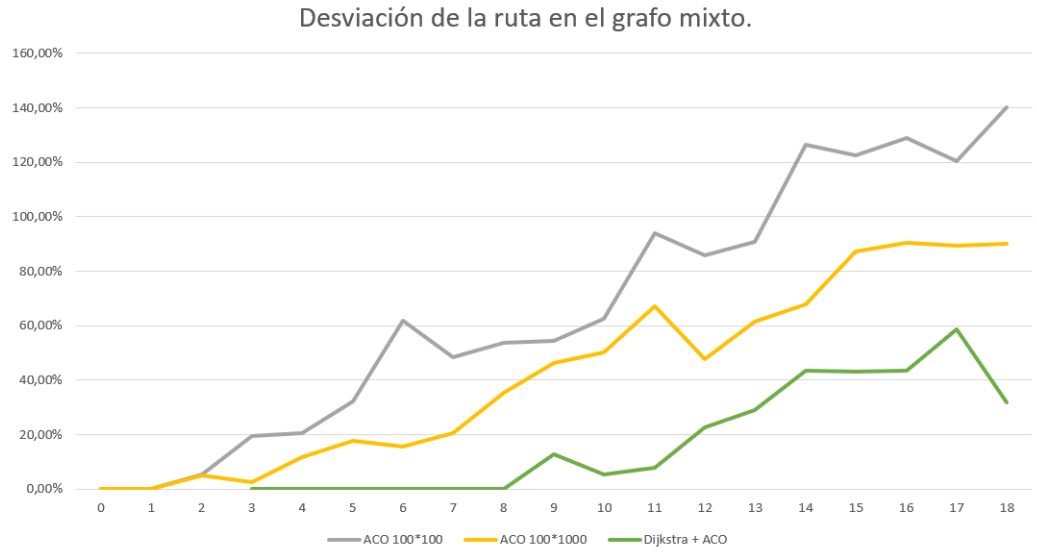


Figura 8.10 – Calidad del camino encontrado por el algoritmo MMASsobre el grafo mixto.

8.5. Pruebas sobre un grafo denso.

Con este grafo se quiere poner a prueba a los algoritmos desarrollados. Este grafo tiene como características un gran número de nodos y también un gran número de aristas, el orden del grafo es $G(276{,}389)$, cuyos valores triplican a los de los grafos anteriores. Se quiere observar el comportamiento de cada algoritmo en una situación en la que las posibilidades de rutas posibles aumentan enormemente respecto a las pruebas anteriores. Los resultados de las pruebas se pueden leer en la Tabla 8.3.

Igual que en los grafos anteriores, se saca la gráfica del tiempo de cómputo de cada prueba, como se puede observar en la Figura 8.11. Como cabía esperar, el tiempo de computo en los algoritmos exactos ha crecido en todas las pruebas con respecto a los grafos anteriores. En cambio el algoritmo MMAS sobre la matriz reducida del grafo a mantenido un tiempo constante, parecido al empleado en las pruebas realizadas en los grafos anteriores.

Por otro lado, el comportamiento del algoritmo MMAS sobre la matriz original del grafo es muy pobre, como se puede observar en la gráfica de la Figura 8.12. Muy pocas hormigas son capaces de encontrar una ruta. La prueba que suelta mil hormigas en cada iteración, siendo el número de iteraciones de cien, lo que hace un total de cien mil hormigas, solo 87 de ellas han sido capaces de encontrar una ruta sin

GRAFO DENSO										
P	**Dijkstra**	**ACO 100*100**			**ACO 100*1000**			**D+R**	**D+ACO**	
	T	T	σ	C	T	σ	C	T	T	σ
0	0.5”	1’ 11”	0 %	15	11’ 27”	0 %	87	–	–	–
1	0.59”	1’ 15”	6 %	7	11’ 54”	14 %	56	–	–	–
2	0.6”	X	X	0	12’ 46”	57 %	7	0.47”	0.6”	0 %
3	1”				13’ 16”	102 %	1	0.5”	0.73”	0 %
4	3”				X	X	0	0.52”	1”	0 %
5	13”							0.48”	1.42”	0 %
6	1’ 29”							0.58”	2”	0 %
7	11’ 37”							0.52”	3”	1.4 %
8	1h 45’ 25”							0.63”	4”	1.4 %
9	17h 24’ 59”							0.63”	6”	7 %
10								0.75”	9”	11 %
11								0.75”	13”	28 %
12								1”	17”	34 %
13								1”	24”	39 %
14								2”	32”	41 %
15								9”	40”	32 %
16								23”	52”	42 %
17								42”	1’ 8”	45 %
18								3’ 47”	1’ 24”	40 %
19								8’	1’ 47”	61 %
20								37’ 50”	2’ 14”	61
21									2’ 45”	??
22									3’ 21”	??
23									4’ 07”	??
24									4’ 54”	??
25									5’ 54”	??

Tabla 8.3 – Pruebas de los algoritmos sobre el grafo denso.

puntos intermedios en ella. Por lo tanto el algoritmo MMAS sobre la matriz original del grafo no es óptimo para este tipo de grafos.

