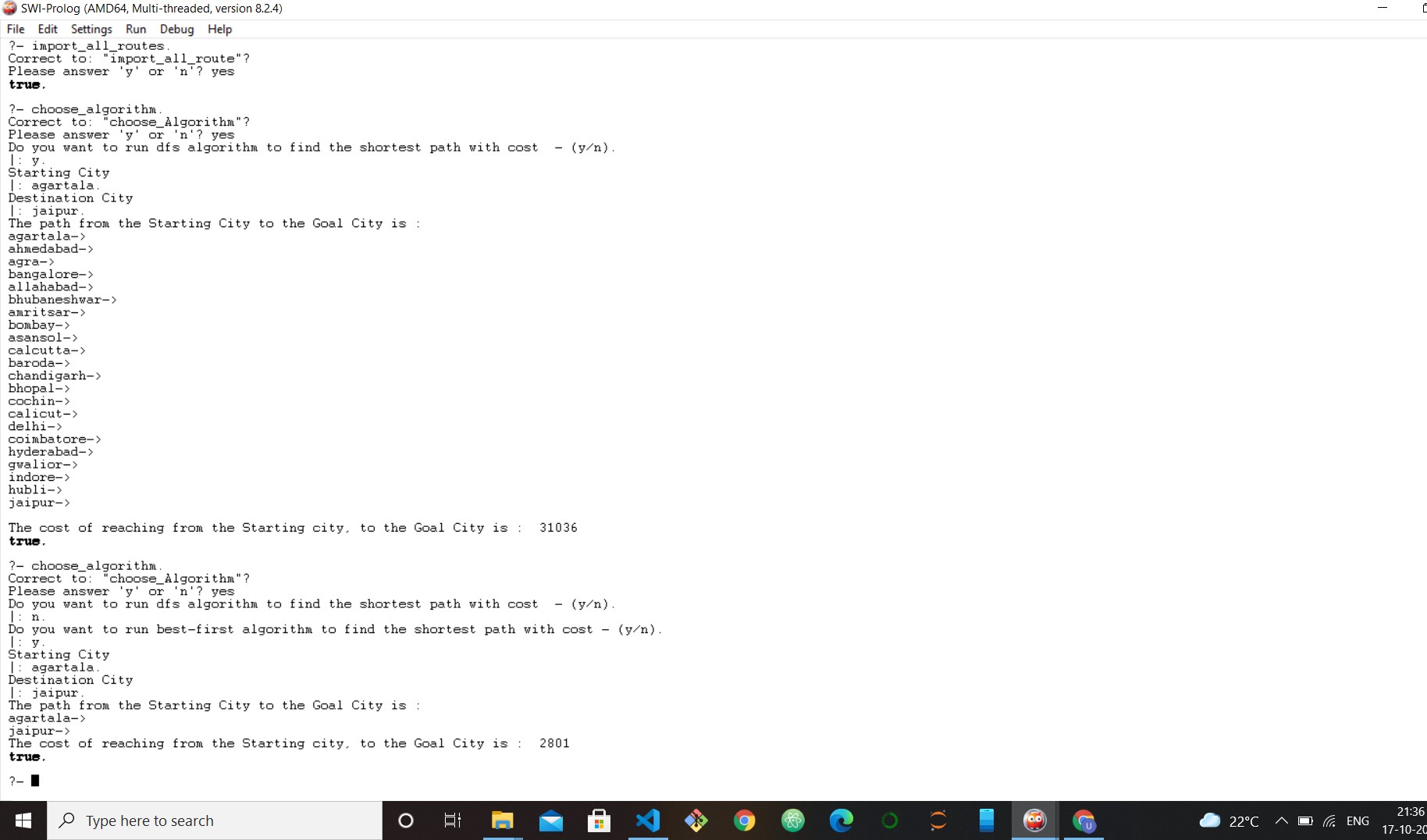
# AI Assignment 2 :

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

Screenshot from the prolog program :

1. 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).