Project Report

Introduction

In this project, I build a road pathfinder between two cities. It can find the path in two ways. I have implemented the system for Depth First Search and Best First Search. It takes the algorithm to use and the two cities as input and outputs the path and distance you have to cover following that path.
The whole program is written in the Prolog programming language. It involves using basic concepts of Prolog such as Backtracking, Recursion, etc.

Heuristics and Data representation

I have explicitly represented the data for knowledge. I used a python script to read the CSV file. I have taken all the cities as directly connected as present in the CSV file for depth-first search.
In the case of Best First Search, I took a median of all the data of distance given between two cities and used the cities with distance less than the median as directly connected and others as connected using these cities. This helped in making heuristics clear and more resembling real life.
Eg:- There is not a direct road from Delhi to Bombay which would not involve any other city. Further, driving directly that long path doesn't make much sense as you have to pass through cities that lay between. So, giving an idea of those cities makes heuristics more practical.

Code

The code of the program is in a file 'assn. pl' attached along with this file. To run the program, follow these steps:-

1. Consult the file 'assn.pl'.
2. It will ask to choose the preferred method for finding a path.
3. You can enter the desired method.
4. It will ask for start city and end city.
5. After entering the details, the code will print the path and total distance as output.

It will become more clear with the below snippets and running code screenshots.

The code starts with the representation of the distance between cities. I used a python script to read it. I represented it in form *distance(start city, end city, magnitude)*.

```
%Rules to know distance between two citi
%Go to line 1930
distance(agartala, ahmedabad, 3305).
distance(ahmedabad, agartala, 3305).
distance(agartala, bangalore, 3824).
distance(bangalore, agartala, 3824).
```

In this way, I have expressed the data given in the CSV file.

Next, I have expressed connected cities in form of rules. I used the idea of Array lists which we generally use in Java to represent graphs. I took the name of the city as the first element and the list as the next.

```
%Rules for Depth first search to find connected cities.
list(agartala, [ahmedabad,bangalore,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune]).
list(agra, [ahmedabad,bangalore,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune]).
list(ahmedabad,[bangalore,bhubaneshwar,bombay,calcutta,calicut,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune,agartala,
list(allahabad,[ahmedabad,bangalore,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune]).
list(amritsar,[ahmedabad,bangalore,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune]).
list(asansol,[ahmedabad,bangalore,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune]).
list(bangalore,[ahmedabad,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune,agartala,agra,all
list(baroda,[ahmedabad,bangalore,bhubaneshwar,bombay,calcutta,chandigarh,cochin,delhi,hyderabad,indore,jaipur,kanpur,lucknow,madras,nagpur,nasik,panjim,patna,pondicherry,pune]).
```

This is data representation for the DFS.
In the below snippet, this part of the code will be used for taking inputs from the user and directing it to the correct searching paradigm, and give the required outputs.

```
go():-
    write('Hi! Welcome to Road Distance finder application.'),
    write('\n'),
    write('Which method would you like to choose:-'),
    write('\n'),
    write('1. Depth First Search \n'),
    write('2. Best First Search \n'),
    read(Input),
    write('\n'),
    write('Enter the start city \n'),
    read(Input1),
    write('Enter the destination city \n'),
    read(Input2),
    method(Input,Input1,Input2).

method(1,Input1,Input2):-
    dfs(Input1,Input2).

method(2,Input1,Input2):-
    bestFirst(Input1,Input2).
```

In the next snippet, this part will start the dfs code. The dfs function calls a helper function. The helper function checks if the destination is reached. In this case, it prints the current stack and total distance. Else, it finds the connected nodes to the current city and adds all the unvisited cities in the stack.

```
% Function to call for DFS
dfs(Src,Dest):-
    helper(Src,Dest,[],[]).

%helper function to check for destination. If reached then it prints the stack.
helper(Src,Src,Stack,Vis):-
    remove_duplicates(Stack,New),
    reverse(New,New1,[]),
    print1(New1),
    write(Src),
    write('\n'),
    toList(Src,Source),
    append(New1,Source,New2),
    disCalc(New1,A).

helper(Src,Dest,Stack,Vis):-
    toList(Src,Source),
    append(Source,Stack,New),
    append(Source,Vis,Visited),
    list(Src,Nodes),
    looper(Src,Nodes,Dest,Visited,New).
```

