Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Prolog Project

Air Route Finder

Artificial Intelligence

Bsc in Computer Science

Jose Ramón Morera Campos

(alu0101471846@ull.edu.es)

INDEX

1. Introduction

1.1 Problem Definition

The project consists of a system for searching air routes between two given airports, considering both direct flights and connections. Flight departure and arrival times are taken into account so that only feasible connections are proposed. The goal is to minimize travel time, defined as the sum of the durations of the flights taken.

The output shows the flights that must be taken to travel from the origin airport to the destination airport with the shortest possible flight time.

1.2 Application Domain

The problem belongs to the field of logistics, specifically route search and optimization.

1.3 Related Work

Several projects with similar features developed in Prolog were found:

* '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. Development process

2.1 Data adquisition

First, I searched for the data to include in the knowledge base, in order to determine what the program could and could not do depending on the characteristics of that data.

I finally decided to use two knowledge bases, located in the `data` folder: `airports.pl` and `flights.pl`. The first contains information on 90 airports from the Canary Islands, Spain, and several other countries. This information was obtained from https://openflights.org/data.php

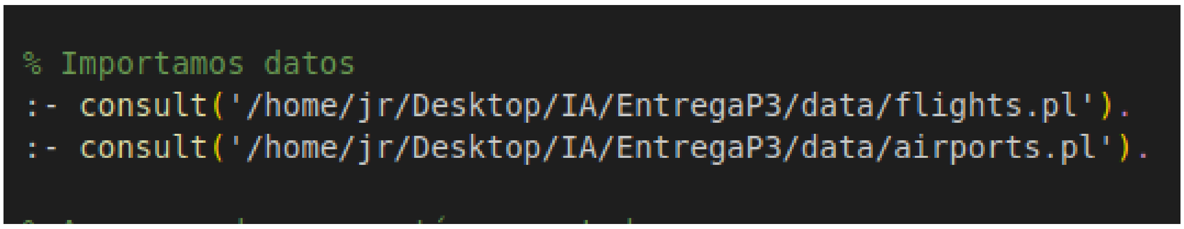
The raw data needed processing to discard attributes irrelevant to the project and to format them in Prolog style, so that each row became a predicate. Regular expressions were used for this task.

The second base, `flights.pl`, contains a list of flights between the airports in the first base. To generate it, I used an AI tool, providing it with the list of airports and the desired format for flight predicates, resulting in 250 flights.

It should be noted that realism of the data was never the goal, but rather representativeness to build a “toy” knowledge base that would allow testing in pseudo-realistic scenarios.

To simplify the problem, it was assumed that all flights occur daily at the same schedule.

For data loading, the Prolog built-in predicate `consult` was used, indicating the absolute path of the files to be scanned. These predicates were declared as directives in `main.pl` using the `:-` operator, so they are processed upon loading the file.



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

2.2 A* Algorithm

The problem tackled is the search for minimum-duration routes between airports. To perform efficient searches, I implemented the A* algorithm.

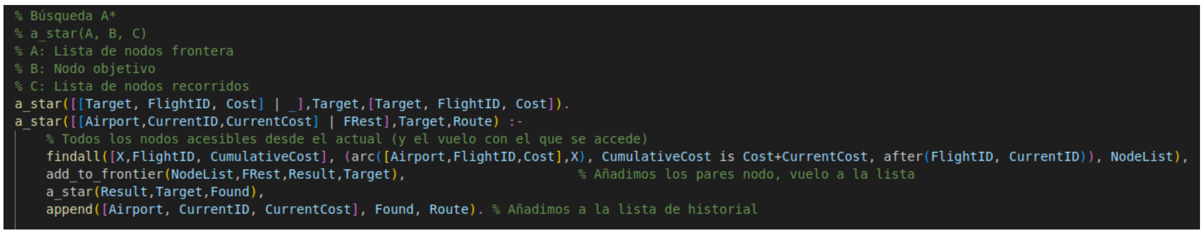
I chose flight time as the cost to minimize. Although I initially considered including ticket prices, I concluded this factor was unsuitable because flight prices are determined by non-deterministic factors, fluctuate in real time, and vary between airlines, times, etc.

