Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Proyecto de Prolog

Buscador de rutas aéreas

Inteligencia artificial

Grado en Ingeniería informática

Jose Ramón Morera Campos

(alu0101471846@ull.edu.es)

ÍNDICE

1. Introducción

1.1 Definición del problema

El proyecto consiste en un sistema para la búsqueda de rutas aéreas entre dos aeropuertos dados, estudiando vuelos directos y conexiones. Se considerarán las horas de salida y llegada de los vuelos, de forma que solo se planteen conexiones posibles. Se buscará minimizar la duración temporal del viaje, siendo esta la suma de las duraciones de los vuelos tomados.

Se mostrará como resultado los vuelos que se deben tomar para llegar del aeropuerto inicial al aeropuerto destino con el menor tiempo de vuelo posible.

1.2 Dominio de aplicación

El problema presentado se enmarca dentro del ámbito de la logística, concretamente la búsqueda de rutas y la optimización.

1.3 Trabajos similares

Se han encontrado multitud de proyectos con características similares desarrollados usando Prolog:

* 'Logic flight' https://github.com/rckmath/logic-flight
* 'Flight-Rec-Pl' https://github.com/emNakamoto/Flight-Rec-Pl
* 'Airline System' https://github.com/kashaniarya/Airline-Reservation-System
* 'Connecting Flights' https://github.com/Rosyparadise/Connecting-flights
* 'flightpaths-prolog' https://github.com/srijithas/flightpaths-prolog

2. Proceso de realización

2.1 Obtención de los datos

En primer lugar, realicé la búsqueda de los datos con los que cuenta la base de conocimiento, de cara a determinar qué podría hacer y qué no el programa según las características de dichos datos.

Finalmente decidí usar dos bases de conocimiento, que en el proyecto se ubican en la carpeta 'data', son 'airports.pl' y 'flights.pl'. La primera contiene información sobre 90 aeropuertos de Canarias, de España y de algunos otros países. Dicha información la obtuve de https://openflights.org/data.php, como me sugirió el compañero Samuel Lorenzo en el foro de la asignatura.

Sobre los datos obtenidos de la citada fue necesario realizar un tratamiento para descartar atributos que no eran de interés para el proyecto y para formatearlos estilo Prolog, de forma que cada fila fuera un predicado. Para ello hice uso de expresiones regulares.

La segunda base, 'flights.pl' cuenta con un listado de vuelos entre los aeropuertos de la primera base. Para generarla hice uso de una herramienta de inteligencia artificial a la que le facilité el listado de aeropuertos y el formato deseado de los predicados de los vuelos, obteniendo 250 vuelos.

Cabe destacar que en ningún momento se ha buscado el realismo de los datos, sino una cierta representatividad para conformar una base de conocimiento 'de juguete' que permita probar el programa en entornos pseudo-realistas.

Para simplificar el problema trabajado se ha considerado que todos los vuelos tienen lugar todos los días con el mismo horario.

Para la carga de los datos se ha empleado el predicado built-in de Prolog 'consult', indicando fla ruta absoluta de los ficheros que se deben escanear para cargar las bases de conocimiento. Dichos predicados se han declarado como directivas del fichero main.pl, mediante el operador ':-', de forma que son procesados al cargar el archivo.

```
% Importamos datos
:- consult('/home/jr/Desktop/IA/EntregaP3/data/flights.pl').
:- consult('/home/jr/Desktop/IA/EntregaP3/data/airports.pl').
```

2.2 Algoritmo A*

El problema tratado es la búsqueda de rutas de mínima duración entre aeropuertos. Para poder realizar una búsqueda eficiente, he optado por la implementación del algoritmo A*.

He elegido como criterio a minimizar el tiempo de vuelo, ya que aunque a priori estudié factorizar también el coste, concluí que dicho factor no es adecuado para búsquedas eficientes, ya que los costes de los vuelos se determinan mediante diferentes casuísticas no deterministas e incluso fluctúan en tiempo real, además de poder variar entre aerolíneas, horas etc.

Por tanto, en este trabajo me he ceñido a minimizar la duración total de vuelos, es decir, el coste acumulado es la suma de la duración de cada vuelo tomado.

Para la implementación del algoritmo A* tomé como referencia el repositorio https://github.com/ciaranfinn/a-star. No obstante, he realizado una

implementación propia con cambios significativos, ya que el contenido del repositorio es un mero ejemplo en el que, por ejemplo, no se contempla que puedan existir el mismo nodos con dos costes distintos.