La calidad de la ruta encontrada por el algoritmo MMAS sobre la matriz original no se va a analizar, ya que solo han sido capaz de encontrar una ruta con hasta cuatro puntos intermedios. En cambio el algoritmo MMAS sobre la matriz reducida ha llegado mucho más lejos de lo que lo ha hecho ningún otro. La gráfica de la

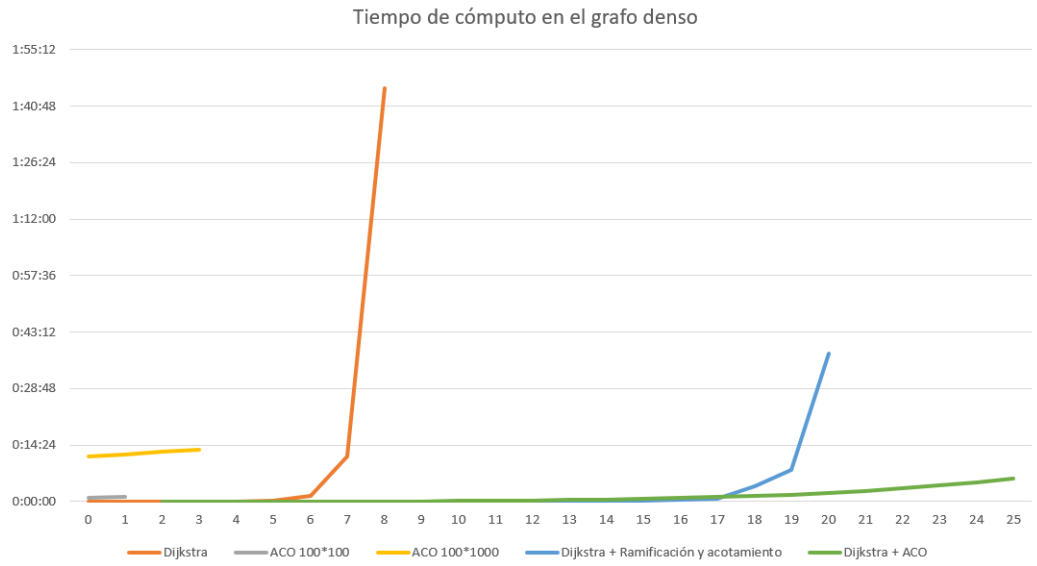


Figura 8.11 – Tiempo de cómputo de los algoritmos en un grafo denso.

Figura 8.13 muestra una desviación de alrededor del 40 % en el peor de los casos, aunque a partir de 21 puntos intermedios no se puede saber la calidad del camino ya que no se ha podido obtener el óptimo.

8.6. Conclusiones de las pruebas sobre el grafo denso.

Las conclusiones que se pueden obtener con las pruebas realizadas sobre un grafo denso son las siguientes.

- Las pruebas realizadas sobre la matriz reducida del grafo, como son la de ramificación y acotamiento, y la del algoritmo MMAS en la versión de la matriz reducida, no se ven afectadas por el aumento de nodos y aristas del grafo original.

- El algoritmo MMASutilizando la matriz original del grafo es totalmente ineficaz para este tipo de grafos.

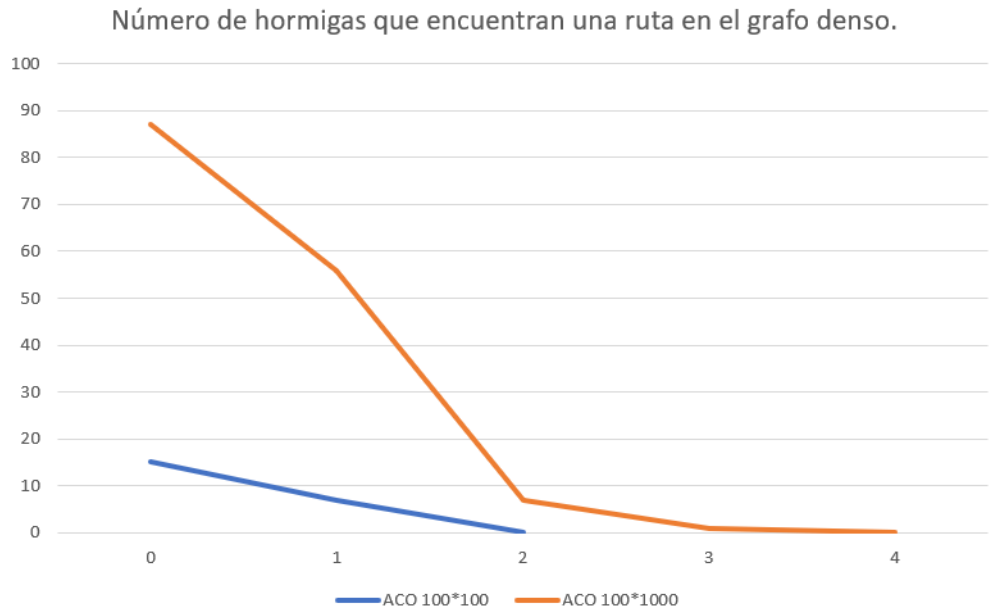


Figura 8.12 – Éxito de las hormigas para encontrar una ruta en el grafo denso sobre la matriz original.

