**Student:** Giusppe Esposito
**Student number:** 22702705

---

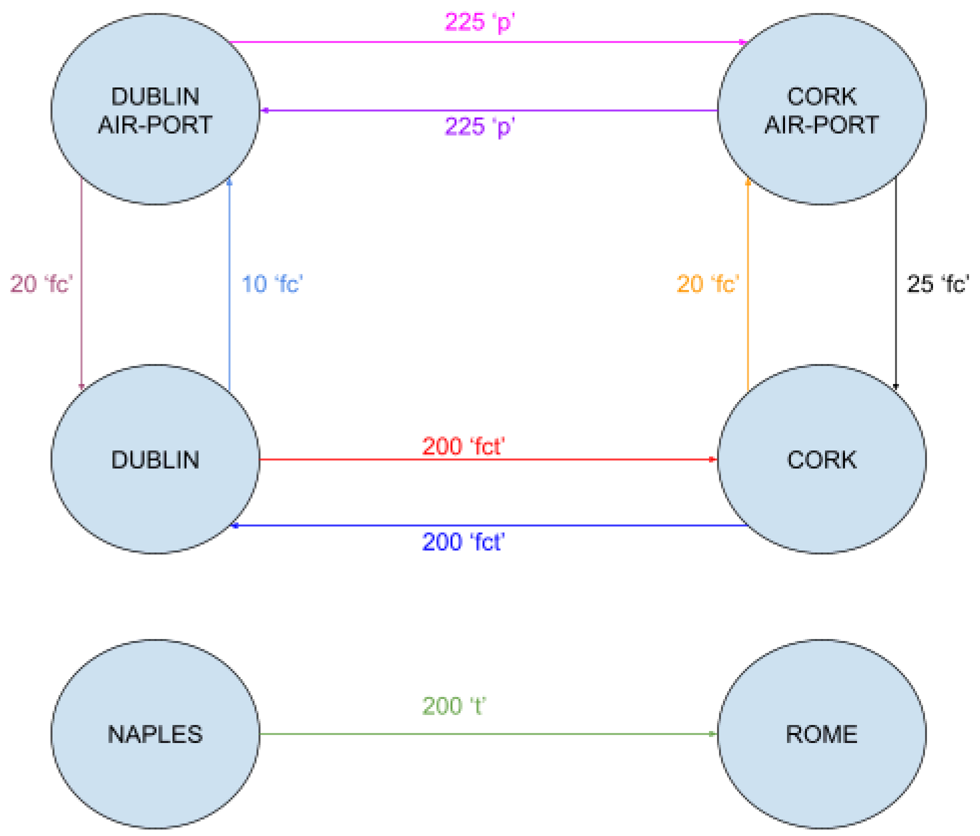
## -Structure of the Facts:

```
%Travel Modes (speed in km/h)
speed('f', 5).
speed('c', 80).
speed('t', 100).
speed('p', 500).
```

These facts defines the speed for each travel mode:
-’f’  stands for on foot,    with speed      5 km/h
-’c’ stands for by car,       with speed     80 km/h
-’t’  stands for by  train,   with speed    100 km/h
-’p’ stands for by plane,  with speed    500 km/h.

```
%Routes (distance in km)
route(dublin, cork, 200, 'fct').
route(cork, dublin, 200, 'fct').
route(cork, corkAirport, 20, 'fc').
route(corkAirport, cork, 25, 'fc').
route(dublin, dublinAirport, 10, 'fc').
route(dublinAirport, dublin, 20, 'fc').
route(dublinAirport, corkAirport, 225, 'p').
route(corkAirport, dublinAirport, 225, 'p').
route(naples, rome, 200, 't').
```

Each route facts of this form:
**route(PointA, PointB, Dist, TravelMode).**
-Establish a direct one way connection from **PointA** to **Point B**,
-The distance of this connection is **Dist** (an integer in Km),
-And travel modes that we can use are the characters in **TravelMode.**



The set of previous facts roughly represents this graph above.
So there is a circle between Points ( Dublin, Dublin-airport, Cork and Cork-airport), and then two detached points (Naples and Rome).

# -Structure of the Predicates and Rules:

The goal of the program is to *“Write a Prolog predicate journey(S, D, M) that calculates the quickest journey between S and D only using the travel modes included in the string M. Your predicate must be able to handle cycles in a set of facts.”*

## -This is the main predicate:

```
journey(S, D, M):-
        get_all_paths(S, D, M, Paths),
        get_min_path(Minpath, Paths),
        writeln(Minpath),!.
```

This predicate use:

**get_all_paths(S, D, M, Paths)**: to get a list of all the paths called **Paths** where each element is of the form [time, path], where time is the time needed to travel the path, and path is the list of points that leads you from S (start) to D (destination).

**get_min_path(Minpath, Paths)**: to get the quickest path **MinPath**, by selecting the element with the smallest time.

**writelen(Minpath)**: And finally it prints the Minpath.

## -How does get_all_paths work?

```
get_all_paths(Start, Destination, Modes, Paths):-
        findall([Time, Path], build_path(Start, Destination, Modes, [Start], Path, Time), Paths).
```

It just uses the built-in function findall, to fill the variable **Paths** with all the returned result from **build_path()** predicate.

