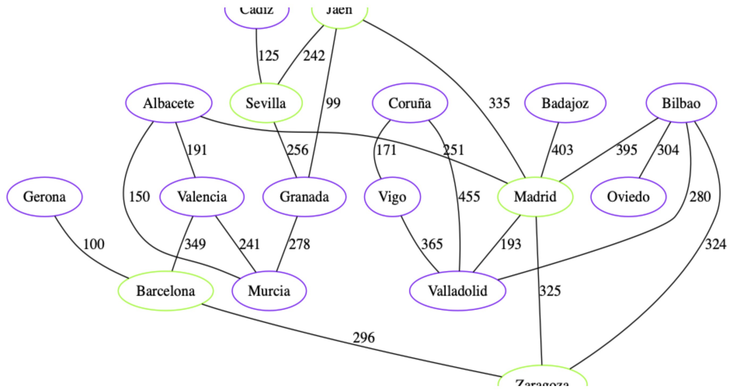
1. INTELIGENCIA ARTIFICIAL

ÍNDICE

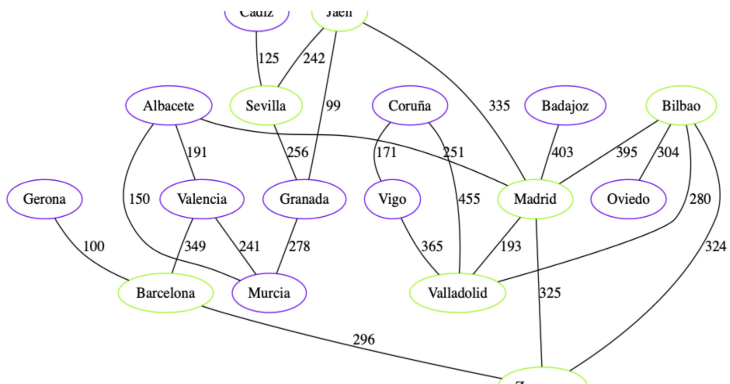
ALGORITMOS IMPLEMENTADOS

Se han realizado los algoritmos de **búsqueda en anchura**, **búsqueda en profundidad** y **búsqueda optimal o de primero mejor**.

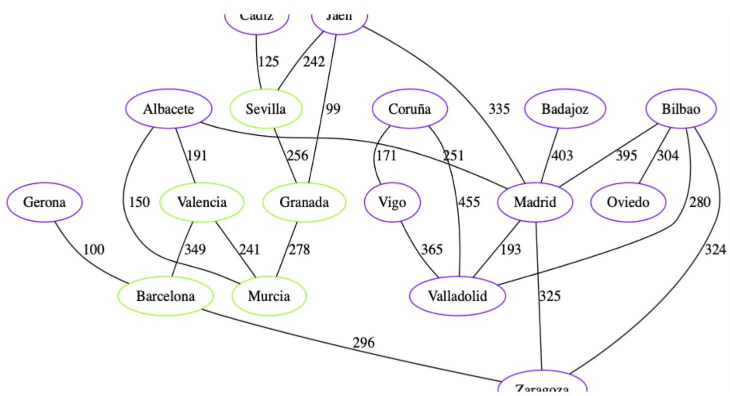


Se adjunta el camino obtenido para la ejecución de cada uno de los algoritmos sobre los datos de ciudades presentes en la documentación de la práctica.

En los resultados presentados se puede ver a la perfección las características de cada uno de los algoritmos.



Búsqueda en anchura garantiza que si hay solución la proporcionará siendo esta de la menor longitud de camino posible, aunque puede que no sea el camino óptimo desde el origen al objetivo. Se trata a la lista de abiertos como una cola.



Búsqueda en profundidad, ya que utilizamos una lista de cerrados siempre encontrará un camino si lo hubiera, aunque este no será óptimo. Se trata a la lista de abiertos como una pila.

Búsqueda optimal o de primero mejor garantiza que siempre se encontrará camino y que este será de coste mínimo.

```
> (pintaBusqueda '("Sevilla" "Barcelona") ciudades "anchura")
'("Sevilla" "Jaén" "Madrid" "Zaragoza" "Barcelona" 1198)
> (pintaBusqueda '("Sevilla" "Barcelona") ciudades "profundidad")
'("Sevilla" "Jaén" "Madrid" "Valladolid" "Bilbao" "Zaragoza" "Barcelona" 1670)
> (pintaBusqueda '("Sevilla" "Barcelona") ciudades "primero")
'("Sevilla" "Granada" "Murcia" "Valencia" "Barcelona" 1124)
```

ESTRUCTURA DEL CÓDIGO

Tras analizar los algoritmos de búsqueda comprobamos que la mayor parte de las tereas que realizan son comunes a todos ellos. Teniendo en cuenta solo los tres algoritmos implementados sus diferencias pueden reducirse únicamente a la forma en la que la lista de siguientes es insertada en la lista de abiertos.