8.7. Configuración de la solución híbrida determinada de manera empírica

Con los análisis realizados de las pruebas realizadas sobre los tres diferentes tipos de grafos, se ha realizado la siguiente configuración para la búsqueda de rutas dependiendo del número de puntos intermedios que contenga.

- Para rutas que no contengan puntos intermedios o solamente contenga un punto intermedio se ejecuta el algoritmo de Dijkstra.
- Para rutas que contengan entre dos y catorce puntos intermedios se empleará la solución híbrida de Dikjstra más ramificación y acotamiento.
- Para rutas de más de catorce puntos intermedios se empleara la solución híbrida de Dijkstra más ACO.

Figura 8.13 – Calidad del camino encontrado por el algoritmo MMASsobre el grafo denso.

9 Adaptar el código al entorno de la aplicación

Este trabajo forma parte de un proyecto mayor llamado Batto. La aplicación Batto cuenta, entre otras cosas, con un frontend, con un backend, una app móvil, una base de datos MySQL y otra base de datos MongoDB. Una de las tareas incluidas en el trabajo, era integrar el algoritmo de navegación en Batto.

La arquitectura de Batto es compleja, cada componente de Batto está dockerizado. El algoritmo de navegación va ha estar aislado del resto, encapsulado en un docker. El docker de navegación tiene que interactuar con otros tres dockers, el del backend, la base de datos MySQL y la base de datos MongoDB, ya que estas 3 estructuras también están dockerizadas. Se muestra una imagen de la arquitectura de Batto en el momento de la redacción de este documento en la Figura 9.1. Esta arquitectura tendrá cambios en un futuro.

Los tareas a realizar son.

- Integrar en el Backend las llamada al algoritmo de navegación y trabajar la respuesta.
- Mostrar en el Frontend la ruta recibida por el Backend
- Implementar la pantalla de navegación en una app móvil.

9.1. Backend

El Backend con el que se va a trabajar utiliza el framework Grails[1].

[1] https://grails.org/

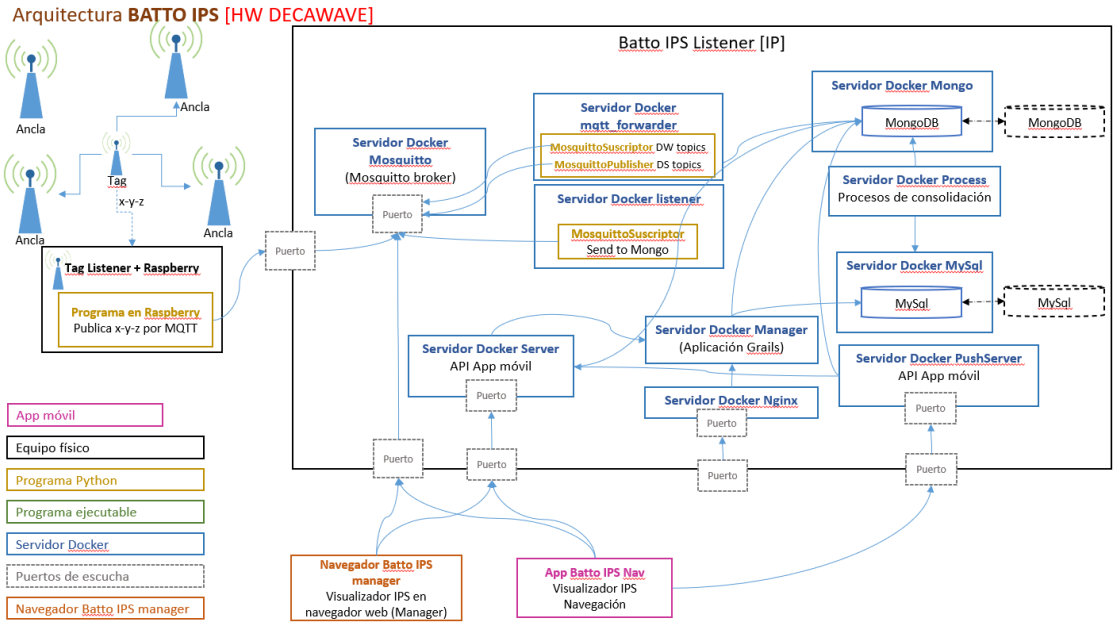


Figura 9.1 – Arquitectura de la aplicación Batto.

Figura 9.2 – Logotipo de Grails [2]

Grails es un framwork para aplicaciones web libre, el lenguaje que utiliza es Groovy, el cual está basado en java. Sus principales características son tres. No tiene configuración XML, entorno de desarrollo preparado para funcionar desde el primer momento y funcionalidad disponible mediante métodos dinámicos.

En Grails se han realizado varias tareas. Primero se han recibido los parámetros mandados desde el Frontend o de la aplicación móvil para el cálculo de ruta. Una vez se tiene los datos se mandan al docker del algoritmo de navegación para que calcule la ruta. Cuando se recibe la ruta desde el docker del algoritmo de navegación se trabaja para mandarla de manera óptima al Frontend y a la aplicación móvil.

[2] https://grails.org/images

También se ha programado el Backend para que añada o borre nodos o aristas en la base de datos MySQL. Esta acción se realiza a demanda del Frontend, la aplicación móvil no tiene esta funcionalidad.