## -How does build_path work?

```
build_path(Start, Destination, Modes, Visited, [Start, Destination], Time):-
        route(Start, Destination, Distance, PossibleModes),
        not_in(Destination, Visited),

        get_travel_modes(PossibleModes, Modes, TravelMods),
        get_delta_time(TravelMods, Distance, Time).



build_path(Start, Destination, Modes, Visited, [Start|Path], Time):-
        route(Start, Between, Distance, PossibleModes),
        not_in(Between, Visited),

        get_travel_modes(PossibleModes, Modes, TravelMods),
        get_delta_time(TravelMods, Distance, Dtime),

        build_path(Between, Destination, Modes, [Between|Visited], Path, TailTime),

        Time is Dtime + TailTime.
```

The general from of this predicate is
**build_path(Start, Destination, Modes, Visited, Path, Time):**

**Start** is the point where we start;
**Destination** is the point that we want to reach;
**Modes** are the travel Modes that we can use;
**Visited** is a list to store all the place already visited to avoid loop, due to cycle in the graph, we start that as **[Start],** than we update it;
**Path** is a list of all the point that we go through to get to our Destination;
**Time** is the amount of time needed to walk **Path**.

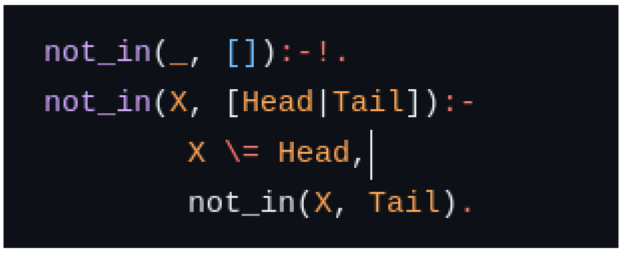
This predicate is divided in two rules:

1)The first is the base case, where there is a direct link between the **Start** and **Destination** (see first line), defined by a route fact in the datPathabase.

First it updated the **Path** so it became [Start, Destination].

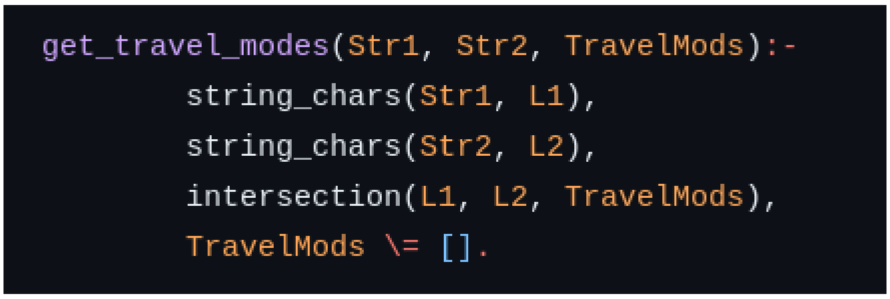
The route() call gives us access to the **Distance** between Start and Destination and **PossibleModes** that we can use.

Then we check if the **Destination** is already visited to avoid a loop using **not_in()**.



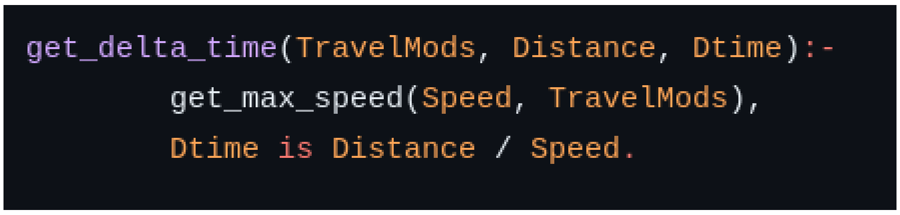
```
not_in(_, []):-!.
not_in(X, [Head|Tail]):-
        X \= Head,
        not_in(X, Tail).
```

Then we get a list of characters that represent all the possible travel modes called **TravelModes**, by using **get_travel_modes()**, this predicate just take the intersection of the modes that we want to use (**Modes**), and the modes that we need to use (**PossibleModes**), and store that in the list **TravelModes**. And it fails if there is no intersection. *Because for example if I want to use a plane to go from A to B, but the only way allowed is on foot, I can't go there.*



```
get_travel_modes(Str1, Str2, TravelMods):-
        string_chars(Str1, L1),
        string_chars(Str2, L2),
        intersection(L1, L2, TravelMods),
        TravelMods \= [].
```

In the end we set the variable **Time** using **get_delta_time(TravelModes, distance, Time)**, that is just calculating the time to go from Start to Destination, using the quickest TravelMode speed, get with **get_max_speed()**.



```
get_delta_time(TravelMods, Distance, Dtime):-
        get_max_speed(Speed, TravelMods),
        Dtime is Distance / Speed.
```

We are setting and not updating the **Time** variable because we are in the base case, so the last recursion call before starting go back, during the backtracking we are going to update **Time**. This technique is the opposite one that we use to build the **Path.** In fact, we update the path before the recursion call.
Therefore in the last call (the base case) the **Path** is already complete.

