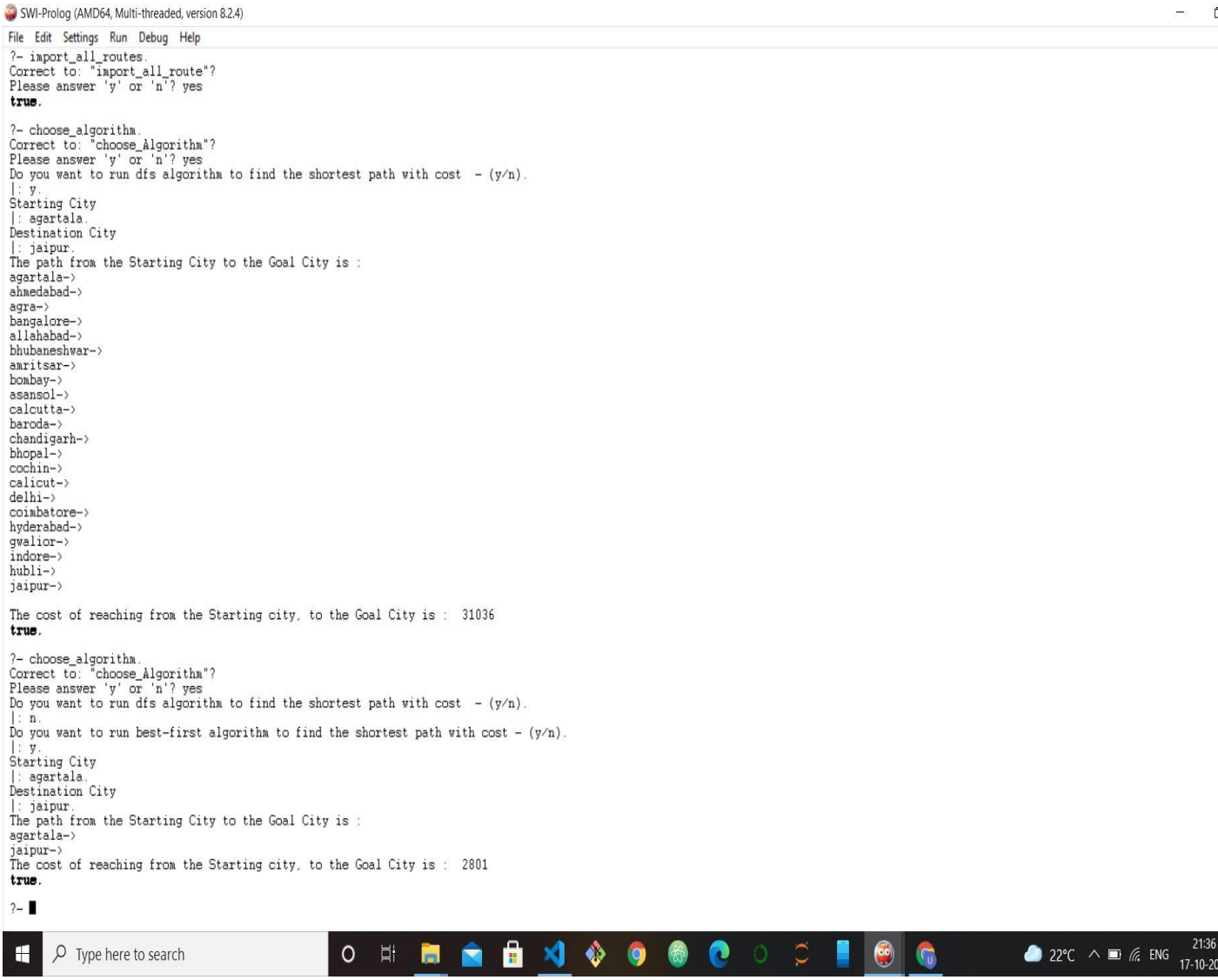
AI Assignment 2 :

1. Program Screenshot on both the dfs and best first algorithm .

Screenshot from the prolog program :



2.Prolog Program Code :

/*For running the program you have to import the csv file called by road_distance.csv

By calling this way:

import_all_routes.