9.2. Frontend

El Frontend utiliza el framework AngularJS[3]. Esta basado en JavaScript de código abierto. AngularJS está mantenido por Google y es utilizado para crear y mantener aplicaciones web de una sola página.

En el Frontend de la aplicación web es donde más funcionalidad se le ha dado a la navegación.

Una de las funciones es la de incluir los parámetros para el cálculo de rutas. Para ello se han incluido cuatro selectores, se muestra en la Figura 9.3, las cuales son.

- Nodo como destino
- Tag o Hueco de Picking como destino
- Nodo como parada, selección múltiple.
- Tag o Hueco de Picking como parada, selección múltiple.

Además en esta página se puede visualizar el grafo y ver las estancias, como se puede apreciar en la Figura 9.4. Otra funcionalidad de esta página es poder hacer zoom del mapa y controlar los margenes del mapa para que el mapa no pueda salir de la pantalla.

Otra pantalla desarrollada en este proyecto es la de la visualización de la ruta. En esta pantalla se muestra la ruta encontrada y la recorre en forma de bucle una bola para poder verla con más claridad. Además se muestra la distancia de la ruta y los valores seleccionados para la búsqueda, Figura 9.5.

Para poder interactuar con el mapa, se ha utilizado la librería D3.js[4]. Esta librería permite interactuar con formatos "svg", el cual es el formato con el que esta creado el mapa. Además permite añadir animaciones a la página.

[3] https://angularjs.org/
[4] https://d3js.org/

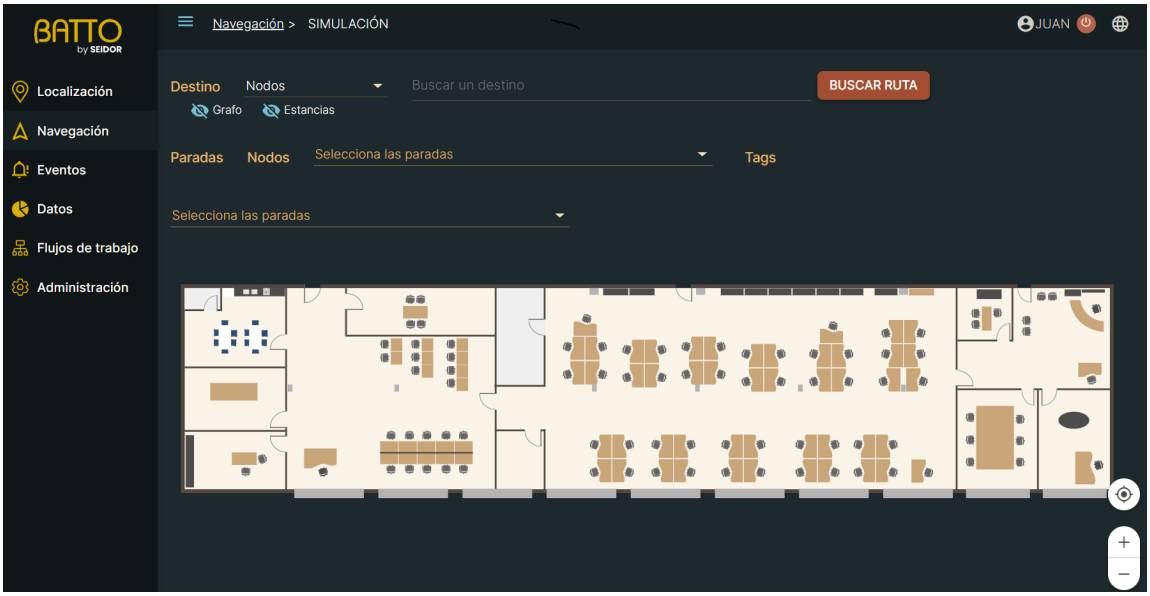


Figura 9.3 – Página de selección de ruta.

9.3. Aplicación móvil

Por último, se ha desarrollado la navegación de la aplicación móvil. La aplicación móvil esta desarrollada con el framework Ionic[5] en su sexta versión. Ionic permite desarrollar aplicaciones para iOS nativo, Android y la web, desde una única base de código. Ionic se puede integrar, entre otros, con Angular[6] que es con el que se ha trabajado en la aplicación móvil. En Ionic también se ha introducido la librería D3.js, explicada en la Sección 9.2, para interactuar con el mapa.

La navegación consta de dos pantalla, la de selección de ruta y la de mostrar y seguir la ruta.

La pantalla de selección de ruta, Figura 9.6a, Figura 9.6b, Figura 9.7a y Figura 9.7b, tiene las siguientes funcionalidades.

- Seleccionar destino
- Seleccionar paradas intermedias
- Ver en el mapa los posibles destinos fijos, es decir, no se incluyen ni los tags ni los huecos de picking.

