ECM2418 CA Report

Question 1:

Predicate **multiple_lines(S)** computes the stations **S** serving more than one line. The predicate finds every pair of non-equal lines and checks for the existence of a station on both lines. Also, a query to this predicate can return the same station as an answer multiple times.

Question 2:

Predicate **termini(L, S1, S2)** computes the two terminal stations **S1** and **S2** of the line **L**, where **S1** and **S2** are the respective first and last stations on line **L**. **non_termini(L, E)** verifies if a station is non-terminal by comparing **E** with the stop number of each station on the line **L**. If it can't find a stop number greater than **E**, it returns false. **termini(L, S1, S2)** checks that station **S1** is the first stop on the line and then finds which stop **S2** is at by checking, via negation of **non_termini**, that it is not possible to prove that **S2** is non-terminal.

Question 3:

Predicate **list_stops(L, List)** computes the list **List** of all the stations on the line **L**, with stations in the list ordered from the first to the last station on the line. This is achieved by using a modified version of the insertion sort algorithm which sorts by stop numbers for a given line, by taking an extra **Line** parameter. **list_stops(Line, List)** uses findall to create a list of all stops that exist on **Line** and unifies the result with **First_list.** Finally, it sorts this list by station order using the modified **isort**.

Question 4:

Predicate **path(S1,S2,Path)** computes a path from the station **S1** to the station **S2,** and stores the result in **Path**.

member_of_path(segment(Line, S1, S2), Path) checks if the **Line** of the segment has already been traversed by checking if it appears in an existing segment in **Path**. It is implemented to satisfy the requirement that computed paths should not suggest to take a line more than once. The predicate checks if the head of the list (first segment in the path) has already used that **Line** before, returning true if it has, otherwise recursively calling on the tail of the list so it can traverse the entire path.

Predicate **station_traversed(S, Path)** checks whether station **S** is traversed in one of the segments of the **Path.** It is a modified version of the station_traversed from the Appendix of the coursework. It guarantees that the code does not get stuck in a loop. The first modification was to remove the cut operator at the end of the first rule, to allow for backtracking. The other modification was changing the condition 'N =< N2' to 'N < N2'. This ensures that the station at position N2 (final station in segment) is not classified as traversed, making it possible to compare paths that involve multiple segments. It allows a recursive call to the tail of the path, which ensures the entire path is checked and not just the initial segment.

stations_traversed(Path, Stations) calls findall on the previously-defined predicate **station_traversed** to generate the list of all **Stations** already traversed in **Path.** It uses findall rather than setof because although they both find the same set (assuming the database is correct, stations are not supposed to be repeated on a line), findall does not sort the list and therefore is less expensive.

segment_adds_cycle(segment(L,S1,S2),Path) checks if a **segment** makes a **Path** cyclic by checking if any of the stations in the given segment are a member of **Stations**, the list of **traversed stations** found via **stations_traversed**. This ensures the **path** predicate only generates finite paths between given stations.

path(S1, S2, Path) computes a path from station **S1** to station **S2,** and returns it in **Path**. In order to avoid endless loops, it has to keep track of the stations already traversed. This allows it to check before adding a segment to the path (built so far) if it will generate a cyclic graph or traverse a line that has already been visited. Since it only takes three arguments, a predicate **path_helper** had to be written that has an additional argument **Attempted_path** that meets the requirements of the task.

Predicate **path_helper(S1, S2, Attempted_path, Path)** tests whether there exists a **Path** from **S1** to **S2**, using **Attempted_path** to prevent stations and lines from being traversed more than once. The base case occurs when stations **S1** and **S2** exist on the same line, with **S2** coming after **S1** in the station ordering for that line. In this case, it ensures a path doesn't traverse stations and lines more than once by negating the previously defined **member_of_path** and **segment_adds_cycle** predicates, passing the currently traversed path **Attempted_path** as argument. The idea behind this reasoning is that if we cannot prove that we are not able to disprove these predicates, then the negations will pass and the last statement assigning the segment to **Path** is reached. If a path is greater than one segment in length, the recursive case is called. The remaining segments are added to **Path** after each of the recursive calls have succeeded, hence the last statement in which we add the correct segment to the head of the list **Tail_path**. **Tail_path** stores the current path between **S_Middle** and **S2** for a given **Attempted_path**. This allows for a recursive construction of the path **Path,** from the final to the first segment, ensuring segments are stored and returned in the correct order.