And then just run the program by typing

choose_algorithm. */

/*

importing the file distance_route.csv in order to make the facts of distance between 2 cities */

```
import_all_route :-
        csv_read_file('road_distance.csv' ,R, [functor(give_Distance), arity(3)]),
        maplist(assert, R).

% User choose the Dfs algorithm to proceed further.
choose_Algorithm :-
            write("Do you want to run dfs algorithm to find the shortest path with cost  - (y/n)."),
            nl,
            read(Ans),
            Ans=y,
```

```
        !, % cut is used here so cannot backtrack from here
        write("Starting City"),nl,
        read(Start_city),
        write("Destination City"),nl,
        read(End_city),
       solve_With_Dfs(Start_city, End_city).

% Best first search initialization.
choose_Algorithm :-
        write("Do you want to run best-first algorithm to find the shortest path with cost - (y/n)."),
        nl,
        read(Ans),
        Ans=y,
        !, % cut is used here so cannot backtrack from here
        write("Starting City"),nl,
        read(Start_city),
        write("Destination City"),nl,
        read(End_city),
         solve_With_BestFirst(Start_city, End_city).
```

```
% solving with the depth first search algorithm

solve_With_Dfs(Start_city,End_city) :-
        write("The path from the Starting City to the Goal City is : "), nl,
        depthfirstSearch([], Start_city, End_city,Solution),
        reverse(Solution,Path),
        print_Path_dfs(Path),nl,
        write("The cost of reaching from the Starting city, to the Goal City is :  "),
        find_Cost(Path,Best_cost),
        write(Best_cost), nl ,!.

% writing the logic for depth first search

% when the Start_city mathces with the End_city.

depthfirstSearch(Path, Node, Node, [Node|Path]).

% Recusrion and backtracking for the dfs

depthfirstSearch(Path, Start_city, End_city, Sol) :-
```

```
        check_Edge(Start_city, New_city),
        dif(End_city, Start_city),
        not(member_Checked(New_city, Path)),
        depthfirstSearch([Start_city|Path],New_city, End_city, Sol).

/*for printing the path */

print_Path_dfs([]).

print_Path_dfs([Current_city|Remaining]) :-
         write(Current_city),
         write("->"),nl,
         print_Path_dfs(Remaining).

% finding the cost to travel

%cases when there is 1 city or empty case

find_Cost([],0). %empty
find_Cost([_],0). %single

%using our knowlegdge base give_Distance we write a recursive
logic
```

```
find_Cost([Start,Next|Rem],Cost):-
                find_Cost([Next|Rem],C1),
                give_Distance(Start,Next,C2),
                Cost is C1 + C2.

%Checking the member

member_Checked(Head, [Next|Rem]):-
                 dif(Head,Next),
                 member_Checked(Head,Rem).



member_Checked(Head, [Head|_]).

% checking the edge is it connecnted or not

check_Edge(A,B) :- give_Distance(A,B,_).

/* let us write the logic for the best first search with heuristic value */
```

```
solve_With_BestFirst(Start_city,End_city):-
        generate_heuristic_Val(Start_city,End_city, Hval),
        vertex(Start_city, nil, Hval,Current_city),
        Finished=[],
        priority_Queue_Insertion(Current_city, [] ,Started ),
        best_first_Search(Started, Finished, End_city).

% generating heuristic value
% We have used the simple gereration of heuristic value using findall
% and min_list inbuilt functions
% findall will fill the bag of starting node w.r.t cost of all the nodes
% end with the End_city destination, then by using the min_list we are
% taking the minimum straight distance with the destination , can be say
% it as euclidean distance.

generate_heuristic_Val(Start,_,End):-
findall(Y,give_Distance(Start,_,Y),Bag), min_list(Bag,End).

%vertex logic
```

```
vertex(State_node, Parent_node, Hval ,
[State_node,Parent_node,Hval]).


%logic to use the insert_priority_queue



priority_Queue_Insertion(State_node, [First|Rem],
[First|Rem_new]):-priority_Queue_Insertion(State_node, Rem,
Rem_new).

priority_Queue_Insertion(State_node,[First|Rem],[State_node,First|Re
m]):-insertion_order(State_node,First).

priority_Queue_Insertion(State_node, [], [State_node]).


% logic of insertion


insertion_order([_,_,V1], [_,_,V2]) :- V1 =< V2.


% writing the logic of best first search







best_first_Search(Started,_,_) :-

                        Started = [],
```

```
                write("Cities are not connected").

best_first_Search(Started, Finished, End_city) :-
delete_priority_queue(Curr_vertex, Started,_),
                        vertex(State, _, _,Curr_vertex),
                        State = End_city,
                        write("The path from the Starting City
to the Goal City is : "),nl,

show_path(Curr_vertex,Finished,Path),
                        write("The cost of reaching from the
Starting city, to the Goal City is :  "),
                        find_Cost(Path,Best_cost),
                        write(Best_cost), nl.

best_first_Search(Started, Finished, End_city):-
                       delete_priority_queue(Curr_vertex,
Started, Rest_of_started),

findall(Child,list_new_cities(Curr_vertex,Started, Finished, Child,
End_city), Childvertices),

insert_list_to_queue(Childvertices,Rest_of_started,New_started),
                        add_to_Finsihed(Curr_vertex,
Finished, New_finished),
```

```
                                    best_first_Search(New_started,
New_finished, End_city),!.

% logic for deleting the priority queue

delete_priority_queue(Begin, [Begin|Rem], Rem).

%logic for showing the path of the best first search

show_path(Next_vertex, _,[State]):-
                        vertex(State, nil,_, Next_vertex),
                        write(State),
                        write("->"),nl.

show_path(Next_vertex, Finished, [State|Rem]):-
                        vertex(State, Curr_vertex, _, Next_vertex),
```

```
        vertex(Curr_vertex, _,_, Vertex_record),
        member_Checked(Vertex_record, Finished),
        show_path(Vertex_record, Finished, Rem),
        write(State),
        write("->"),nl.
```