[5] https://ionicframework.com/
[6] https://angular.io/

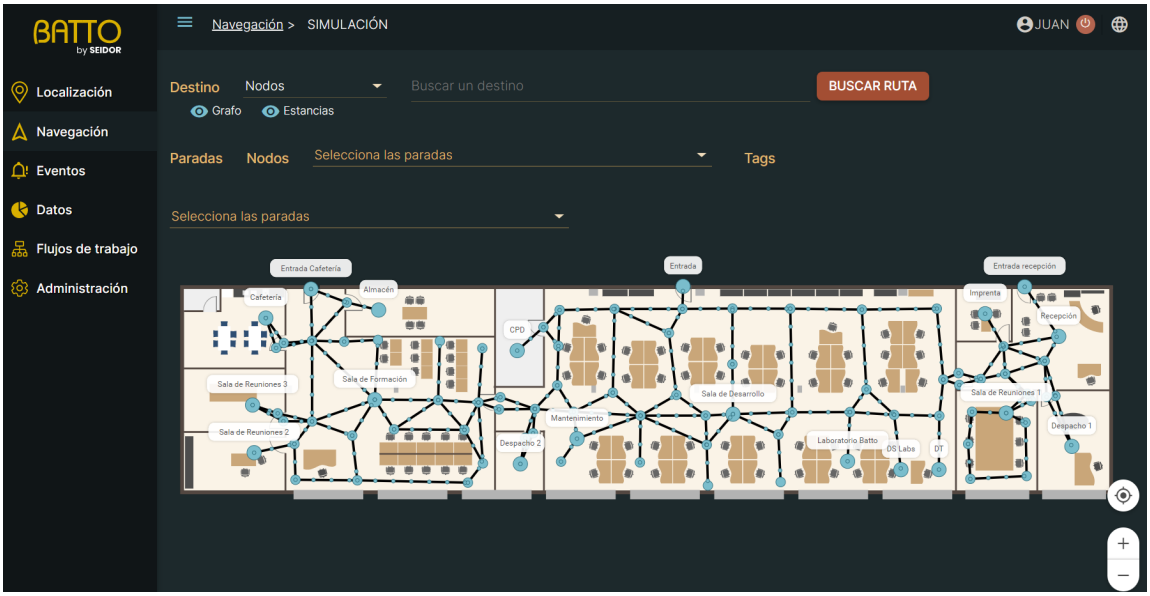


Figura 9.4 – Visualización del grafo y de las estancias de la página de selección de rutas.

- Se centrar la posición actual en la pantalla del móvil.
- Mostrar en la pantalla las paradas seleccionadas para navegar hasta ellas.
- Dibujar en el mapa las paradas seleccionadas.
- Poder hacer zoom sobre el mapa.
- Que el mapa no escape por los bordes de la pantalla.

La pantalla de mostrar ruta, Figura 9.8a y Figura 9.8b, es la otra pantalla desarrollada en la aplicación móvil.

Antes de explicar la pantalla se quiere explicar un detalle de la aplicación móvil. Cuando entras en la aplicación móvil, es requisito necesario asociarlo a un tag, que será el dispositivo que lleve encima el propietario del móvil. Cuando se dice que se hace un seguimiento del móvil por la ruta, en realidad el seguimiento es al tag asociado al móvil.

Esta pantalla tiene las siguientes funcionalidades.

- Mostrar la ruta dibujada en el mapa.
- Mostrar el tiempo estimado de ruta.

