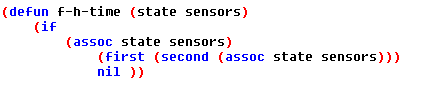
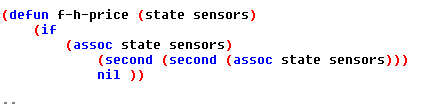
***Inteligencia Artificial: Práctica 2***

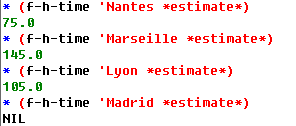
***Ejercicio 1:***

Para el ejercicio, hemos desarrollado dos funciones que calculan los valores de las dos posibles heurísticas:

1. **F-h-time**: Si en la lista de heurísticas está el estado pedido, devolvemos el primer elemento de la segunda sublista (la que contiene los valores), que es el que corresponde al tiempo.
2. **F-h-price**: Si en la lista de heurísticas está el estado pedido, devolvemos el segundo elemento de la segunda sublista (la que contiene los valores), que es el que corresponde al coste.

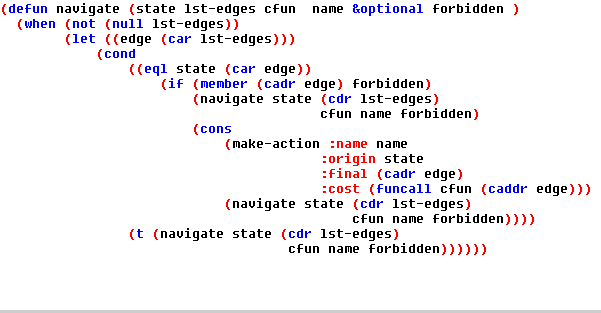


Ejecutando los casos de prueba, obtenemos lo siguiente:



***Ejercicio 2:***

En este segundo ejercicio, se nos pide hacer una función que devuelva las posibles acciones desde un nodo, dado el estado (nombre del nodo). Para ello, hemos desarrollado una función general, y seguidamente una serie de funciones secundarias que, depende de lo pedido, llaman a esta primera con distintos argumentos. Las funciones son las siguientes:

1. **Navigate**: Función general, que recibe como argumentos un estado, la lista de conexiones del grafo, la función para acceder al valor buscado (en el caso del ejemplo, tiempo o precio), el nombre a dar a las acciones generadas y opcionalmente, una lista de ciudades a las que no se puede llegar en tren. Esta función recorre los elementos de la lista de conexiones, comprobando si el origen de cada conexión es el estado pasado como argumento. En tal caso, si el destino no está en la lista de prohibidos, crea una acción con los campos correspondientes, y sigue recorriendo las conexiones. En cualquier otro caso, ignora esa conexión y sigue recorriendo.
2. **Navigate-canal-time:** Obtiene las posibles acciones por canales y según el tiempo desde un estado dado. Usa Navigate.



1. **Navigate-canal-price**: Obtiene las posibles acciones por canales y según el precio desde un estado dado. Usa Navigate.

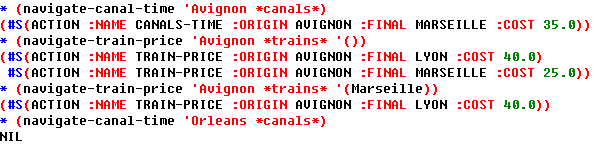


1. **Navigate-train-time:** Obtiene las posibles acciones en tren y según el tiempo desde un estado dado. Usa Navigate.



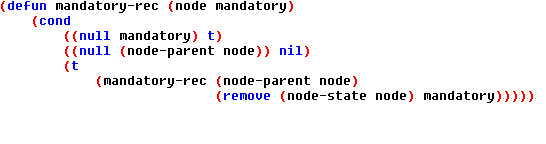
1. **Navigate-train-price:** Obtiene las posibles acciones en tren y según el precio desde un estado dado. Usa Navigate.

