Hafaki

ET4727 project report

Oliver Gerler
9954465
ET4727: Artificial Intelligence & Expert Systems
University of Limerick, Ireland

December 4, 1999

To Anarwenya

Introduction

Project definition

The specification demands the implementation of a timetable for train services between Limerick, Cork and Dublin in both directions. The program has to contain information on the validity dates of the timetable, the actual timetable, the days of operation, the buffet service on bord and the originating station of the used train. It must further be able to tell the user if a change of trains along the desired route will be necessary.

The program has to be implemented using the SWI-`PROLOG` software package.

On Finding the Name

There is a complex, large and fast (German) implementation of a timetable for European train services named *Hafas*. This database has been of great use and can be found at `http://bahn.hafas.de/`. The Austrian Federal Railways use the same database with a different frontend (`http://fahrplan.oebb.at/`). It contains all actual train services in Europe, but only extends to Dublin Port, Dun Laoghaire and Rosslare Harbour. There is no data for Irish trains.

This project now tries to represent a small version of *Hafas*. It seemed suitable to name it **Hafaki**, using the Greek diminutive ending -*aki*.

Status

As the aim of this project was to build a partial timetable for IrishRail, the contents and possibilities of this implementation grew with time, so that *Hafaki* now contains all 105 served Irish trainstations and all 14 routes.

Still missing are the specific timetables for the routes, except routes 1, 2, 3, which connect Dublin-Cork, Dublin-Limerick-Ennis (via Limerick Junction) and Ballybrophy-Limerick (via Nenagh) respectively, as was stated in the project definition.

Design And Testing

Idea

As described earlier the idea of this project is to implement a timetable for the train services run by Irish Rail.

Features

In contrary to the basic outline as given in the project specification, this implementation is not restricted to the 3 routes. In especially the mapping of the routes, the connections between stations and the outline of the routes is completely independent of the function of the main blocks in the program. It does, therefore, not matter what the layout and networking of the routes looks like; the engine is unspecialized.

The implementation state at this time consists of all 105 train stations that are currently served by Irish Rail. The structure and design of all 14 different routes is "known" to the system. The timetable of the 3 required routes is stated in the definition part of the program. This three routes are from Cork and Limerick to Dublin via Limerick Junction and from Limerick to Ballybrophy via Nenagh (where connections to the first and second route are made).

The departure and arrival station can be chosen freely from any station on the implemented route. It is for example possible to ask for a timetable between Cloughjordan and Kildare. The system will recognise, that the first route ends at Ballybrophy and a change must be made to another route. This is not restricted to two consecutively routes but can be any number of connections between any stations.

The main engine is able to find multiple alternatives. If it is asked to find the routes from for example Roscrea to Rosslare it will answer with 3 different routes: one via Dublin, one via Kildare and Waterford and one via Limerick Junction and Waterford. Effort has been made to build this main engine as unspecialized as possible and so it is in effect not restricted to any given layout.

The input and output of the station names can either be in English or in Gaelic. The guidance of the user will be English at any time.

Usage

The usage will be explained on a example. In this example the timetable for trains connecting Limerick and Nenagh is requested, using the Gaelic names. Only bold face text in the example has to be entered by the user.

```
184 ?- timetable.

                 ***** HAFAKI *****
               (C) 1999 Oliver Gerler
          valid from 19/09/99 to 27/05/00.

Which language do you prefer for the names of the stations?

1.english     2.gaelic     2

Gaelic names chosen.

On which day do You want to travel?

1.mo 2.tu 3.we 4.th 5.fr 6.sa 7.su     1

Monday chosen.

Do You want to start AM or PM?

1.am 2.pm     1

AM chosen.

Please enter stationnames.
Enclose them in apostrophes and end with full stop and return.

From:'Luimneach'.
To:'An tAonach'.

     Days of operation:  mtwtfs.
      Onboard services:   none

             Luimneach:      0715
           Ballybrophy: 0858-1018
            An tAonach: 1110
Press return.
     Days of operation:  mtwtfs.
      Onboard services:   none

             Luimneach:      0715
            An tAonach: 0804
Press return.
```

```
Yes
185 ?-
```

The program has to be started by consulting the program file by entering `consult(hafaki).` at the `PROLOG`-prompt and using `timetable.` as entry.

Only 5 questions have to be answered to get a full timetable with alternative routes (where possible) between two stations. First the user is asked to choose the language in which the names of the stations are expected and will be written. It is followed by questions for day and time of the desired travle. To define the start point and the destination, the names of the stations must be entered enclosed in apostrophes and with a trailing full stop. This is due to `PROLOG`. If the name was misspelled or the station does not exist an error message will be displayed and the name is prompted again. The execution of the programm can be stopped by pressing Ctrl-C at any time.

Principle of Function

As `PROLOG` offers the functionality heavy use of recursions has been made.

The program is mainly split into two sections: a definition part and a procedural part. These two parts are independent, so changes can be made without afflicting the other part.