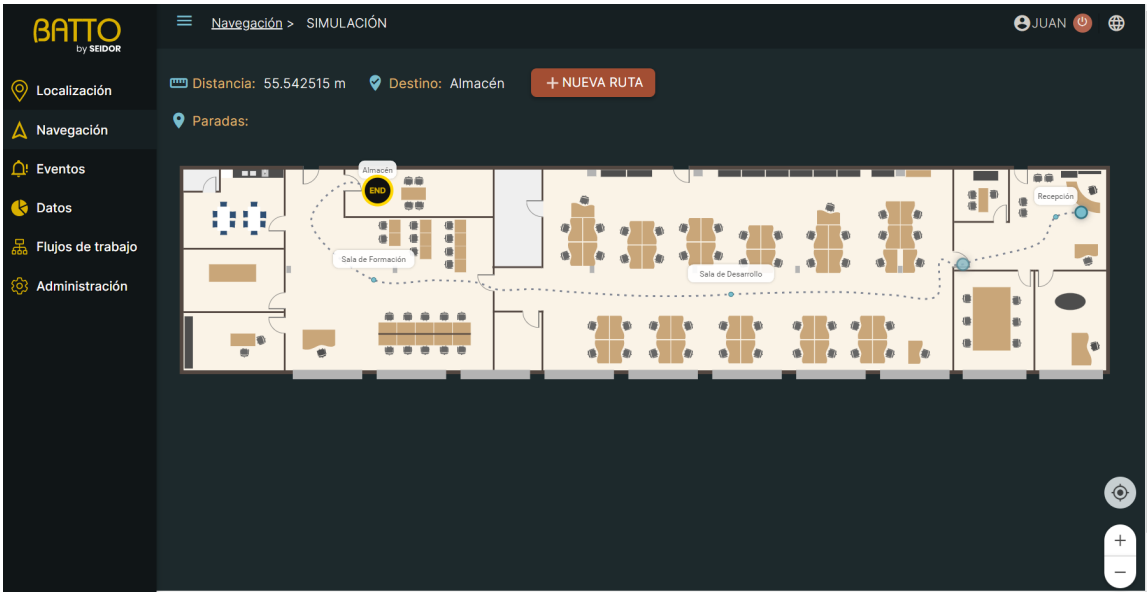
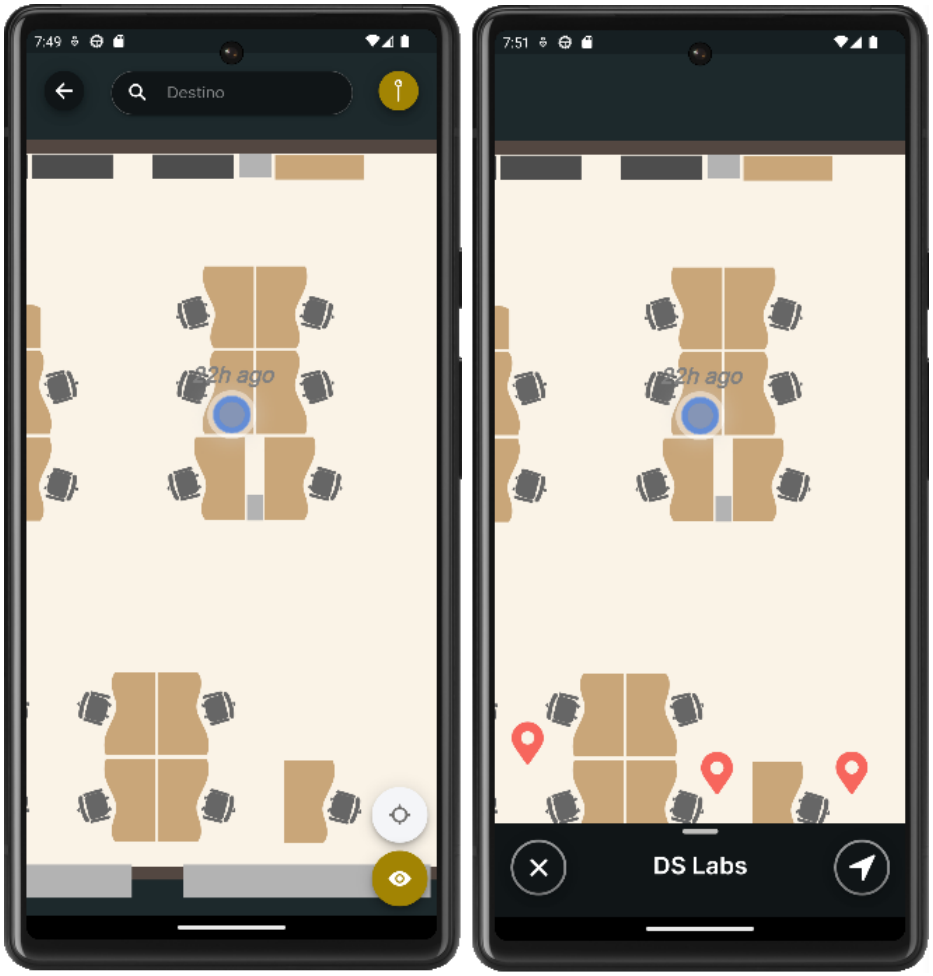


Figura 9.5 – Visualización de la ruta en la aplicación web.

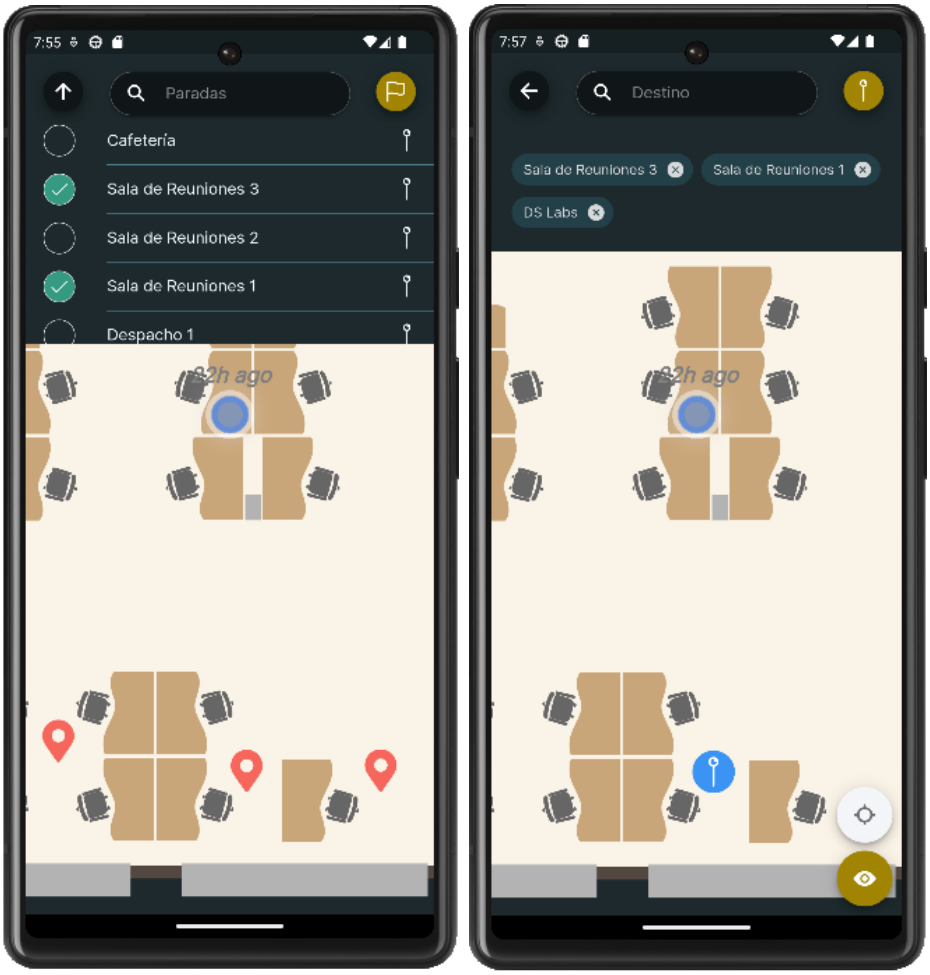
- Mostrar la distancia de la ruta.
- Seguimiento del dispositivo móvil a través de la ruta.
- Poder ver la planificación de la ruta.