The helper calls a looper function which expands the first node present in the stack and sends it to a checker function. The checker checks if it is visited earlier or not. If it was visited earlier, it doesn't expand it and move to the next node. Else, it will expand that node and call the looper function for it. If the current node doesn't lead to the destination. The stack will pop that node and try it with the new peak element. In this way, it backtracks and finds the solution.

```
%recursive function for DFS
looper(Src,Nodes,Dest,Visited,Stack):-
    Nodes=[H|T],
    checker(Src,H,T,Dest,Visited,Stack).

%check if a city is already visited
checker(Src,Head,T,Dest,Visited,Stack):-
    \+ member(Head,Visited),
    helper(Head,Dest,Stack,Visited).

%if not add it in visited and stack
checker(Src,Head,T,Dest,Visited,Stack):-
    member(Head,Visited),
    looper(Src,T,Dest,Visited,Stack).

%When array for current city is empty it calls for next element in stack
checker(Src,Head,[],Dest,Visited,Stack):-
    Stack = [H|T],
    T = [Main|Tail],
    helper(Main,Dest,Visited,T).
```

The next function is to remove duplicates from a list. It is used to prevent any city from adding twice to the stack or visited list.

```
%Function to remove duplicates in a list
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
    member(Head, Tail), !,
    remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
    remove_duplicates(Tail, Result).
```

The next snippet is to print the list recursively. It stops when all the elements are printed.
The second function is to reverse the list as the stack contains the latest element at the top. We need to reverse the list to print the answer in the correct order.

```
% Function to print the list
print1(X):-
    X=[H|T],
    format('~w -> ',[H]),
    check(T).

check([]):- !.

check(T):-
    print1(T).

%Function to reverse the list
reverse([],Z,Z).

reverse([H|T],Z,Acc) :- reverse(T,Z,[H|Acc]).
```

The next part of the code is a representation of rules for the best-first search. It has city and list of cities directly connected to that city.

```
% Best First Search starts

%Heuristics for Best First Search. All cities with distance less than median of all distances
list1(agartala,[bhubaneshwar,calcutta,delhi,jaipur,kanpur,lucknow,nagpur,patna]).
list1(agra,[ahmedabad,chandigarh,delhi,indore,jaipur,kanpur,lucknow,nagpur,patna]).
list1(ahmedabad,[agra,baroda,bhopal,bombay,delhi,indore,jaipur,nagpur,nasik,pune,surat]).
list1(allahabad,[calcutta,chandigarh,delhi,indore,jaipur,kanpur,lucknow,nagpur,patna]).
list1(amritsar,[ahmedabad,chandigarh,delhi,indore,jaipur,kanpur,lucknow,nagpur,patna]).
list1(asansol,[bhubaneshwar,calcutta,delhi,indore,jaipur,kanpur,lucknow,nagpur,patna]).
```

The next part of the code is the starting point for the best-first search. It calls a driver function which is same as the helper function in case of DFS. The driver2 function calls the sort and calls the driver function recursively with the sorted list according to distance from the destination.

```
% Function to call Best First Search
bestFirst(Src,Dest):-
    driver(Src,Dest,[],[]).

%driver function
driver(Src,Src,Open,Closed):-
    remove_duplicates(Closed,New),
    reverse(New,New1,[]),
    print1(New1),
    write(Src),
    write('\n'),
    toList(Src,Source),
    append(New1,Source,New2),
    disCalc(New2,A).

driver(Src,Dest,Open,Closed):-
    list1(Src,Nodes),
    append(Nodes,Open,New1),
    toList(Src,Done),
    append(Done,Closed,Close),
    sort1(New1,Dest,Sorted),
    driver2(Src,Dest,New1,Close,Sorted).

%This returns a sorted array and calls the driver function recursively
driver2(Src,Dest,Open,Closed,Sorted):-
    Sorted=[H|T],
    driver(H,Dest,Open,Closed).
```

Now the sort1 function starts the sorting of the open list according to the distance from the destination. It calls a start method which further calls the sortdis method that finds the distance list of cities present in the open list. When it finds all the distance has been found. It sorts the distance and arranges cities in that particular order. The sorted list is returned to the driver and it checks whether the destination is found in the first position or not.

```
%Sorts cities according to distance from destination.
sort1(New1,Dest,Sorted):-
    start(New1,Dest,[],New1,Sorted).

start([],Dest,List,Main,Sorted):-
    sort(List,List1),
    sortDis(List1,List1,Dest,[],Sorted).

%Helper functions for sorting
start(New1,Dest,List,Main,Sorted):-
    New1=[H|T],
    distance(H,Dest,K),
    toList(K,Dist),
    append(List,Dist,New),
    start(T,Dest,New,Main,Sorted).

start(New1,Dest,List,Main,Sorted):-
    New1=[H|T],
    start(T,Dest,List,Main,Sorted).

sortDis(List,[],Dest,Temp,Temp):-!.

sortDis(List,Main,Dest,Temp,Sorted):-
    Main=[H|T],
    distance(A,Dest,H),
    toList(A,Node),
    append(Temp,Node,Temp1),
    sortDis(List,T,Dest,Temp1,Sorted).
```