The definition contains the structure of the network, the names and affiliations of the stations, the definition of the routes and the entries for the timetable.

In the procedural part the engine for finding the routes between two stations and assigning the timetable to the chosen routes can be found. This section makes the major part of the actual program. It has been designed not to be specialized in any way and to work with any given data structure within the given syntax.

The collecting of the data for a timetable is done in two phases. First a data sturcture containing the possible different routes between two stations will be constructed and filled. This structure will then be compared to the time tables and suitable entries chosen. So it is no problem to ask for the routing data only without having to know about the additional information concerning the timetable.

Modular Structure

The procedural section of the program is built very modular. This modules can be changed or altered without the knowledge of the internas of the other modules beside their interface.

This modularization is processed throughout the whole program. Each module is encapsulated within its well defined interface and does not otherwise interfere with other modules of the program.

It has been tried to keep the modules small and simple for improving the readability and testability. This, however, has resulted in more than 1000 lines of source code and only the most mportant parts can be shown in the listing at the end of this report.

Central Function - `timetable`

The program can be started with `timetable.` without any parameters. It is of course possible to use the other facilities within the porgram if their interface is known.

The `timetable`-module is constructed as below.

```
% timetable - build timetable of different routes

timetable :-
nl,
tab(20),write('  ***** HAFAKI *****  '),nl,
tab(20),write('(C) 1999 Oliver Gerler'),nl,
tab(15),write('valid from 19/09/99 to 27/05/00.'),nl,nl,
input(Src, Dest, Lang, Day, Time),
findroutes(Routes, Src, Dest),
findroutelinesentry(Routes, StartStopList),
finddirections(StartStopList, DirectionList),
expand(DirectionList, TimesList),
output(TimesList, Lang, Day, Time).
```

As can be seen in this part of the listing, there are only 6 main modules which are necessary for a full timetable. `input` asks the user the questions, which have been described earlier. `findroutes` tries to find as many possible routes from the starting point to the destination as possible. This routes are described as lists containing the stations passed by this route. To get the imformation for the timetable it is necessary to know which line has to be used and if changes are necessary, where they have to take place. `findroutelinesentry` is the module fo that purpose. `finddirections` assign the directions to the routes to be able to read out the timetable in the correct way. Finally `expand` builds the list which contains all necessary timetable related information about a journey from the starting point to the destination. `output` prints this information on the screen.

It seems to be useful to gather information of the routing decisions made by the program. The user can enter `findroutes(R,2,17).` to have the routes between station 2 and 17 (what happens to be Limerick and Waterford) displayed as value `R`.

Data structures

4 main data structures are used in the definition part of the program. This structures define the existing stations, their connections and hence the layout of the network and they contain the data for the timetable.

station

```
station(82, '    Boyle', '    Mainistir na Bille', [83]    , [7]).
```

The stations are defined as facts within the `PROLOG` program. Each fact contains information for one stations, being their internal number, the name of the station in english and gaelic and two fields which can connect multiple entries. The first field contains the "father" of this station and so defines the layout of the network. The root station is No.33, Dublin Connolly. The second field contains all the lines which stop at or pass by this station. The names of the stations are rightbound within a field of 22 charcters for easy printing in a table. All 105 currently served stations are in the database.

line

```
line( 2, [  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]).
```

This defines the lines and the stations wchich are served by this line. The first item in this fact is the internal number of the line followed by a list of the numbers of the stations. There are currently all 14 lines contained in the databse.

train

```
train( 2, 'Back', [
        [' m...... '],  % days
        ['  none   '],  % services
        [ 11,   0,0535],        % Dublin Heuston
        [ 10,   0,   0],        % Newbridge
        [  9,   0,   0],        % Kildare
        [  8,   0,   0],        % Portarlington
        [  7,0625,0625],        % Portlaoise
        [  6,   0,   0],        % Ballybrophy
        [  5,   0,   0],        % Templemore
        [  4,0656,0656],        % Thurles
        [  3,0715,0715],        % Limerick Junction
        [  2,0727,0727],        % Limerick
        [  1,0752,0752] % Ennis
        ]).
```

The `train` facts contain all the time table information that is available for this line. The first number defines the line which is described in this fact. There are two such facts for each line so the second entry defines the direction of the trains herein. The last entry in the fact is a list which contains the actual data. This list contains ine entry for the operating days of the trains, one entry of the available services on bord and one entry for each station that is served. Each station entry is preceeded by its station number. There is one item per train for the operating days (here only monday), one item for th services and two items for each the arriving and the departing time for this train at that station. Not all of the data for the timetables has been entered yet.

node

```
node(66).
```

The node fact states, that in the station with the number contained in the node fact two lines are connecting or departing. These nodes are needed for finding multiple routes to the destination.

Frontend

The frontend consists of two converters, one for input and one for output. Their purpose is to translate the questions entered by the user to the required format, so they can be processed and to reconvert the answers from the system to human readable text and tables.

These converters can be found in `input()` and `output()`.