Therefore, this project focuses only on minimizing total flight duration, i.e., the accumulated cost is the sum of the durations of the flights taken.

For implementation, I used https://github.com/ciaranfinn/a-star as a reference. However, I developed my own version with significant modifications, since that repository is just a simple example that, for instance, does not account for the same node having two different costs.

The problem must be modeled as a multigraph, since each node (airport) can connect to others through multiple edges (flights). Each flight is represented by the flight ID and cost (duration in minutes). Thus, the project works with tuples `[destination_airport, flight, accumulated_cost]`.

Here, the function used is shown:



```prolog
% 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
```

The A* function is recursive: while the frontier/open list has nodes and the current node is not the target, the current node is explored, successors are generated, and a recursive call is made with the remaining nodes in the frontier.

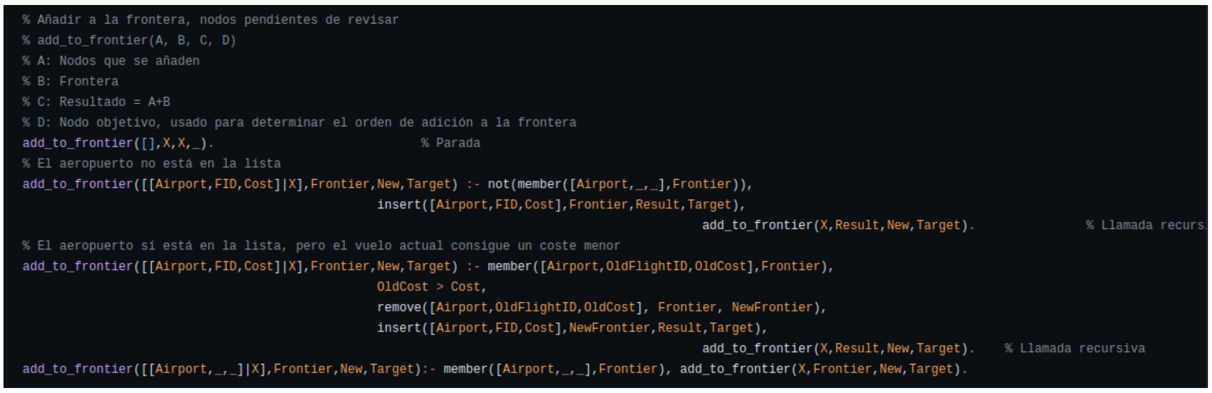
Key points:

- The `findall` function is used to find all successors of the current node, considering only flights departing later than the current one.

- Frontier management: initially, any node could be added. Later, I used the built-in `member` predicate to check whether a node represented an already-listed airport. If the airport was already in the list but with a higher cost, the new node replaced it; otherwise, it was discarded.

- Since updating an element in a list is not trivial, an auxiliary function was implemented to remove the node with the higher cost, and another to insert the new one.

- The insertion is ordered so that the frontier head is always the node with the lowest estimated cost *f(x) = g(x) + h(x)*, where *g(x)* is the accumulated cost and *h(x)* is the heuristic.



```prolog
% 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 sí 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).
```

2.3 Heuristic Function

The heuristic used is the distance between the current airport and the target airport.

To compute this distance, the knowledge base provides latitude and longitude for each airport. Thus, the `haversine` (great-circle distance) formula was used as an auxiliary function.

```
% 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.
```

Another heuristic I considered was the airport's connectivity level, i.e., the number of flights it operates. An airport with more flights should be prioritized, since it is more likely to connect with the target. This could be expressed as adding `100 / (number of flights)` or similar. However, calibrating and verifying the admissibility of such a heuristic required more effort than was feasible within the project scope.

