# CA208 Assignment Explanation Doc

Name: Diana Williams Oshun

Student Number: 23101961

Date: 07/04/24

The assignment specified to make a predicate called journey that takes 3 parameters, a start location, an end location, and the desired modes of transport to be used for the journey.

To start off the program need to have some predefined routes so that I can find the fastest path. The route fact takes 4 parameters, a start location, an end location, the distance between the two locations and the modes of transport you can use to travel between those two locations.

```
/*FACTS*/
route(dublin, cork, 200, 'fct').
route(cork, dublin, 200, 'fct').
route(cork, corkAirport, 20, 'fc').
route(corkAirport, cork, 25, 'fc').
route(dublin, dublinAirport, 10, 'fc').
route(dublinAirport, dublin, 20, 'fc').
route(dublinAirport, corkAirport, 225, 'p').
route(corkAirport, dublinAirport, 225, 'p').
route(dublin, wicklow, 100, 'fct').
route(dublinAirport, ukAirport, 500, 'p').
route(ukAirport, dublinAirport, 500, 'p').
```

To start off the user will call upon the journey predicate, the user inputs the start, finish and modes of transport they want to use. With that my program then calls upon the Dijkstra predicate. The Dijkstra predicate calculates the shortest path between the start and finish and keep a list K of all the nodes visited in the shortest path. K is then sent to the path predicate. This predicate builds the path between the start and finish location and uses the K list to fill in the exact locations that get travelled in between. It also makes a list called Modes that holds all the modes of transport used during that path. The Modes list is then reverse so that they are in the correct order and finally all this information is outputted to the user. The user gets the fastest path and the fastest modes of transport needed to get from the start to the finish.

```
% Finds the fastest route and modes of transport between two locations and
prints this out to the user
journey(Source, D, M) :- dijkstra([Source, null, 0, _], D, M, [], [], K),
path(Source, D, K, Path, Modes), reverse(Modes, ReverseModes), write("From "),
write(Source), write(" to "), write(D), write(" the fastest route available
is: "), nl, writeln(Path), write("You need to use these modes of transport:
"), nl, writeln(ReverseModes).
```

The dijkstra predicate is initially called from the journey predicate with parameters [Source, null, 0, _] representing the starting node with no parent, zero time, and no mode. It also gets the parameters Target (destination), Modes (modes of transport), List (initial list of nodes with starting node), Complete (empty list representing visited nodes), and Helper (to store the final path). It traverses through the List and checks if the current node is not the Target node. If it is, it reverses the Complete list and stores it in Helper, effectively ending the traversal. It then uses the travel predicate to find possible surrounding nodes reachable from the current node using the specified modes of transport in Modes. It makes sure that each surrounding node is not already in the Complete list. It updates the travel times and modes for the surrounding nodes based on the current node's information using the update predicate. It adds the updated nodes to the List to continue traversal. Finally, after updating the nodes, Dijkstra predicate recursively calls itself with the next node from the updated list and the updated Complete list. When the target node is reached, the recursion ends, and the final Helper list contains the shortest path from Source to Target. The Helper list is returned to the journey predicate, which then uses it to construct the final path using path predicate and outputs the results.

```
% Uses Dijkstra algorithm to find the fastest path between two locations using
the fastest modes of transport for that route
dijkstra(Node, Target, Modes, List, Complete, Helper):- (Node =
[NodeN,_,_,_]), (NodeN \= Target), travel(Modes, NodeN, Surrounding), \+
hasNode(Complete, NodeN), update(Node, Modes, Surrounding, Complete, List,
List1), !, (List1 = [Next|List2]), dijkstra(Next, Target, Modes, List2,
[Node|Complete], Helper).
dijkstra(Node, Target, _, _, Complete, Helper):- (Node = [NodeN, _, _, _]),
(NodeN = Target), reverse([Node|Complete], Helper).
```

The travel predicate serves as an interface predicate for finding a route between two locations using the specified modes of transport.

```
% Using the modes of transport between two locations, it finds a route and
returns the route in K
travel(Modes, Y, K) :- travelHelper(Modes, Y, [], K).
```

The travelHelper predicate is where all the legwork happens. It uses recurtion and backtracking to find all the possible routes until a valid route is found to Y, then it checks if node Z can reach Y using the specified modes of transport and makes sure that Z hasn't been visited before checking if it's a member of X. The intersection of RouteList and TravelList determines if a valid route exists with the specified modes. Then it recursively explores the surrounding nodes until a valid route to the destination is found or it exhausts all the possible paths.

```
% This is a helper predicate for travel, so that I can recursively find
successful routes between the locations
travelHelper(Modes, Y, X, Helper) :- route(Y, Z, _, Travel),atom_chars(Travel,
TravelList), atom_chars(Modes, RouteList), intersection(RouteList, TravelList,
[_|_]), \+ member(Z, X), !, travelHelper(Modes, Y, [Z|X], Helper).
travelHelper(_, _, X, X).
```