Question 5:

Predicate **minimum_path(S1,S2,Path)** verifies if **Path** is a path from station **S1** to station **S2** with a minimum number of changes (i.e. minimum number of segments)**.** It is implemented by manipulating the builtin setof predicate such that **AllPaths** stores a list of all possible paths between **S1** and **S2** and their respective lengths (in (Length,Path) tuples). setof was used here over findall, because we need the list of paths to be sorted by length. As a result of this sorting, we can use the builtin nth0 predicate to store the first element of **AllPaths** and be sure this is either the shortest path or one of the shortest paths from **S1** to **S2**. Finally, **more_paths** is called to find any remaining paths of the same length.

more_paths(AllPaths, MinPath, Path) is a predicate that (recursively) traverses a list of (Length,Path) tuples and checks if the Length is equal to **Min** (the minimum length argument). If paths of equal length are found, they are returned in **Path**. The implementation works by first inspecting the head element and checking if this path's number of changes is equal to **Min,** storing it in **Path** if it is**,** then making a recursive call with the tail of the list **T** to inspect remaining tuples. This way **Path** stores all possible paths between **S1** and **S2** that have length **Min**.

```
% question(1)

multiple_lines(S):- line(X), line(Y), X\==Y, stop(X, _, S), stop(Y, _, S).


% question(2)

non_termini(L, E):- stop(L, E1, _), E1 > E.
termini(L, S1, S2):- stop(L, 1, S1), stop(L, E, S2), \+non_termini(L, E).


% question(3)

isort(Line, [], []).
isort(Line, [H|T], L) :- isort(Line, T, Ts), insert(Line, H, Ts, L).

insert(Line, X, [H|T], [H|Ti]) :- stop(Line, E1, X), stop(Line, E2, H), E1 > E2, !, insert(Line, X, T, Ti).
insert(Line, X, L, [X|L]).

list_stops(Line, List):- findall(S, stop(Line, _, S), First_list), isort(Line, First_list, List).


% question(4)

member_of_path(segment(Line, _, _), [segment(Line, _, _)|_]).
member_of_path(segment(Line, _, _), [_|T]):- member_of_path(segment(Line, _, _), T).

station_traversed(S, [segment(L,S1,S2)|_]):- stop(L, N, S), stop(L, N1, S1), stop(L, N2, S2), N1 =< N, N < N2, N1 =< N2.
station_traversed(S, [_|R]):- station_traversed(S, R).

stations_traversed(Path, Stations):- findall(X, station_traversed(X, Path), Stations).

segment_adds_cycle(segment(L, S1, S2), Path):- station_traversed(S, [segment(L, S1, S2)]), stations_traversed(Path, Stations), member(S, Stations).

path(S1, S2, Path):- path_helper(S1, S2, [], Path).

path_helper(S1, S2, Attempted_path, Path):- stop(X, N1, S1), stop(X, N2, S2), N1 < N2,
                                            \+ member_of_path(segment(X, S1, S2), Attempted_path),
                                            \+ segment_adds_cycle(segment(X, S1, S2), Attempted_path),
                                            Path = [segment(X, S1, S2)].
path_helper(S1, S2, Attempted_path, Path):- stop(X, N1, S1), stop(X, N_Middle, S_Middle), N1 < N_Middle,
                                            \+ member_of_path(segment(X, S1, S_Middle), Attempted_path),
                                            \+ segment_adds_cycle(segment(X, S1, S_Middle), Attempted_path),
                                            path_helper(S_Middle, S2, [segment(X, S1, S_Middle)|Attempted_path], Tail_path),
                                            Path = [segment(X, S1, S_Middle)|Tail_path].


% question(5)

minimum_path(S1,S2,Path) :- setof((Length, Path), (path(S1, S2, Path), length(Path, Length)), AllPaths),
                            nth0(0, AllPaths, MinPath),
                            more_paths(AllPaths, MinPath, Path).

more_paths([(Len, Pat)|_], (Min,_), Path) :- Len == Min, Path = Pat.
more_paths([_|T], MinPath, Path) :- more_paths(T, MinPath, Path).
```