The last function in the program is used to calculate the total distance with respect to the final list of cities. It recursively finds it between two consecutive cities.

```
% Distance calculator after finding the path
disCalc(List,Dis):-
    List = [H|T],
    Dis is 0,
    calc1(H,T,Dis).

calc1(H,[],Dis):-
    write('Distance you will have to cover is '),
    write(Dis),
    write(' km.').

calc1(Head,Tail,Dis):-
    Tail = [H|T],
    distance(Head,H,A),
    B is  Dis+A,
    calc1(H,T,B).

% Code ends.
```

Working of code

1. First, consult the given prolog file.

```
shivansh@shivansh-Vostro-3578:~/Prolog$ prolog
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

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

?- [assn].
Warning: /home/shivansh/Prolog/assn.pl:2012:
Warning:    Singleton variables: [Vis,New2,A]
Warning: /home/shivansh/Prolog/assn.pl:2035:
Warning:    Singleton variables: [Src,T]
Warning: /home/shivansh/Prolog/assn.pl:2045:
Warning:    Singleton variables: [Src,Head,H,Tail]
Warning: /home/shivansh/Prolog/assn.pl:2136:
Warning:    Singleton variables: [Open,A]
Warning: /home/shivansh/Prolog/assn.pl:2155:
Warning:    Singleton variables: [Src,T]
Warning: /home/shivansh/Prolog/assn.pl:2163:
Warning:    Singleton variables: [Main]
Warning: /home/shivansh/Prolog/assn.pl:2175:
Warning:    Singleton variables: [H]
Warning: /home/shivansh/Prolog/assn.pl:2179:
Warning:    Singleton variables: [List,Dest]
Warning: /home/shivansh/Prolog/assn.pl:2194:
Warning:    Singleton variables: [H]
true.
```

It will give some warnings, just ignore them as it is not effecting the working of program in any way.

2. Enter go(). and press enter. It will start the program and you will be asked to choose the algorithm to find the path.

```
?- go().
Hi! Welcome to Road Distance finder application.
Which method would you like to choose:-
1. Depth First Search
2. Best First Search
|:
```

3. I will first demonstrate the DFS part. I will the first show for two directly connected cities let's say Delhi and Bombay.

```
Enter the start city
|: delhi
|: .
Enter the destination city
|: bombay.
delhi -> ahmedabad -> bangalore -> bhubaneshwar -> bombay
Distance you will have to cover is 3939 km.
true .
```

You can observe that it demonstrates the path from Delhi to Bombay as found by DFS. Now, getting it using best-first search.

4. I will select the second option now after the go method. It will provide me with the respected solution.

```
1. Depth First Search
2. Best First Search
|: 2.

Enter the start city
|: delhi.
Enter the destination city
|: bombay.
delhi -> ahmedabad -> bombay
Distance you will have to cover is 1463 km.
true .
```

You can observe now that the solution is better than one found by DFS as heuristics were used and the sorting function helped with a better solution. Now, applying DFS for two not directly connected cities.

5. I will select the DFS option again and show it for two not directly connected cities let's say Agartala and Hubli.

```
?- go().
Hi! Welcome to Road Distance finder application.
Which method would you like to choose:-
1. Depth First Search
2. Best First Search
|: 1.

Enter the start city
|: agartala.
Enter the destination city
|: hubli.
agartala -> ahmedabad -> bangalore -> bhubaneshwar -> bombay -> calcutta -> chandigarh -> cochin -> delh
i -> hyderabad -> indore -> jaipur -> kanpur -> lucknow -> madras -> nagpur -> nasik -> panjim -> patna
-> pondicherry -> pune -> agra -> allahabad -> amritsar -> asansol -> baroda -> bhopal -> calicut -> coi
mbatore -> gwalior -> hubli
Distance you will have to cover is 31142 km.
true .
```

6. Now, finding the path for the same cities using best-first search.

```
?- go().
Hi! Welcome to Road Distance finder application.
Which method would you like to choose:-
1. Depth First Search
2. Best First Search
|: 2.

Enter the start city
|: agartala.
Enter the destination city
|: hubli.
agartala -> bhubaneshwar -> hyderabad -> hubli
Distance you will have to cover is 3816 km.
true .
```

The result is far better than the one provided by DFS.
So, the code works perfectly in both the cases.