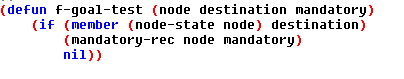


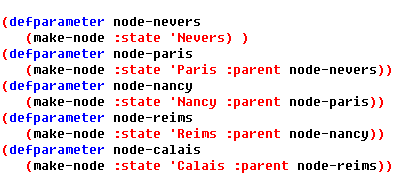
Ejecutando los casos de prueba, obtenemos:

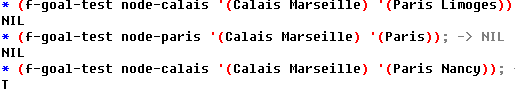
***Ejercicio 3:***

Se nos pide desarrollar una función que compruebe si se ha alcanzado el objetivo. Para ello, hay que comprobar que el estado del nodo dado es un estado final, y que ha pasado por todas las ciudades obligatorias. Para lo segundo, hemos creado una función auxiliar. En total tenemos estas dos funciones:

1. **Mandatory-rec:** Va recorriendo los antecesores del nodo dado, comprobando en cada iteración si la ciudad visitada está en la lista de obligatorias y, en tal caso, eliminándolas de esta. La función devolverá si se han visitado todas o si se ha llegado al final y no ha pasado por todas.
2. **F-goal-test:** Si el estado del nodo es un estado final, devuelve lo que devuelva Mandatory-rec, en otro caso devuelve nil.

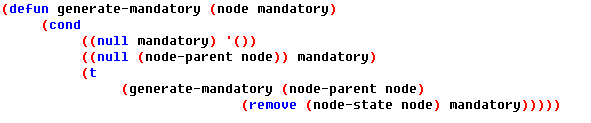


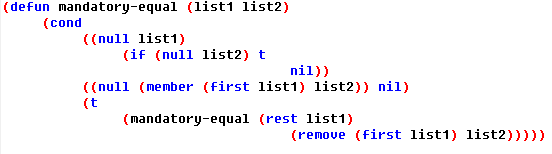
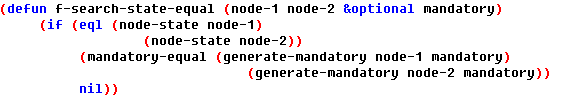
Ejecutando los casos de prueba, obtenemos:



***Ejercicio 4:***

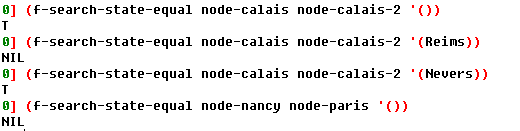
En este ejercicio, se nos pide codificar una función que compare dos nodos , comprobando si son iguales a nivel estados de búsqueda. Para ello, hemos desarrollado dos funciones auxiliares, utilizadas por una tercera general que da lugar al resultado buscado. Las funciones son:

1. **Generate-mandatory:** Recorre todos los antecesores del nodo dado, comprobando si están en la lista de obligatorios, y en tal caso, borrándolos de la misma. Al terminar, la función devolverá las ciudades obligatorias que le quedan al nodo por visitar.

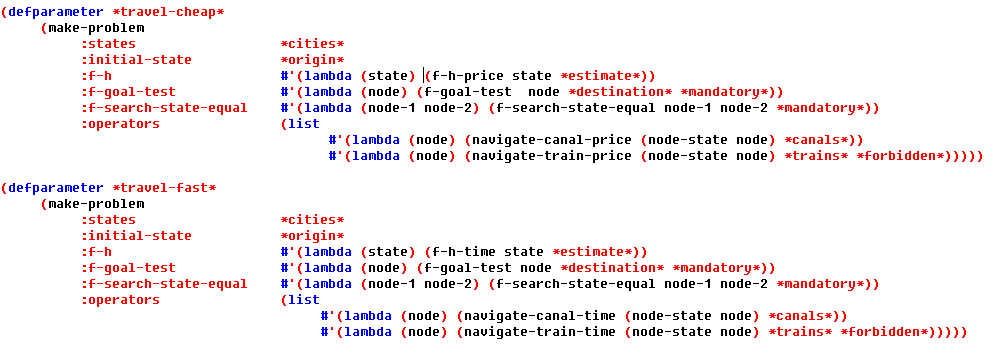
1. **Mandatory-equal:** Recibe dos listas y las compara, recorriendo la primera elemento a elemento y comprobando si cada uno está en la otra.
2. **F-search-state-equal:** Función general que, usando las dos anteriores, compara si dos nodos son iguales. Para ello, comprueba primero que sus estados sean iguales, y seguidamente genera la lista de estados obligatorios aun por recorrer de cada nodo, y las compara. En caso de ser iguales, ambos nodos son idénticos a nivel de estado de búsqueda.