2.4 Results restrictions

```
% 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
```

Not all flights departing from the current airport are valid—only those that depart **after** the current one. Times were declared in the knowledge base as strings in `HH:MM` format. To compare them, the `@>` operator was used, which compares strings alphabetically and thus numerically in this case.

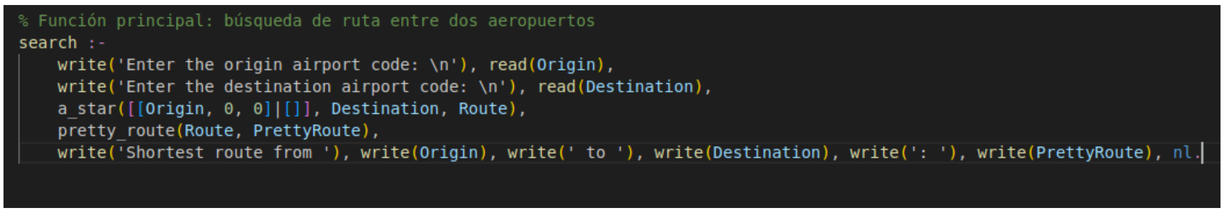
2.5 Input/Output formatting

User interaction is managed with Prolog predicates:

- `write()` – to display information
- `read()` – to read user input

The `read()` predicate reads Prolog clauses, which means user input must **end with a period**—a minor inconvenience.

I explored alternatives, but most tutorials and resources still use `read()`. Although the official documentation notes it is not ideal for literal I/O, I found no reasonable alternative, so `read()` was chosen despite its limitations.

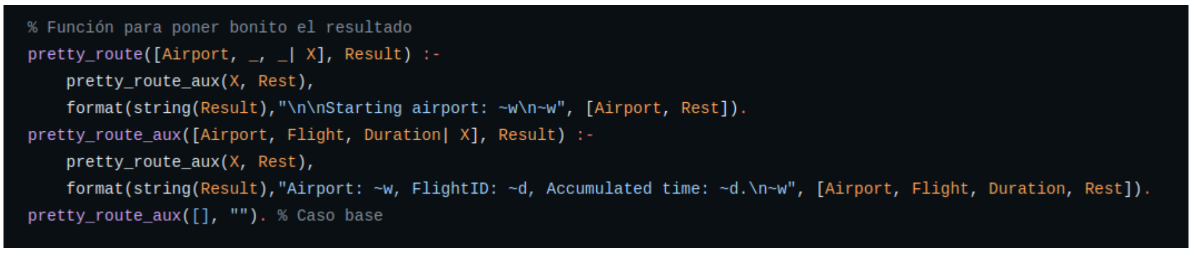


```
% Función principal: búsqueda de ruta entre dos aeropuertos
search :-
    write('Enter the origin airport code: \n'), read(Origin),
    write('Enter the destination airport code: \n'), read(Destination),
    a_star([[Origin, 0, 0]|[]], Destination, Route),
    pretty_route(Route, PrettyRoute),
 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

Implementation issue:
If the airport code was read on the same line where the prompt was printed, the terminal bugged out: entering quotes caused text overlap. I solved this by inserting a newline, so user input is always entered on the following line.

Output formatting:
To display results in a user-friendly way, I implemented an auxiliary function that converts each flight in the route into a string using Prolog's `format/3` predicate. This predicate takes:

- A result string (which will save the result)
- A format string
- A list of variables to substitute using the indicated format



```
% 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. Opportunities for improvement

Several enhancements could be made but were not addressed due to time constraints. Some were already mentioned; others are listed here in more detail:

- Input/Output improvement: instead of requiring a specific airport, allow the user to enter a province/region and list possible airports (data already exists in the knowledge base).

- Multiple searches: currently only one origin and one destination airport are supported. Could extend to regions (e.g., "Tenerife").

- Consider layover time at airports as part of total cost.
- Improve the heuristic function by considering more or different factors.
- Consider the economic cost of flights, either to rank routes or as a search criterion.

