CSE 643: Assignment 2

Abhimanyu Gupta - 2019226

Brief about working:

The prolog program takes the algorithm to be used for path-finding as input from the user returns the path if exists or displays no path exists message.

Steps to run the program:

1. Load the program using consult or [code] in prolog terminal.
2. Run the path-finding program using the find_path command.
3. Supply the source city, destination city and algorithm to be used as input.
4. At last, the program would display the path between two cities if it exists along with the distance. In case of no path between the two cities, a message mentioning the same would be displayed.
5. Heuristics are made by calculating the shortest distances between all cities through a python script.

Working Examples:

1)

```
?- cd('d:/Codes/CSE643_AI/A2').
true.

?- [code].
Warning: d:/codes/cse643_ai/a2/code.pl:51:
Warning:    Singleton variables: [Title]
Warning: d:/codes/cse643_ai/a2/code.pl:65:
Warning:    Singleton variables: [Header,Functor,Arity]
true.
```

```
?- find_path.
Welcome to Path Finding System
Enter the source city
|: 'Delhi'.
Enter the destination city
|: 'Delhi'.
Select the algorithm to be used for path finding (depth/best)
|: 'depth'.
Distance betwee Delhi and Delhi is 0 units and the path joining them is
true .
```

```
?- find_path.
Welcome to Path Finding System
Enter the source city
|: 'Surat'.
Enter the destination city
|: 'Agra'.
Select the algorithm to be used for path finding (depth/best)
|: 'depth'.
Distance betwee Surat and Agra is 28100 units and the path joining them is Surat -> Ahmedabad -> Bangalore -> Bhubaneshwar -> Bombay
 -> Calcutta -> Chandigarh -> Cochin -> Delhi -> Hyderabad -> Indore -> Jaipur -> Kanpur -> Lucknow -> Madras -> Nagpur -> Nasik ->
Panjim -> Patna -> Pondicherry -> Pune -> Agra
true .
```

```
?- find_path.
Welcome to Path Finding System
Enter the source city
|: 'Agra'.
Enter the destination city
|: 'Surat'.
Select the algorithm to be used for path finding (depth/best)
|: 'best'.
Distance betwee Agra and Surat is 1267 units and the path joining them is Agra -> Nasik -> Surat
true .
```

Code Snippets:

```
% Abhimanyu Gupta - 2019226
% CSE643 - Artificial Intelligence - Assignment 2

:- [library(csv)].

:- dynamic connected_cities/3, heuristic/3.

find_path :-
    clear_all,
    convert_csv_to_facts('roaddistance.csv', connected_cities),
    convert_csv_to_facts('heuristic_roaddistance.csv', heuristic),

    write('Welcome to Path Finding System\n'),

    write('Enter the source city\n'),
    read(SourceCity),
    write('Enter the destination city\n'),
    read(DestinationCity),

    write('Select the algorithm to be used for path finding (depth/best)\n'),
    read(Algorithm),
    (
        \+is_valid_algorithm(Algorithm) -> write('Invalid algorithm!!, program exiting!\n'), fail;
        path(SourceCity, DestinationCity, Path, Distance, Algorithm) ->
        (
            format('Distance betwee ~w and ~w is ~w units', [SourceCity, DestinationCity, Distance]),
            write(' and the path joining them is '),
            show_path(Path)
        );
        write('Path does not exists')
    ),

    clear_all.
```

```
is_valid_algorithm('depth').
is_valid_algorithm('best').

clear_all :-
    retractall(connected_cities(_, _, _)),
    retractall(heuristic(_, _, _)).

show_path([]).
show_path([CurrentCity | NextCities]) :-
    format('~w', [CurrentCity]),
    isempty(NextCities);
    format(' -> '),
    show_path(NextCities).

isempty([]).

convert_csv_to_facts(CSVName, Functor) :-
    csv_read_file(CSVName, RawData, [functor(Functor)]),

    RawData = [Title | RestRows],
    RestRows = [Header | Rows],
    functor(Header, _, Arity),

    create_facts(Header, Rows, Functor, Arity),
    clean_facts.

clean_facts :-
    retractall(connected_cities(X, X, _)),
    retractall(connected_cities(_, _, '-')).