Ejecutando los casos de prueba, obtenemos:





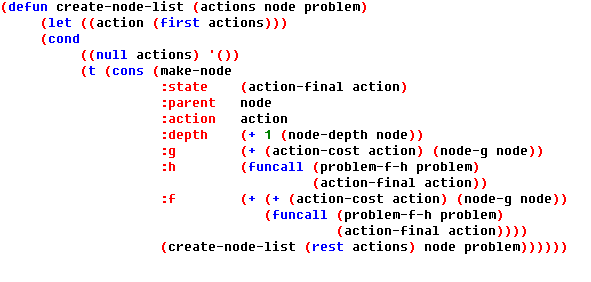
***Ejercicio 5:***

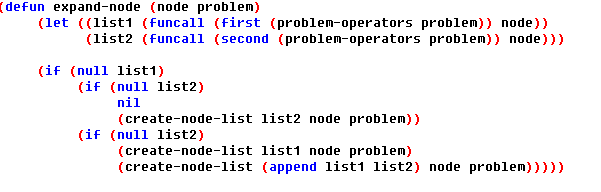
Para este ejercicio hemos creado dos estructuras de tipo problema, una para el que optimiza el precio y otra para el que optimiza el tiempo. Por lo tanto, los campos los hemos rellenado con las funciones correspondientes, las que obtienen las acciones y las heurísticas correspondientes a cada problema, y las funciones de comparación y de estado final ,que coinciden junto al origen y al conjunto de estados en ambos problemas.

***Ejercicio 6:***

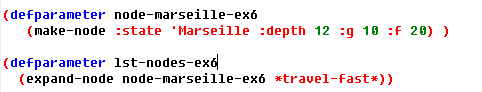
En este ejercicio, se nos pide desarrollar la función de expansión de un nodo. Para ello, hemos codificado una función auxiliar que genera a partir de una lista de acciones, los nodos que surgen. En total, tenemos dos funciones:

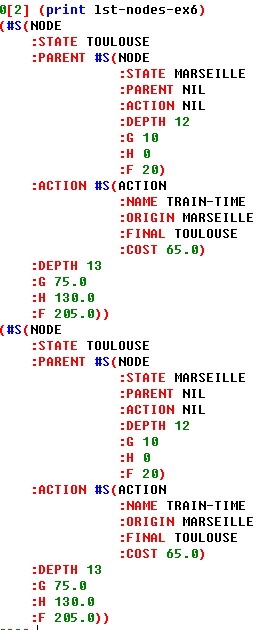
1. **Create-node-list:** Recibiendo un nodo y una serie de acciones como parámetro, genera los nodos que tienen a este primero como padre y que surgen a partir de las acciones dadas.



1. **Expand-node:** Primeramente, se generan las listas de acciones a raíz de los operadores del problema. Estas listas se le pasan a create-node-list, y esta genera la lista de nodos. Lo que realmente hace esta función es comprobar si alguna de las listas generadas por los operadores es nil, y en tal caso le pasa a la función auxiliar solo la que corresponda, o un cons de ambas.