En primer lugar hay que considerar que el problema presentado se debe modelizar con un multigrafo, puesto que cada nodo, es decir, cada aeropuerto, puede estar unido a otros por más de una arista, cada vuelo, representado por la ID de vuelo y el coste (duración en minutos). Por esto, en el proyecto se trabaja con t-uplas [aeropuerto_destino, vuelo, coste_acumulado].

A continuación se muestra la función A* empleada

```
% Búsqueda A*
% a_star(A, B, C)
% A: Lista de nodos frontera
% B: Nodo objetivo
% C: Lista de nodos recorridos
a_star([[Target, FlightID, Cost] | _],Target,[Target, FlightID, Cost]).
a_star([[Airport,CurrentID,CurrentCost] | FRest],Target,Route) :-
    % Todos los nodos acesibles desde el actual (y el vuelo con el que se accede)
    findall([X,FlightID, CumulativeCost], (arc([Airport,FlightID,Cost],X), CumulativeCost is Cost+CurrentCost, after(FlightID, CurrentID)), NodeList),
    add_to_frontier(NodeList,FRest,Result,Target),                    % Añadimos los pares nodo, vuelo a la lista
    a_star(Result,Target,Found),
    append([Airport, CurrentID, CurrentCost], Found, Route). % Añadimos a la lista de historial
```

Se trata de un algoritmo recursivo en el que, mientras la frontera/ lista abierta tiene nodos, y el nodo estudiado no es el objetivo, se explora el nodo actual, generando sus sucesores, y se realiza una llamada recursiva con los nodos restantes en la frontera.

Es reseñable el uso de la función findall para encontrar todos los nodos sucesores del actual, considerando una hora de salida posterior a la del actual.

He realizado un desarrollo incremental por partes, en el que en un primer momento se consideraban todas las conexiones posibles, y posteriormente añadí la restricción indicada.

Respecto a la función para añadir a la frontera, en un primer momento, por motivos de desarrollo, permití que en la frontera se añadiera cualquier nodo, posteriormente, usé el método built-in “member” para detectar si un nodo representa un aeropuerto que ya estaba en la lista. De esta forma, si el nodo estudiado representa un aeropuerto ya presente, pero con coste menor, se añade. De lo contrario, se descarta.

Al tratarse de una lista, actualizar un elemento no es trivial, por lo que se usa una función auxiliar para eliminar el nodo con coste mayor, y otra para insertar el nuevo.

```
% Añadir a la frontera, nodos pendientes de revisar
% add_to_frontier(A, B, C, D)
% A: Nodos que se añaden
% B: Frontera
% C: Resultado = A+B
% D: Nodo objetivo, usado para determinar el orden de adición a la frontera
add_to_frontier([],X,X,_).                          % Parada
% El aeropuerto no está en la lista
add_to_frontier([[Airport,FID,Cost]|X],Frontier,New,Target) :- not(member([Airport,_,_],Frontier)),
                                                    insert([Airport,FID,Cost],Frontier,Result,Target),
                                                                                    add_to_frontier(X,Result,New,Target).              % Llamada recurs
% El aeropuerto si está en la lista, pero el vuelo actual consigue un coste menor
add_to_frontier([[Airport,FID,Cost]|X],Frontier,New,Target) :- member([Airport,OldFlightID,OldCost],Frontier),
                                                    OldCost > Cost,
                                                    remove([Airport,OldFlightID,OldCost], Frontier, NewFrontier),
                                                    insert([Airport,FID,Cost],NewFrontier,Result,Target),
                                                                                    add_to_frontier(X,Result,New,Target).    % Llamada recursiva
add_to_frontier([[Airport,_,_]|X],Frontier,New,Target):- member([Airport,_,_],Frontier), add_to_frontier(X,Frontier,New,Target).
```

Respecto a la función de inserción, se realiza inserción ordenada, de tal forma que la cabeza de la frontera es siempre el nodo con menor coste estimado f(x), siendo f(x) = g(x) + h(x), con g(x) = coste acumulado y h(x) = heurística.

2.3 Función heurística

En este caso se ha empleado como función heurística la distancia entre el aeropuerto estudiado y el aeropuerto objetivo.