Testing

The testing of a program can never be finished. There has been thorough testing and extensive bug hunting during development and finalization of this project, though.

The functions of the implemented modules are as desired in normal execution conditions, but cannot be guaranteed in special cases or with uncertain or missing data. Especially because of the lack of the full timetable there will be some situations in which the program in the current state will fail.

Nevertheless, the usage as required (especially the limitation to the 3 specified routes) will work as expected.

Conclusions

The work on this program has been done using telnet connections to the SUN computers. As the programming environment to `PROLOG` is text based this proved to be a very practical way not to be bound to the restrictive opening hours of the laboratories at the UL.

The interpreter which executes the `Prolog` file runs fast and stable. There was no recognizable delay in execution of the program. The meaning of the error messages of the interpreter is quite problem related and can lead to a solution.

There have been some conceptual problems related to the nondeterministic behaviour of `Prolog` which is at least uncommon for programmers who are used to a highly deterministic language and execution of their programs.

Bibliography

[1] Grout Ian Dr., *Handouts to module ET4727,* University of Limerick, 1999/2000

[2] Bratko Ivan, *PROLOG programming for artificial intelligence,* 2nd ed. 1990, repr. 1999, Addison-Wesley, ISBN 0-201-41606-9

[3] Debusmann Ralph, *ADE port of SWI Prolog for AMIGA computers* Aminet:dev/lang/swi020.lha, Email: rade@coli.uni-sb.de

Partial program listing

```
% station names (No., eng.name, gael.name, 'fathers', lines)
station( 1, '      Ennis', '    Inis', [ 2]   , [2,11]).
% lines (route,[stations])
line( 1, [ 78, 77, 90,  3,  4,  5,  6,  7,  8,  9, 10, 11]).
% time table
train( 3, 'Back', [
[' mtwtfs. ',' mtwtfs. ',' ......s '],% days
['  none   ','  none   ','  none   '],% services
[  6,   0,1018,   0,1900,   0,1945],% Ballybrophy
[ 16,1035,1035,1917,1917,2002,2002],% Roscrea
[ 15,1053,1053,1053,1053,2020,2020],% Cloughjordan
[ 14,1110,1110,1952,1952,2037,2037],% Nenagh
[ 13,1134,1134,2016,2016,2101,2101],% Birdhill
[ 12,1143,1143,2025,2025,2110,2110],% Castleconnell
[  2,1200,   0,2045,   0,2126,   0]  % Limerick
]).

% timetable - build timetable of different routes
timetable :-
nl,
tab(20),write('  ***** HAFAKI *****  '),nl,
tab(20),write('(C) 1999 Oliver Gerler'),nl,
tab(15),write('valid from 19/09/99 to 27/05/00.'),nl,nl,
input(Src, Dest, Lang, Day, Time),
findroutes(Routes, Src, Dest),
findroutelinesentry(Routes, StartStopList),
finddirections(StartStopList, DirectionList),
expand(DirectionList, TimesList),
output(TimesList, Lang, Day, Time).

% input
input(Src,Dest,Lang, Day, Time) :-
askLang(Lang),
askDay(Day),
askTime(Time),
```

```
write('Please enter stationnames.'),nl,
write('Enclose them in apostrophes and end with full stop and return.'),nl,nl,
askstation('From:', Lang, Src),
askstation('  To:', Lang, Dest).

% find route between two stations Src and Dest.
findroutes(Routes, Src, Dest) :-
findroutes33(Route1, [[Src]], [[Src]]),
findroutes33(Route2, [[Dest]], [[Dest]]),
findroutescross(Routes, Route1, Route2, [], Route1).

% output - prints traintimes in most beautiful form
output([], _, _, _).

output([TimesListHead|TimesListTail], Lang, Day, Time) :-
output(TimesListTail, Lang, Day, Time),
outputroutes(TimesListHead, OutputList, Day, Time),
tab(5),write('Days of operation:'),printInfo(Days),nl,
tab(6),write('Onboard services:'),printInfo(Services),nl,nl,
printList(NewOutputList, Lang),
write('Press return.'),
get0(_).

% expand - find times to lines
expand([], []).

expand([RouteListHead|RouteListTail], TimesList) :-
expand(RouteListTail, OldTimesList),
expandroute(RouteListHead, TimesListHead),
TimesList = [TimesListHead|OldTimesList].

% finddirections - build new list with THERE and BACK directives
finddirections([], []).

finddirections([SSLHead|SSLTail], DirectionList) :-
finddirections(SSLTail, OldDirectionList),
adddirection(SSLHead, NewSSLHead),
DirectionList = [NewSSLHead|OldDirectionList].

% findroutelinesentry - start recursive search through all routes
findroutelinesentry([], []).

findroutelinesentry([RouteHead|RouteTail], NewStartStopList) :-
findroutelinesentry(RouteTail, OldStartStopList),
findroutelines(RouteHead, StartStopList),
conc(OldStartStopList, [StartStopList], NewStartStopList).
```