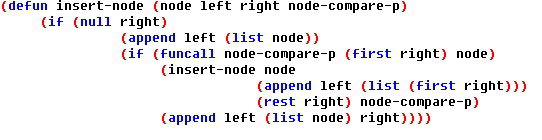
Ejecutando los casos de prueba, obtenemos:

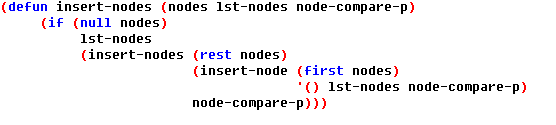




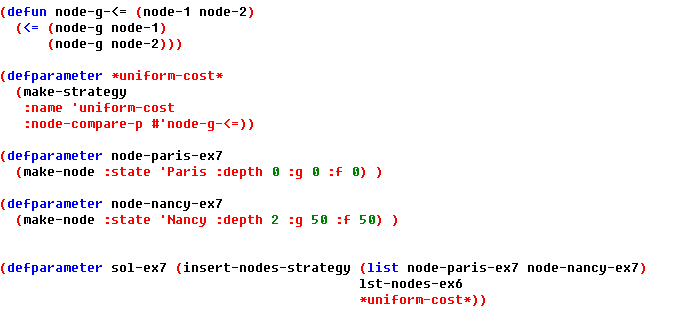
***Ejercicio 7:***

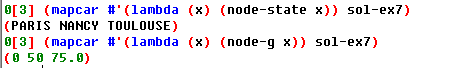
En este caso, hemos decidido dividir el trabajo en tres funciones, a cada cual más general, y siendo el objetivo final introducir una serie de nodos en orden en una lista ordenada de nodos. Las funciones son:

1. **Insert-node:** Recorre la lista, comparando el nodo a insertar usando la función dada, y determinando que nodos van antes (izquierda) y cuales después (derecha) de este mismo, situándolo en su lugar.
2. **Insert-nodes:** Va insertando los nodos de una lista no vacía en la lista ordenada, llamando recursivamente a insert-node.



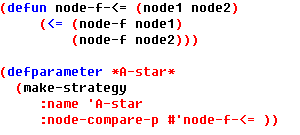
1. **Insert-nodes-strategy:** Llama a insert-nodes, extrayendo la función de comparación de la estrategia dada.

Ejecutando los casos de prueba, obtenemos:



***Ejercicio 8:***

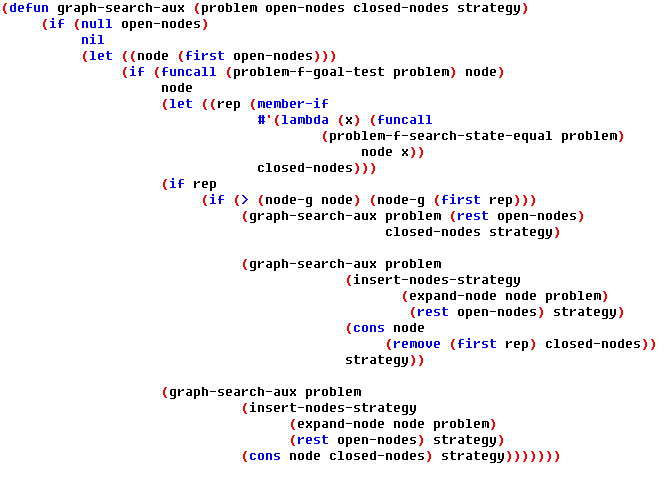
En este ejercicio, simplemente definimos la función de comparación de la f de dos nodos, y la incluimos en la definición de la estrategia A estrella, la cual declaramos como parámetro.

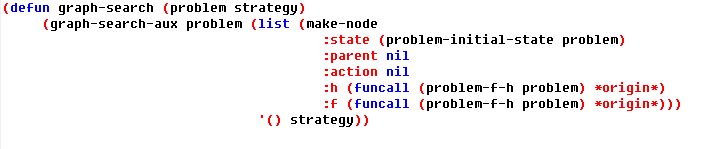


***Ejercicio 9:***

Se nos pide desarrollar la función de búsqueda en grafo, que aplicará la estrateia pasada por argumento. Para ello, hemos codificado tres funciones:

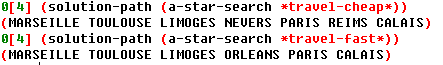
1. **Graph-search-aux:** Aplica el algoritmo de búsqueda según la estrategia dada, siguiendo el pseudocódigo del enunciado.