(a) Pantalla de navegación de la aplicación móvil.

(b) Se ha seleccionado un destino de los mostrados en la página y te da la posibilidad de buscar la ruta hasta él.

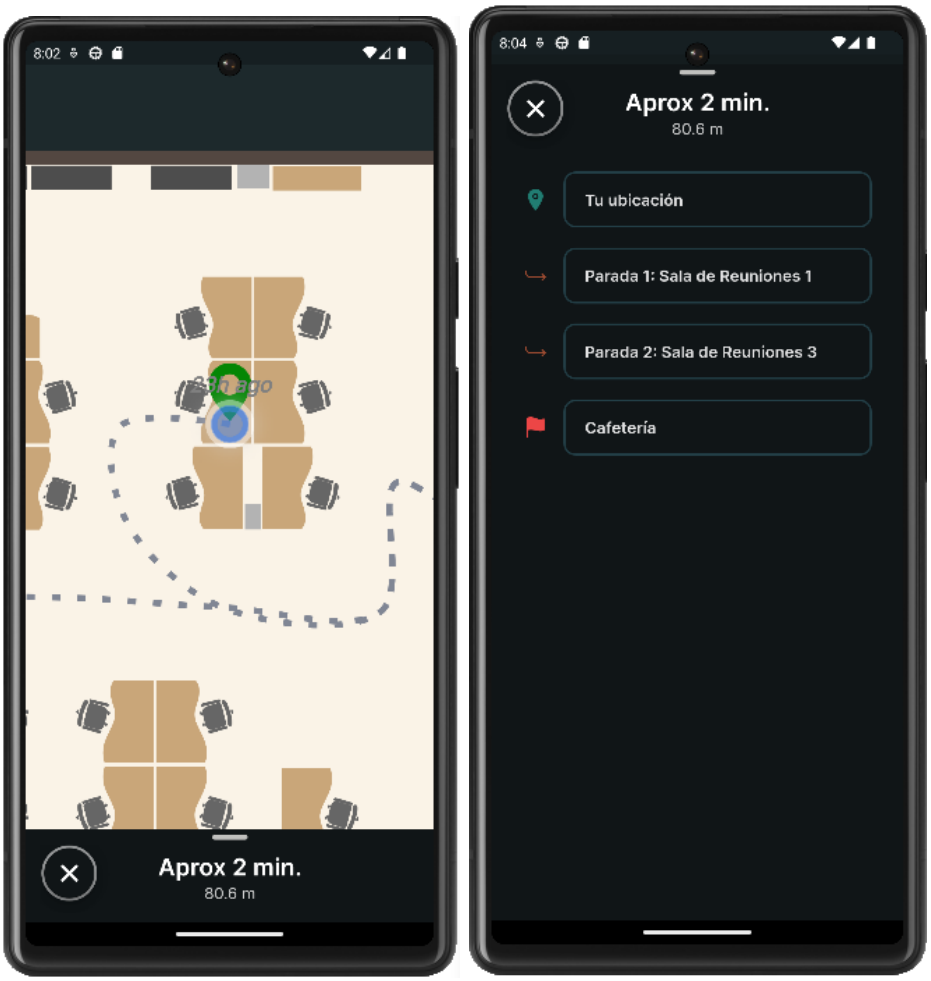
Figura 9.6 – Diferentes situaciones de la pantalla de navegación en la APP - 1



(a) Selección de las paradas intermedias.

(b) Visualización de las paradas en la pantalla y en el mapa.

Figura 9.7 – Diferentes situaciones de la pantalla de navegación en la APP - 2



(a) Mostrar y seguimiento de la ruta.

(b) Mostrar la planificación de la ruta.

Figura 9.8 – Diferentes situaciones de la pantalla de mostrar ruta en la APP

10

Conclusiones generales e investigaciones a futuro.

Después de todas las pruebas realizadas, y haberlas analizado en conjunto por cada tipo de grafo se pueden realizar las siguientes conclusiones generales.

- La solución híbrida es la que mejor resultados ha arrojado en todos los tipos de grafos. El motivo es que a esta solución no le afecta si el grafo es dirigido o no y el número de nodos y aristas que contenga.