4. Ejecución del programa

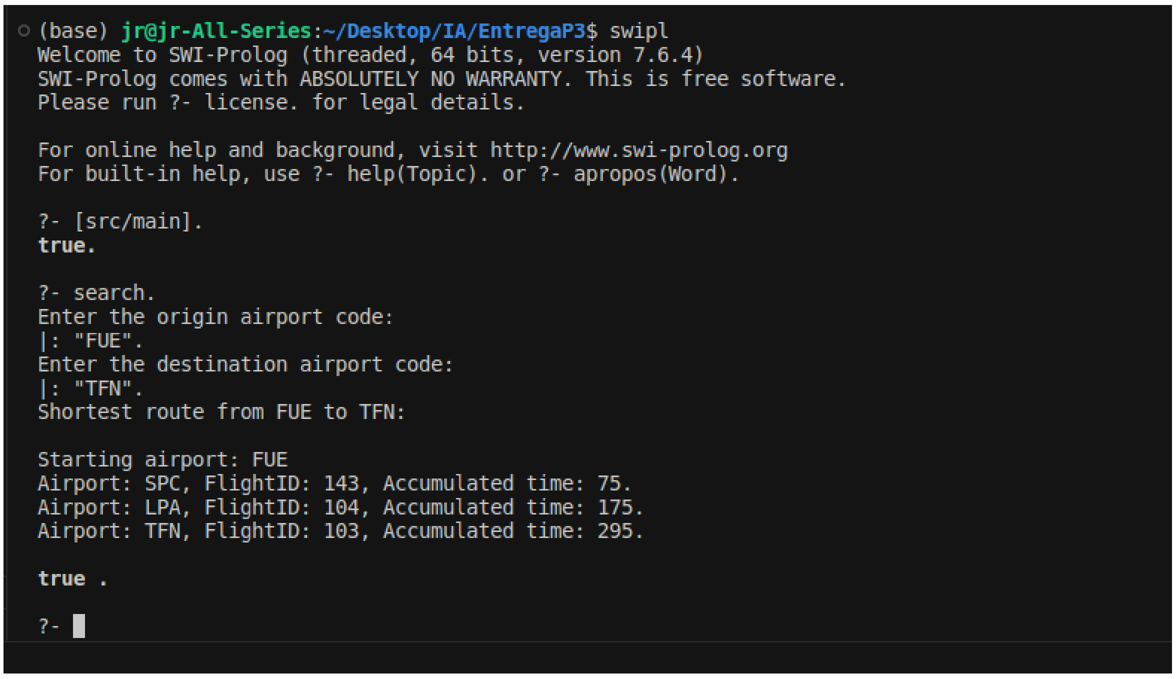
4.1 Instrucciones de uso

To run the program, start SWI-Prolog, load the file containing the program (`main.pl`), for example using: ?- *['src/main'].*

Then call the main predicate: ?- *search.*

This launches an interactive console interface where the user must input the IATA codes of the origin and destination airports, enclosed in double quotes and ending with a period.

4.1 Example



```
(base) jr@jr-All-Series:~/Desktop/IA/EntregaP3$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [src/main].
true.

?- search.
Enter the origin airport code:
|: "FUE".
Enter the destination airport code:
|: "TFN".
Shortest route from FUE to TFN:

Starting airport: FUE
Airport: SPC, FlightID: 143, Accumulated time: 75.
Airport: LPA, FlightID: 104, Accumulated time: 175.
Airport: TFN, FlightID: 103, Accumulated time: 295.

true .

?-
```

In this example, the shortest route between Fuerteventura (FUE) and Tenerife North (TFN) is searched. In the available knowledge base, this route consists of 3 flights:

- A first flight to Santa Cruz de La Palma (SPC), lasting 75 minutes, with flight code 143.
- A second flight from La Palma to Las Palmas de Gran Canaria (LPA), with flight code 104, bringing total accumulated flight time to 175 minutes.
- Finally, a third flight from Gran Canaria to Tenerife North, with ID 103, reaching a total accumulated duration of 295 minutes.