Para calcular esta distancia, partimos de que la base de conocimiento de los aeropuertos contiene la longitud y latitud de cada uno. De esta forma, se puede calcular la distancia haciendo uso de la función ‘haversine’ o función de semiverseno, que he empleado como función auxiliar.

```
% Calcula la distancia entre 2 coordenadas
haversine_distance(Lat1, Lon1, Lat2, Lon2, Distance) :-
    R is 6371, % Earth radius in kilometers
    DLat is (Lat2 - Lat1) * pi / 180,
    DLon is (Lon2 - Lon1) * pi / 180,
    A is sin(DLat / 2) * sin(DLat / 2) + cos(Lat1 * pi / 180) * cos(Lat2 * pi / 180) * sin(DLon / 2) * sin(DLon / 2),
    C is 2 * atan2(sqrt(A), sqrt(1 - A)),
    Distance is R * C.
```

Otras heurísticas que consideré fue el nivel de conectividad del aeropuerto, siendo este el número de vuelos que opera. Es decir, un aeropuerto con más vuelos debería ser estudiado antes que otro con menos vuelos, ya que es más probable que conecte con el destino, por lo que se deberia sumar 100/(número de vuelos), o algun valor similar. No obstante, consideré que la calibración y la comprobación de la aceptabilidad de la heurística suponía demasiado trabajo para el alcance del proyecto.

2.4 Restricción de resultados

```
% Comprueba si el vuelo A comienza después del vuelo B
after(A, B):- flight(A, _, _, TimeA, _, _), flight(B, _, _, TimeB, _, _), TimeA @> TimeB.
after(_, 0). % Se reserva el código 0 para la iteración inicial
```

No todos los vuelos que parten del aeropuerto actual son válidos, solamente consideramos aquellos que comienzan a una hora posterior al actual. Las horas se han declarado en la base de conocimiento como strings del formato “HH:MM”, por lo que para poder compararlas se ha empleado el operador ‘@>’, que permite comparar cadenas alfabéticamente, y numéricamente por ende.

2.5 Formateo de E/S

Para la interacción con el usuario se han empleado los predicados Prolog “write()”, para mostrar información, y “read()”, para leer la entrada del usuario. Debe señalarse que el predicado “read()” lee cláusulas Prolog, por lo que requiere que la entrada del usuario termine necesariamente en un punto, lo que puede considerarse una ligera inconveniencia.

Exploré otras alternativas al predicado read(), no obstante, la mayoría de tutoriales e información usan dicho predicado, a pesar de que en la propia documentación de Prolog se indica que no es el más adecuado para realizar entrada y salida literal con el usuario. No encontré ninguna alternativa razonable que realizara la lectura de entrada, por lo que el predicado read(), pese a sus limitaciones, parece una opción razonable.

```
% Función principal: búsqueda de ruta entre dos aeropuertos
search :-
    write('Enter the origin airport code: \n'), read(Origin),
    write('Enter the destination airport code: \n'), read(Destination),
    a_star([[Origin, 0, 0]|[]], Destination, Route),
    pretty_route(Route, PrettyRoute),
    write('Shortest route from '), write(Origin), write(' to '), write(Destination), write(': '), write(PrettyRoute), nl.
```

Como problema durante la implementación del programa, puede mencionarse que si se intentaba leer el código del aeropuerto en la misma línea que se imprimía la solicitud, se producía un bug con la terminal en el que al introducir las comillas el texto se solapaba con el ya impreso. Esto lo solucioné simplemente insertando un salto de línea, de tal forma que la entrada del usuario se introduce en una línea posterior al texto mostrado que indica la información solicitada.

Para poder mostrar la salida en un formato amigable al usuario he determinado que la mejor solución consiste en usar una función auxiliar, que para cada vuelo de la ruta lo transforma en una cadena usando el predicado de Prolog ‘format/3’. Dicho predicado toma una cadena para el resultado, otra cadena con el formato y una lista de las variables que se deben sustituir en el formato indicado.

```
% Función para poner bonito el resultado
pretty_route([Airport, _, _| X], Result) :-
    pretty_route_aux(X, Rest),
    format(string(Result),"\n\nStarting airport: ~w\n~w", [Airport, Rest]).
pretty_route_aux([Airport, Flight, Duration| X], Result) :-
    pretty_route_aux(X, Rest),
    format(string(Result),"Airport: ~w, FlightID: ~d, Accumulated time: ~d.\n~w", [Airport, Flight, Duration, Rest]).
pretty_route_aux([], ""). % Caso base
```

3. Oportunidades de mejora

Existen varias mejoras posibles para el proyecto que no he podido acometer por limitaciones de tiempo, algunas ya se han mencionado a lo largo del documento y sólo se listan brevemente, mientra que las inéditas se describen en más detalle.

- Mejora de la E/S: en vez de que se introduzca el aeropuerto, se podría introducir una provincia/región y listar todos los posibles aeropuertos, ya que esta información está en la base de conocimiento.
- Realización de múltiples búsquedas: actualmente se considera un único aeropuerto de partida y de llegada. Se podría permitir que se introduzca una región de partida (por ejemplo, Tenerife)
- Considerar el tiempo pasado en los aeropuertos sin tomar vuelos como parte del coste.
- Mejora de la función heurística considerando más factores o factores distintos.
- Considerar el coste económico de los vuelos, de cara a ordenar posibles rutas, o usarlo como criterio de búsqueda.