```

```
create_facts(Header, [], Functor, Arity).
create_facts(Header, [TopRow | RestRows], Functor, Arity) :-
    create_facts_from_row(Header, TopRow, Functor, Arity),
    create_facts(Header, RestRows, Functor, Arity).

create_facts_from_row(Header, Row, Functor, Arity) :-
    arg(1, Row, City1),
    create_facts_for_cells(2, Arity, City1, Row, Header, Functor).

create_facts_for_cells(CurIndex, LastIndex, City1, Row, Header, Functor) :-
    CurIndex > LastIndex;
    (
        arg(CurIndex, Header, City2),
        arg(CurIndex, Row, Distance),
        (
            Functor = connected_cities -> assert(connected_cities(City1, City2, Distance));
            Functor = heuristic -> assert(heuristic(City1, City2, Distance))
        ),
        NextIndex is CurIndex + 1,
        create_facts_for_cells(NextIndex, LastIndex, City1, Row, Header, Functor)
    ).

connected(City1, City2, Distance) :-
    connected_cities(City1, City2, Distance);
    connected_cities(City2, City1, Distance).
```

```
path(SourceCity, DestinationCity, Path, Distance, Algorithm) :-
    SourceCity = DestinationCity ->
    (
        Path = [],
        Distance = 0
    );
    (Algorithm = 'depth' -> path_depth(SourceCity, DestinationCity, Path, Distance));
    (Algorithm = 'best' -> path_best(SourceCity, DestinationCity, Path, Distance)).

% START: Depth First Search

path_depth(SourceCity, DestinationCity, Path, Distance) :-
    move_depth(SourceCity, DestinationCity, [SourceCity], PathStack, Distance),
    reverse(PathStack, Path).

move_depth(DestinationCity, DestinationCity, CurrentPath, CurrentPath, 0).
move_depth(SourceCity, DestinationCity, CurrentPath, PathStack, Distance) :-
    connected(SourceCity, NextCity, ConnectingDistance),
    \+member(NextCity, CurrentPath),
    move_depth(NextCity, DestinationCity, [NextCity | CurrentPath], PathStack, FurtherDistance),
    Distance is ConnectingDistance + FurtherDistance,
    !.

% END: Depth First Search
```

```
% START: Best First Search

path_best(SourceCity, DestinationCity, Path, Distance) :-
    move_best(SourceCity, DestinationCity, [], [SourceCity], PathFollowed, Distance),
    reverse(PathFollowed, Path).

lesser_heuristic(R, [City1 | Heuristic1], [City2 | Heuristic2]) :-
    Heuristic1 > Heuristic2 -> R = (>) ;
    Heuristic1 < Heuristic2 -> R = (<) ;
    City1 = City2 -> R = (=);
    R = (<) .

part_of(NextCity, [[NextCity |_] | _]).
part_of(NextCity, [[_ |_] | RestCities]) :-
    part_of(NextCity, RestCities).

move_best(DestinationCity, DestinationCity, _, VisitedCities, VisitedCities, 0).
```

```
move_best(SourceCity, DestinationCity, CurrentCities, VisitedCities, PathStack, Distance) :-
    findall(

        [NextCity, Heuristic],
        (
            connected(SourceCity, NextCity, _),
            \+part_of(NextCity, CurrentCities),
            \+member(NextCity, VisitedCities),
            heuristic(NextCity, DestinationCity, Heuristic)
        ),
        NextCities
    ),

    append(CurrentCities, NextCities, UpdatedCurrentCities),
    predsort(lesser_heuristic, UpdatedCurrentCities, SortedCurrentCities),
    [[NextCity | _] | NextCurrentCities] = SortedCurrentCities,

    move_best(NextCity, DestinationCity, NextCurrentCities, [NextCity | VisitedCities], PathStack, FurtherDistance),
    connected(SourceCity, NextCity, ConnectingDistance),
    Distance is ConnectingDistance + FurtherDistance,
    !.

% END: Best First Search
```