1. **Graph-search:** Simplemente llama a graph-search-aux, inicializando la lista de open nodes con el nodo origen del problema.
2. **A-star-search:** Llama a graph-search, pasándole como estrategia A-star.



Ejecutamos el caso de prueba, haciendo uso de la función del ejercicio 10, solution-path, para evitar una solución muy extensa



***Ejercicio 10:***

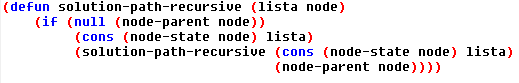
El objetivo de este apartado es mostrar por pantalla el camino que ha seguido el algoritmo de búsqueda. Para ello hemos implementado dos funciones:

1. **Solution-path :** Esta función recibe un nodo como parámetro, si éste es null simplemente devuelve *nil*. En otro caso llama a la función *solution-path-recursive* pasándole una lista vacía donde iremos metiendo los nodos visitados y el nodo.



1. **Solution-path-recursive :** Esta función recibe como parámetros de entrada una lista donde iremos acumulando los nodos visitados y el nodo por el que empezamos a buscar.

Si el padre del nodo actual es *nil* significa que hemos llegado al principio del camino, por lo que simplemente devolvemos la lista de nodos que hemos ido haciendo, añadiendo el actual. En otro caso, devolvemos la función recursivamente con una nueva lista en la que metemos el nodo actual y como nodo el padre, hasta llegar al nodo raíz.



Ejecutando los casos de ejemplo nos queda lo siguiente:



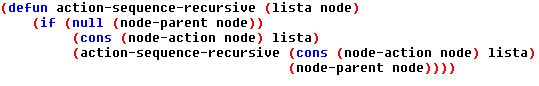


A continuación, hemos implementado otras dos funciones para imprimir las acciones que realizan los algoritmos de búsqueda:

1. **Action-sequence :** Básicamente funciona igual que *solution-path* pues llama a otra función recursiva para calcular la lista.

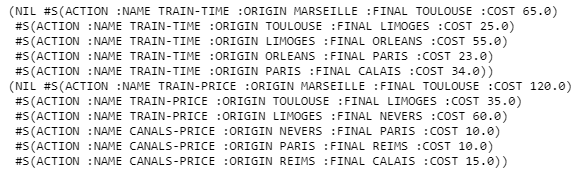


1. **Action-sequence-recursive :** Tiene el mismo mecanismo que *solution-path-recursive* pues realiza recursivamente una búsqueda de las acciones actual, añadiéndolas en una lista, de hijo a padre, hasta llegar al nodo raíz.Es decir, es igual que la función anterior pero cambiando *node-state* por *node-action.*



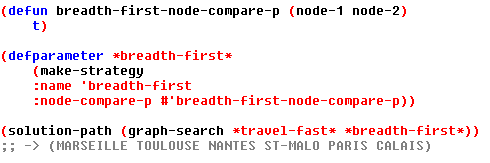
Ejecutando los casos de prueba nos queda lo siguiente:





***Ejercicio 11:***

En primer lugar, la **búsqueda en anchura** la hemos implementado añadiendo una función que al comparar dos nodos devuelve siempre *true*. Como búsqueda en anchura es de tipo FIFO (first in first out), al implementar así la función le estamos indicando que queremos que los nodos descubiertos se pongan al final de la cola, para explorar primero los que están al principio. Podemos observar el resultado de la búsqueda en anchura del problema en la imagen. En éste podemos observar que hay ciudades repetidas, sin embargo, no son ciudades repetidas pues tienen distintos estados ya que hasta llegar a París no se había pasado por la ciudad obligatoria (París), una vez ha pasado son estados distintos.