% inserting the list to queue

```
insert_list_to_queue([State | Tail], Ros, New_started) :-
priority_Queue_Insertion(State, Ros, Ros2),
insert_list_to_queue(Tail, Ros2, New_started).

insert_list_to_queue([], Ros, Ros).
```

%tracking the finished vertices

```
add_to_Finsihed(Head1, [Head|T1], [Head|T2]):-
add_to_Finsihed(Head1, T1,T2).

add_to_Finsihed(Head, [Head|T], [Head|T]).

add_to_Finsihed(Head, [], [Head]).
```

% keep tracking of the vertices to the current city, curr vertex

```
list_new_cities(Curr_vertex, Started, Finished, Child, End_city):-
                                        vertex(State, _,_,Curr_vertex),
                                        check_Edge(State,Next_vertex),
                                        vertex(Next_vertex, _, _,Check),

not(member_pq_Checked(Check,Started)),
                                        not(member_Checked(Check,
Finished)),

generate_heuristic_Val(Next_vertex, End_city,Hval),
                                        vertex(Next_vertex, State, Hval,
Child).



% priority queue member checking
member_pq_Checked(A,B) :- member(A,B).
```

3.References used :

reverse

https://www.swiprolog.org/pldoc/man?predicate=reverse/2

min_list

https://www.swiprolog.org/pldoc/man?predicate=min_list/2

findall

https://www.swi-prolog.org/pldoc/man?predicate=findall/3