- El algoritmo de ramificación y acotamiento es válido cuando se usa hasta un número determinado de puntos intermedios, ya que a partir de ese punto el tiempo de computo es inasumible.

- El algoritmo MMAS desplegado sobre la matriz reducida, es idoneo para la búsqueda de rutas con un número de puntos intermedios alto. Por otro lado, dependiendo del grafo, habría que ajustar los valores heurísticos, el número de hormigas desplegadas y las iteraciones realizadas para aumentar la calidad de la ruta encontrada.

- Para rutas sin puntos intermedios o con un solo punto intermedio, el algoritmo de Dijkstra es el idóneo, este algoritmo arroja la solución óptima en un tiempo de cómputo muy reducido para este tipo de rutas.

- El algoritmo MMAS ejecutado sobre la matriz original del grafo arroja unos resultados nada aceptables. Esto es debido a la gran cantidad de hormigas que no logran encontrar una ruta. Al encontrar pocas rutas la heurística no cumple su función y la calidad de las mismas es muy pobre.

Cabe destacar que con las conclusiones presentadas en este proyecto, se ha implementado la función de búsqueda de rutas intermedias en dos aplicaciones reales, en una aplicación web y en una aplicación móvil. La implementación que se ha realizado es la de utilizar el algoritmo de Dijkstra cuando el calculo de la ruta no tiene puntos intermedios o solamente tiene un punto intermedio. La solución híbrida con Dijkstra y ramificación y acotamiento para rutas que contengan entre dos y catorce puntos intermedios, y Dijkstra más el algoritmo ACO para cuando la ruta tenga más de catorce puntos intermedios.

Este proyecto deja varias líneas futuras de estudio para mejorar las rutas encontradas y el tiempo de cómputo empleado en las mismas. Estas líneas futuras son las que se detallan a continuación.

- El algoritmo de ramificación y acotamiento utiliza una heurística para la poda de las ramas. Existen varias heurísticas para realizar esta función, pero en este proyecto solo se ha probado una. Sería interesante comparar este algoritmo utilizando diferentes heurísticas para la poda de las ramas.

- El algoritmo de ramificación y acotamiento es apto para ser paralelizable. El proceso de búsqueda de los nodos en anchura del árbol se podría paralelizar en varios hilos para, de este modo, reducir considerablemente el tiempo de cómputo.

- El algoritmo MMAStambién es apto para ser paralelizado. En cada iteración se suelta un número determinado de hormigas, las cuales podrían buscar cada una de ellas la ruta en un hilo diferente y después juntar los resultados obtenidos al final de la iteración. Esta mejora también reduciría el tiempo de cómputo.

Bibliografía

Cirre, F. J. (2004). *Matemática discreta*. Anaya.

Cook, W. J. (2012). *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press. Consultado el 26 de febrero de 2023, desde http://www.jstor.org/stable/j.ctt7t8kc

Deneubourg, J. .-., Aron, S., Goss, S., & Pasteels, J. M. (1990). The self-organizing exploratory pattern of the argentine ant. *Journal of insect behavior*, *3*, 159-168.

Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. *IEEE computational intelligence magazine*, *1*(4), 28-39.

Dorigo, M., Maniezzo, V., & Colorni, A. (1991). The ant system: An autocatalytic optimizing process.

Fuentes Penna, A. (2014). Problema del agente viajero. *XIKUA Boletín Científico de la Escuela Superior de Tlahuelilpan*, *2*(3). https://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n3/e5.html

García Merayo, F. (2005). *Matemática discreta* (2a ed.). Thomson.

Gendreau, M., Potvin, J.-Y., et al. (2010). *Handbook of metaheuristics* (Vol. 2). Springer.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence [Applications of Integer Programming]. *Computers & Operations Research*, *13*(5), 533-549. https://doi.org/https://doi.org/10.1016/0305-0548(86)90048-1

Goss, S., Aron, S., Deneubourg, J.-L., & Pasteels, J. M. (1989). Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, *76*(12), 579-581.

Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

Jordán Lluch, C., Seoane Sepúlveda, J. B., & Murillo Arcila, M. (2022). *Teoría de grafos y modelización*. Paraninfo.

Lee, R. C. T., Tseng, S. S., & Chang, R. C. (2014). *Introducción al diseño y análisis de algoritmos : un enfoque estratégico.* McGraw-Hill Interamericana.

Pérez Aguila, R. (2012). Una introducción a las matemáticas para el análisis y diseño de algoritmos. https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=862536

Stützle, T., & Hoos, H. H. (1996). Improving the Ant System: A detailed report on the MAX–MIN Ant System. *FG Intellektik, FB Informatik, TU Darmstadt, Germany, Tech. Rep. AIDA–96–12*.

Veerarajan, T., Nagore Cazarez, G., Valeriano Assem, J., Ríos Flores, A., Miranda García, A., & Veerarajan, T. (2008). *Matemáticas discretas con teoría de gráficas y combinatoria*. McGraw-Hill Interamericana.

Vidal, A. (2013). Algoritmos Heurísticos en optimización. *Universidad de Santiago de Compostela, Facultad de Matemáticas Máster en Técnicas Estadísticas*.