Teniendo esto en cuenta se ha creado un **algoritmo genérico** cuyas tareas son:

- Obtener el elemento actual, es decir, el primer elemento de la lista de abiertos.
- Comprobar si el elemento actual es el elemento objetivo, si lo es habremos terminado.
- Obtener la lista de siguientes que son todos aquellos nodos a los que podemos ir desde el actual, pero a los que no hayamos ido aún pues tenemos una lista de cerrados.
- Eliminar el elemento actual de la lista de abiertos.
- Insertar el elemento actual en la lista de cerrados para no volverlo a visitar.

Para convertir este algoritmo en un algoritmo de búsqueda solo nos quedaría insertar la lista de sucesores en la lista de abiertos. Para realizar esto el algoritmo tomará como parámetro una función lambda que realizará esta tarea. Dicha función insertará los siguientes al inicio de los abiertos para la **búsqueda en profundidad**, al final de los abiertos para la **búsqueda en anchura** o de forma ordenada para **la búsqueda optimal o de primero mejor**.

El resto de la implementación realizada consiste en trabajar con los elementos dentro de las listas según los formatos que hemos elegido para ello.

Cabe destacar que, para poder saber el camino recorrido, así como su coste los elementos que contiene la lista de abiertos están compuestos por una lista de todas las ciudades visitadas estando la primera a la izquierda y la última a la derecha. Es por ello por lo que cuando el algoritmo finaliza al haber llegado al destino solo es necesario retornar el elemento actual para tener toda la información sobre el camino recorrido.

En caso de que la lista de abiertos esté vacía podremos concluir que no es posible realizar la búsqueda con éxito en cuyo caso retornaremos una lista vacía.

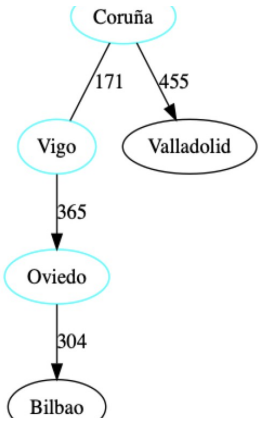
Cabe destacar que siempre que se ha creído conveniente la declaración de funciones se ha anidado para ocultar su visibilidad desde otras partes de código y así aumentar el orden en el mismo.

FORMATO DE LOS DATOS DE ENTRADA

Se ha elegido el formato pensando tanto en la facilidad para crear los algoritmos de búsqueda como en la sencillez a la hora de crearlos.

Los datos serán almacenados en una lista de listas teniendo cada una de las listas interiores el siguiente formato:

‘(“nombre_ciudad” (“ciudad_a_la_que_podemos_ir1” coste1) (“ciudad_a_la_que_podemos_ir2” coste2))



Un ejemplo completo de una posible entrada sería:

'(("Coruña" ("Vigo" 171) ("Valladolid" 455))
("Vigo" ("Coruña" 171) ("Oviedo" 365))
("Oviedo" ("Bilbao" 304)))

Para poder ejecutar los algoritmos de búsqueda se utilizarán las siguientes funciones:

Para ejecutar el algoritmo con interfaz gráfica *(**pintaBusqueda** objetivo ciudades tipo_busqueda)*

Para ejecutar el algoritmo sin interfaz gráfica *(**inicioBusqueda** objetivo ciudades tipo_busqueda)*

- objetivo es una tupla de la forma ‘(“ciudad_inicio” “ciudad_fin”)
- ciudades es una lista de ciudades en el formato explicado con anterioridad.
- tipo_busqueda es una cadena de caracteres: [“anchura” | “profundidad” | “primero”]

Un ejemplo de ejecución podría ser:

(pintaBusqueda '("Oviedo" "Bilbao") ciudades “profundidad”)

(inicioBusqueda '("Santader" "Sevilla") ciudades “anchura”)

LECTURA DE FICHEROS

Adicionalmente se podrá proporcionar un nombre de fichero para que el algoritmo toma su entrada de él. Para hacerlo solo es necesario escribir entre comillas el nombre del fichero almacenado en la raíz del proyecto en el parámetro ciudades de la función.

El formato de los datos en dicho fichero será el del ejemplo publicado en la Blackboard.