The update predicate is called during the execution of the Dijkstra predicate. Its responsible for updating the travel times and modes of the surrounding nodes based on the parent node's info. It recursively processes each node in the node list. For every node it calculates the time and mode of transportation from the parent node using the time predicate. It also calculates the total time it takes from the start node to the parent node. By doing this it can compare the new total time with the exiting time for each of the nodes. And if the new time is faster, it updates the time for that node. If the node previously didn't have a time, it inserts a new time for that node. This continues until all the surrounding nodes of the parent node are processed.

```
% Updates the list of nodes with new travel times and modes based on the
current node.
update(_, _, [], _, List, List).
update(PNode, Modes, [N|Nodes], Complete, In, Out):-hasNode(Complete, N), !,
update(PNode, Modes, Nodes, Complete, In, Out).
update(PNode, Modes, [N|Nodes], Complete, In, Out):- (PNode = [PName, _,
PTime, _]), time( PName, N, Modes, NodeT, M), (NewT is PTime + NodeT),
execute(Modes, PName, N, NewT, M, In, List1), update(PNode, Modes, Nodes,
Complete, List1, Out).
```

The hasNode predicate simply searches t see if the specified node is in the node list, if the nide isn't the head, it checks if the node is in the tail of the list.

```
% Checks if the node is there in a list of nodes
hasNode([[Node,_,_,_]|_],Node) :- !.
hasNode([_|T],Node) :- hasNode(T,Node).
```

The time predicate isn't called upon during the initial execution of the Dijkstra predicate, however it is needed for the predicate to function properly. It calculates the travel time and determines the mode of transport used for the journey from Start to Finish based on the specified Modes. It uses the route predicate to get info about the routes needed including the distance and modes. It checks if the modes intersect with the modes specified when the journey function was called. If I was it, then calculates the time using the speed predicate and distance between the two locations. The speed predicate returns the speed of the specified mode. Only the fastest mode of transport from the intersection of modes is used to calculate the time.

```
% Calculates how long it takes to travel between the locations using the
fastest modes of transport for that route
time(Start, Finish, Modes, Time, M) :- atom_chars(Modes, RouteList),
route(Start, Finish, Distance, Travel), atom_chars(Travel, TravelList),
intersection(RouteList, TravelList, Intersect), fastest(Intersect, F),
speed(F, Speed), (Time is Distance / Speed), (M = F).
```

The speed predicate gives a value to each of the modes of transport. p = plane, t = train, c = car and f = foot. And the values beside them is the speed in km/h each of these modes of transport travel at e.g. 500km/h for a plane.

```
% Initialising all the different modes of transport with their corresponding
speed
speed(p, 500).
speed(t, 100).
speed(c, 80).
speed(f, 5).
```

The fastest predicate finds the fastest mode of transport from a list of modes. Its another interface predicate and it calls upon the fastestHelper predicate to compare the speed of the modes.

```
% Find the fastest mode of transport from a list of modes
fastest([M|Modes], F):- fastestHelper(Modes, M, F).
```

The fastestHelper predicate recursively compares speeds among modes to find the overall fastest mode. It calls the fastestMode predicate to compare the speed of the current mode M with the fastest mode Temp1. If M is faster than Temp1, it updates Temp1 with M as the new fastest mode. The process continues until all modes are compared, resulting in F containing the fastest mode overall.

```
% This is a helper predicate for fastest, it helps find the overall fastest
mode of transport
fastestHelper([M|Modes], Temp1, F):- fastestMode(M, Temp1, Temp2),
fastestHelper(Modes, Temp2, F).
fastestHelper([], F, F).
```

The fastestMode predicate compares speeds between two modes M1 and M2. It uses the speed predicate to get the speed of each mode and compares them. If M1 is faster than M2, it unifies M1 with the result, making M1 the faster mode. If M2 is faster or speeds are equal, M1 remains unchanged as the faster mode.

```
% Determines the faster mode of transport out of two modes
fastestMode(M1,M2,M1):- speed(M1,S1), speed(M2,S2), (S1 > S2), !.
fastestMode(_,M,M).
```