2)The second is the recursive where there is no direct link between **Start** and **Destination**
The approach is:

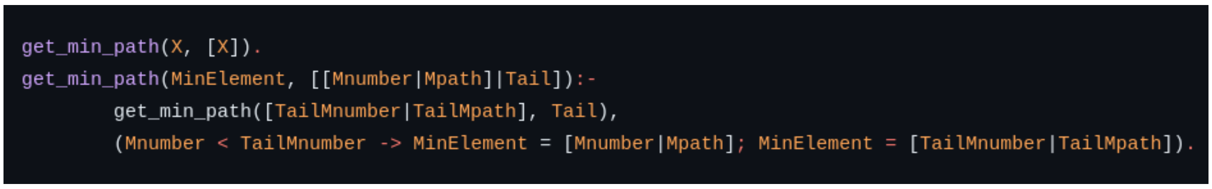
-we search for some point **Between**,that is directly reachable from **Start**,

-we do the same check and get the data as Time is Dtime + TailTime.in the base case (**not_in()**, **get_travel_modes()**, **get_delta_time()**)

-Then we recall the function to search if there is a link from **Between** to **Destination**

NOTE: that we are updating the path before the recursive call in fact is becomes **[Start, Path]**, where **Path** is whatever is going to return the recursive call,
and we are updating the **Time** after the recursive call: **Time is Dtime + TailTime.**

## -How does get_min_path work and get_max_speed?

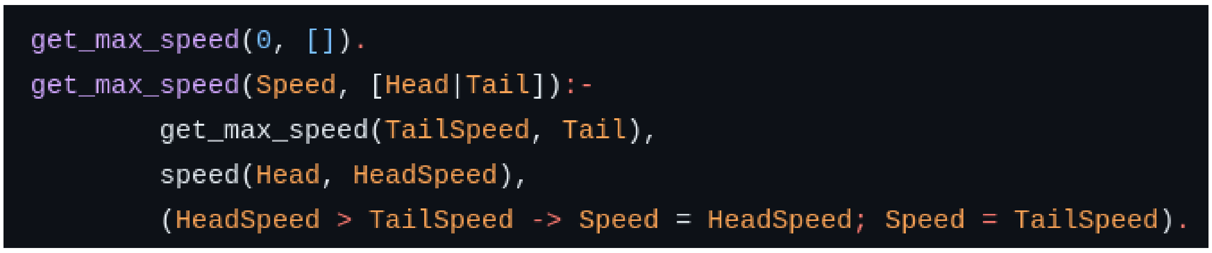


```
get_min_path(X, [X]).
get_min_path(MinElement, [[Mnumber|Mpath]|Tail]):-
        get_min_path([TailMnumber|TailMpath], Tail),
        (Mnumber < TailMnumber -> MinElement = [Mnumber|Mpath]; MinElement = [TailMnumber|TailMpath]).
```

Given a list of elements [Time, Path], it gives you the element with the smaller time, so the quickest.

It works with the assumption that the minimum element of a list composed of just one element is the element itself.

Then we just simplify the problem by saying that the minimum element (**MinElement**) between head and tail is whatever is less between the value of Time in the head(**Mnumber**), and the minimum of the Tail(**TailMnumber**)
.



```
get_max_speed(0, []).
get_max_speed(Speed, [Head|Tail]):-
        get_max_speed(TailSpeed, Tail),
        speed(Head, HeadSpeed),
        (HeadSpeed > TailSpeed -> Speed = HeadSpeed; Speed = TailSpeed).
```

It basically uses the same strategy of **get_min_path()**, but the assumption is that the max element in an empty list is 0, and then it checks for the greater value.

## -Reference:

Everything in this file is considered my own work with the except of the following resources:

**[1]“SWI-Prolog -- findall/3,” *www.swi-prolog.org*. https://www.swi-prolog.org/pldoc/man?predicate=findall/3 (accessed Apr. 05, 2024).**

**[2]“SWI-Prolog -- -%3E/2,” *www.swi-prolog.org*. https://www.swi-prolog.org/pldoc/man?predicate=-%3E/2 (accessed Apr. 05, 2024).**

**[3]“SWI-Prolog -- string_chars/2,” *Swi-prolog.org*, 2020. https://www.swi-prolog.org/pldoc/man?predicate=string_chars/2 (accessed Apr. 05, 2024).**

**[4]W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. Berlin ; New York: Springer-Verlag, 2003.**

**[5]“Prolog Guide - Metainterpreters,” *kti.ms.mff.cuni.cz*. http://kti.ms.mff.cuni.cz/~bartak/prolog/graphs.html#dijkstra (accessed Apr. 05, 2024).**

**[6]“Find the shortest path in between nodes,” *SWI-Prolog*, May 12, 2020. https://swi-prolog.discourse.group/t/find-the-shortest-path-in-between-nodes/2315/3 (accessed Apr. 05, 2024).**