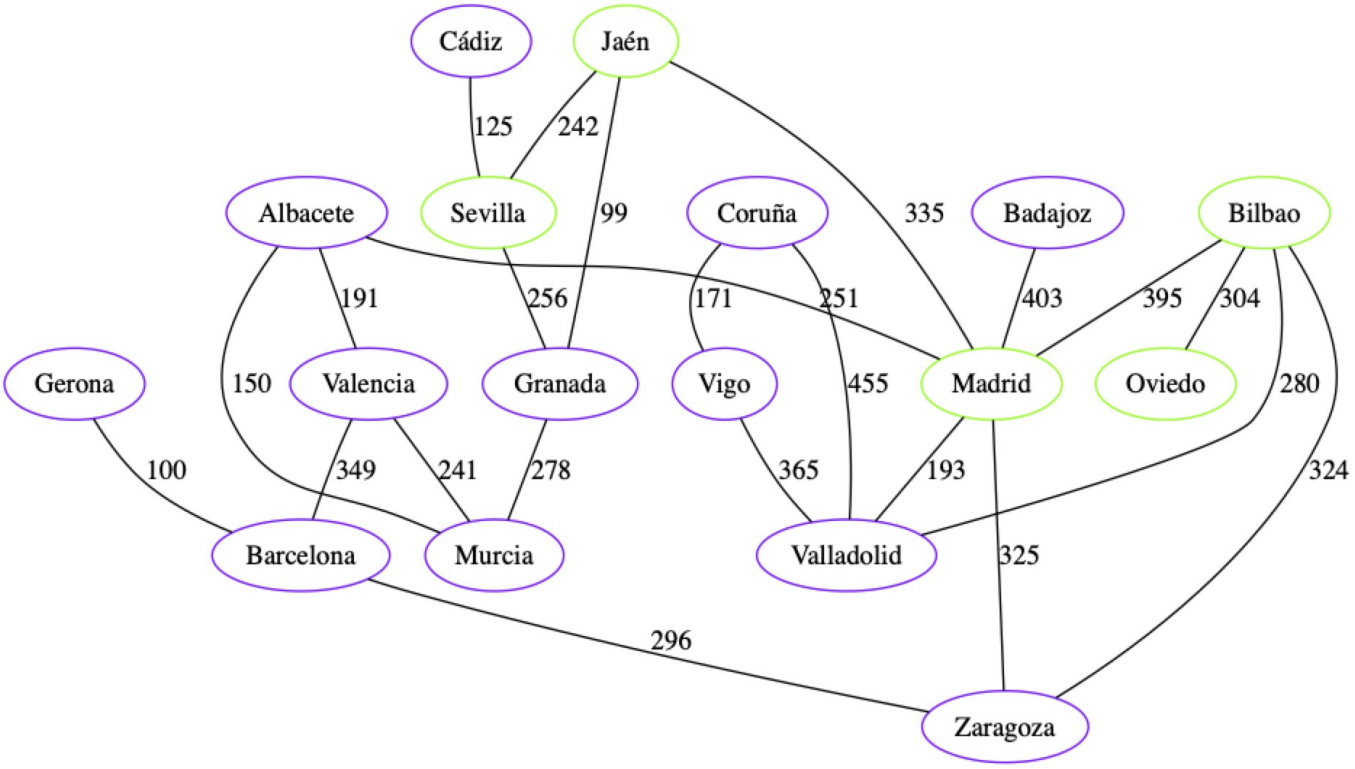
MEJORAS REALIZADAS

Se he realizado una visualización gráfica del grafo que los datos de entrada representan, así como del camino proporcionado por la búsqueda sobre dicho grafo.

Los nodos que forman parte del camino son pintados de color verde claro mientras que los que no forman parte de él se pintan en azul oscuro.

Para poder realizar esto se utilizan dos paquetes de la librería estándar de racket: /graph y /racket/gui/base. Del primero utilizaremos la parte correspondiente a la creación de archivos .dot que representen al grafo, mientras que del segundo lo usaremos para crear las ventanas en las que mostrar la imagen resultado de procesar el archivo .dot.

Para poderlo realizar primero se deben transformar los datos de entrada de cómo se utilizan en el algoritmo para poder realizar la búsqueda al formato que la librería /graph necesita. Adicionalmente será necesario crear una tabla hash que indique el color en el que cada nodo debe pintarse.



ARCHIVOS ENTREGADOS

main.rkt → Contine las funciones con las que iniciar la búsqueda, así como la función de búsqueda en sí misma.
busquedaGrafos.rkt → funciones genéricas de manipulación de listas que son utilizadas por la parte general del algoritmo.
busquedaProfundidad.rkt → función que inserta los siguientes al inicio de actuales.
busquedaPrimero.rkt → función que inserta los siguientes de forma ordenada en actuales.
busquedaAnchura.rkt → función que inserta los siguientes al final de actuales.
grafo.rkt → Contine dos grafos de ejemplo ciudades se corresponde con el proporcionado en la documentación de la práctica y ciudadesExtra es una variante del mimso.
gra.rkt → Contine las funciones que permiten dibujar el grafo.
manejadorArchivos.rky → Contiene las funciones necesarias para leer archivos de texto con el formato indicado en la Blackboard.

Para poderse ejecutar todos los archivos deberán estar en la misma carpeta, también el archivo del que los datos de entrada se vayan a tomar de utilizar esta funcionalidad.
Las dependencias externas del proyecto consisten en las funciones utilizadas para dibujar el grafo que pertenecen al paquete **/graph** y el programa **gravphviz** que es el encargado de procesar los ficheros .dot para generar las imágenes que se puestaran.

Documentación de graph → https://docs.racket-lang.org/graph/index.html
Documentación de graphviz → https://www.graphviz.org