The path predicate constructs the path from the Start location to the Finish location using the information in the Nodes list. It uses the parent-child relationships stored in the Nodes list to trace back from the Finish location node to the Start location node. It recursively traces the path backward from Finish to Start by finding the parent of each node until it reaches the Start node. During this it also collects the fastest modes of transport used along the path.

```
% Makes a path between two locations based on the parent-child relationship
nodes
path(Start, Finish, Nodes, Path, Modes) :- makePath(Start, Finish, Nodes,
RPath, Modes), reverse(RPath, Path).
makePath(N, N, _, [N], []).
makePath(Start, Finish, Nodes, [Finish|Path], [M|Modes]):- parent(Finish,
Nodes, Prev, M), makePath(Start, Prev, Nodes, Path, Modes).
```

The parent predicate searches the list of nodes to find the input that matches to the given Node. When it finds the matching input, it assigns P with the parent node and M with the mode of transport associated with the given node. It uses pattern matching to compare the Node with the first element of each input in the list of nodes. It recursively processes the list until it finds the correct input or exhausts every item in the list. If the given Node is not present in the list, it fails.

```
% Finds the parent node and mode of transport of a given node in the list of
nodes
parent(_, [], null, _).
parent(Node, [[Node, P, _, M]|_], P, M) :- !.
parent(Node, [_|T], P, M) :- parent(Node, T, P, M).
```

The execute predicate handles updates and modifications to the list of nodes during the execution of the Dijkstra predicate. It calls upon the timeList predicate, insert predicate and delete predicate and executes them if the conditions are correct.

```
% Executes updates and modifications to the list of nodes during dijkstra
predicate
execute( _, P, Node, NodeT, M, In, Out):- hasNode(In,Node), !,
timeList(Node,In,PrevT, _), ( NodeT < PrevT -> delete(Node,In,List1),
insert(Node,P,NodeT, M, List1, Out) ; Out = In ).
execute( _, P, Node, NodeT, M, In, Out):- insert(Node, P, NodeT, M, In, Out).
```

The delete predicate deletes a node from a list of nodes. It uses pattern matching to match the first element in the list and removes it to generate the updated list T.

```
% Deletes a node from a list of nodes
delete(Node, [[Node, _, _, _]|T], T) :- !.
delete(Node, [H1|T1], [H1|T2]) :- delete(Node, T1, T2).
```

The insert predicate inserts a node into a sorted list of nodes based on travel time. It recursively compares the travel time of Node with the travel time of the current node H in the list until it finds the correct position to insert Node. Once the correct position is found, it inserts Node into the list to generate the updated sorted list T2.

```
% Inserts a node into a sorted list of nodes based on travel time
insert(Node, P, Time, M, [H|T], [H|T2]) :- (H = [_, _, Time1, _]), (Time >
Time1), !, insert(Node, P, Time, M, T, T2).
insert(Node, P, Time, M, [H|T], [[Node, P, Time, M], H|T]) :- (H = [_, _,
Time1, _]), (Time =< Time1), !.
insert(Node, P, Time, M, [], [[Node, P, Time, M]]).
```

The timeList predicate searches the list of nodes to find the input that matches to the given Node. When it finds the matching input, it combines Time with the travel time and M with the mode of transport associated with the given Node. It uses pattern matching to compare the Node with the first element of each input in the list of nodes. It recursively processes the list

until it finds the correct entry or exhausts the list. If the given Node is not present in the list, it fails.

```
% Finds the time and mode of transport for a specific node in a list of nodes
timeList(Node, [[Node, _, Time, M]|_], Time, M):- !.
timeList(Node, [_|T], Time, M):- timeList(Node, T, Time, M).
timeList(_, [], 0, _).
```

Now that all the predicates have been explained an example of the usage for my journey function would be:

```
journey(dublin, cork, 'fctp').
journey(dublin, wicklow, 'fc').
journey(wicklow, ukAirport, 'ctp').
```

I declare that this is all my Diana Williams Oshun's own work.

I used online resources to help me make this project, my resources are:

1) CA208 Prolog notes
https://www.computing.dcu.ie/%7Edavids/courses/CA208/CA208_Prolog_2p.pdf

2) SWI-Prolog (from the main website where I downloaded the prolog program) https://lpn.swi-prolog.org/lpnpage.php?pageid=online

3) FreeCodeCamp article on Dijkstra's algorithm
https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction

4) Dijkstras Shortest Path Algorithm Explained | With Example | Graph Theory (YouTube Video). https://www.youtube.com/watch?v=bZkzH5x0SKU

Another outside resource is the route facts given in the assignment and the design that the journey predicate takes three arguments.