4. Ejecución del programa

4.1 Instrucciones de uso

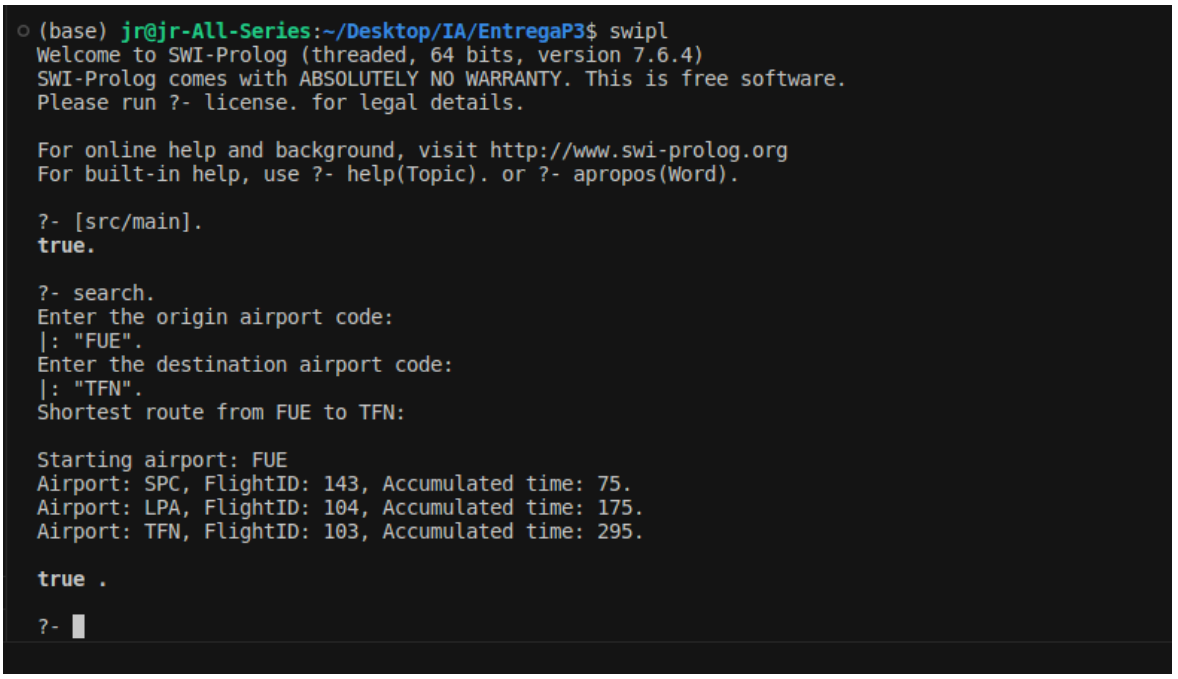
En primer lugar, para que el programa funcione correctamente, es necesario modificar el fichero src/main.pl, concretamente las líneas 14 y 15, de forma que la ruta indicada sea la de las bases de conocimiento en /data, pero debe ser la ruta absoluta de su ordenador.

```
13  % Importamos datos
14  :- consult('/home/jr/Desktop/IA/EntregaP3/data/flights.pl').
15  :- consult('/home/jr/Desktop/IA/EntregaP3/data/airports.pl').
16
```

Para ejecutar el programa es necesario lanzar swiprolog, cargar el fichero que contiene el programa (main.pl), por ejemplo, mediante '[src/main]' y consultar

el predicado principal, 'search.' que muestra una interfaz por consola interactiva, en la que se deben indicar los códigos IATA del aeropuerto origen y el de destino, entre comillas dobles y terminados por un punto cada uno.

4.1 Ejemplo



En este ejemplo se busca la ruta de menor duración entre Fuerteventura (FUE) y Tenerife Norte (TFN). Observamos que en la base de conocimiento de la que disponemos, esta ruta consiste en tomar 3 vuelos:

Un primer vuelo a Santa Cruz de La Palma (SPC) con duración de 75 minutos y código de vuelo 143.

Un segundo vuelo desde La Palma hasta Las Palmas de Gran Canaria (LPA), con código de vuelo 104, llevando una duración de vuelo acumulada de 175 minutos.

Finalmente, un tercer vuelo desde Gran Canaria hasta Tenerife Norte, con ID 103 y suponiendo una duración acumulada de 295 minutos.