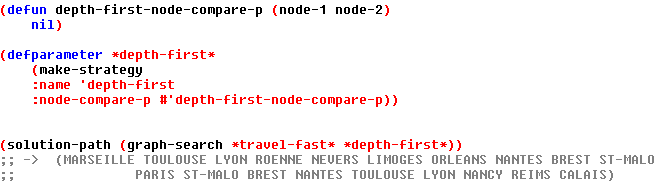




Otra forma de implementarlo sería comparar las profundidades de los nodos como menor igual y saldría el mismo resultado:

*(node-breadth node-1) <= (node-breadth node-2)*

Por otro lado, el algoritmo de *búsqueda en profundidad* es “contrario” al de búsqueda en anchura, es decir, este algoritmo es tipo LIFO (last in first out) de modo que la función de comparación de nodos devuelve siempre *nil* para que añada los nodos al principio de la cola, explorando primero los que acabamos de descubrir. El resultado de la búsqueda se encuentra en la imagen.



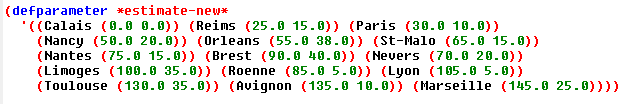


También podríamos implementarlo del siguiente modo, dándonos el mismo resultado:

*(node-depht node-1) >= (node-depth node-2)*

***Ejercicio 12:***

Hemos elegido y definido la siguiente heurística:



Básicamente consiste en elegir el coste más bajo de los nodos salientes respecto a nuestra posición, contando tanto canales como trenes, por ejemplo, desde Marseille podemos llegar a Avignon con un coste de 25, y a Tolouse con un coste de 120, luego elegimos Avignon porque es el coste de salida más barato. Además, Calais tiene heurística 0 pues es el final del camino.

Se puede observar a simple vista que esta heurística es monótona. Sean n, n’ dos nodos de modo que n’ es sucesor de n, con h(n) la heurística de n, h(n’) la heurística de n’ y c(n, n’) el coste de ir de n a n’. Entonces una heurística monótona cumple lo siguiente:

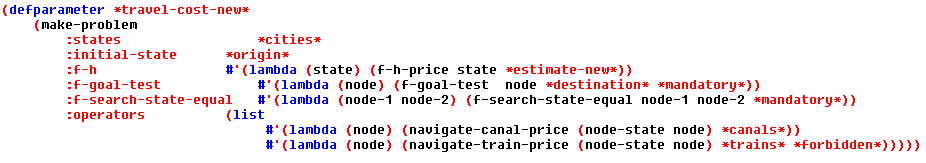


La heurística que hemos implementado en la imagen de arriba es monótona. Se podría demostrar haciendo una tabla con todos los nodos y comprobando que la condición se cumple. Sin embargo, el coste de ir a una ciudad es mucho mayor que la heurística de la propia ciudad, es decir, vayamos a donde vayamos, la heurística de nuestra ciudad siempre va a ser inferior a la suma del coste de movernos a una ciudad, más la heurística de la ciudad a la que nos movemos. Siempre se cumple que:

*c(n, A, n’) > h(n)*

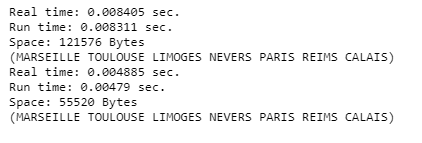
Luego podemos concluir que la heurística es monótona, ya que solamente el coste de movernos a una ciudad distinta es mayor que la heurística de la ciudad.

Finalmente, podemos observar en las siguientes imágenes que, además de obtener el mismo resultado con *\*estimate\** que con la heurística nueva *\*estimate-new\**, obtenemos mejor tiempo. Antes obteníamos un tiempo de 0.008405 segundos mientras que ahora obtenemos más o menos la mitad, 0.00479, lo cual es una gran diferencia en problemas mayores.









***ANEXO: PREGUNTAS***

1. ***¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?***

Se ha realizado este diseño simplemente para facilitar el problema. Básicamente hemos dividido un problema largo y complejo es distintas funciones, resolviendo poco a poco cada paso, de modo que hemos convertido un problema complejo en varios módulos que lo construyen.

De esta forma, es mucho más visual y más sencillo de entender, además de poder abstraerlo a cualquier tipo de problema y de algoritmo de búsqueda

1. ***¿Qué ventajas aporta?***

La ventaja principal es que esta implementación del problema es general, la podemos asignar a cualquier algoritmo de búsqueda y a cualquier problema, sólo debemos crear una estructura que defina el algoritmo y el problema que necesitamos y pasarlo como parámetro de entrada. De esta manera hemos podido implementar, a parte del algoritmo a estrella, los algoritmos de búsqueda en anchura y en profundidad, pues sólo ha hecho falta definir una función para comparar dos nodos y definir una estructura para el algoritmo.

Además, implementando un problema nuevo podríamos reutilizar las mismas funciones para la búsqueda de su solución, solamente haría falta crear las estructuras y las definiciones del problema.

***¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?***

Las funciones lambda solamente dependen de la etiqueta o estado del nodo que es pongamos de modo que queda un problema más abstracto y abierto a implementaciones y problemas distintos, haciendo más fácil el resolver problemas similares en los que necesitemos estos algoritmos.

***3. Sabiendo que en cada nodo de búsqueda hay un campo “parent”, que proporciona una referencia al nodo a partir del cual se ha generado el actual ¿es eficiente el uso de memoria?***

Sí porque en vez de guardar en memoria cada padre de cada nodo, solamente guardamos una referencia o un puntero al nodo padre, de modo que cada hijo tiene una referencia a su padre, y cada padre tiene una referencia a su ancestro hasta llegar al nodo raíz. Básicamente es una lista enlazada de nodos, donde en vez de guardar cada iteración en memoria, guardamos una referencia a la memoria donde se ubique, haciendo buen uso de la memoria.

***4. ¿Cuál es la complejidad espacial del algoritmo implementado?***

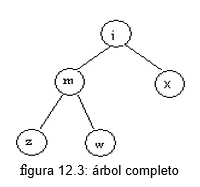
La complejidad del algoritmo no solo depende de la estrategia de búsqueda que utilicemos, sino también de la heurística implementada. A mejor heurística, mejor resultado obtenemos y menor coste requiere.

Por ejemplo, la búsqueda A estrella, si utiliza una heurística óptima, tiene un factor de ramificación *f*, es decir, en el camino óptimo cada nodo tiene *f* sucesores,y el nodo objetivo se encuentra a una profundidad *p*, nos queda la siguiente complejidad espacial lineal:

*f ( p – 1) + 1 = O(f\*p)*

Sin embargo, utilizando por ejemplo, el algoritmo de búsqueda en anchura, como exploramos cada nodo que descubrimos, nos quedaría la siguiente complejidad espacial:

*1 + p +*



Por ejemplo, la figura muestra un árbol de profundidad 2 y de factor de ramificación 2, para ir de *i* a *z* por ejemplo, tendría una complejidad igual a 3 aplicando la fórmula de A estrella, mientras que búsqueda en anchura tendría una complejidad exponencial mucho más grande.

***5. ¿Cuál es la complejidad temporal del algoritmo?***

La complejidad temporal es exactamente la misma que la exponencial, puesto que explorar más o menos nodos implica más o menos coste temporal. Es decir, una estrategia como A estrella tendría, con una heurística óptima, una complejidad temporal lineal (la fórmula del apartado anterior), mientras que un algoritmo que no utilice heurísticas, o que tenga una complejidad espacial mayor, tendrá mayor complejidad temporal, por ejemplo, búsqueda en anchura, también visto en el apartado anterior, también tendría un coste exponencial.

***6. Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción “navegar por agujeros de gusano” (bidireccionales).***

Supongamos que, llegado a un estado, se produce un bucle infinito entre dos nodos bidireccionales con costes óptimos, sin embargo, este camino no nos lleva a ninguna parte.

En primer lugar, podríamos añadir a la estructura del nodo un contador de saltos bidireccionales, de modo que aumente cada vez que realice un salto bidireccional. Otra opción podría ser añadir una lista con los saltos realizados, de modo que si detectamos que se repiten ciertos saltos, parar y continuar por el siguiente camino.

Por otro lado, podríamos añadir un número máximo de saltos a la función *navigate* de modo que no pueda dar más de esos saltos bidireccionales y cambie de sentido para no entrar en un bucle infinito.