PROLOG DESCRIPTIVE ACCOUNT

ECM2418

660037119

NOTE: The code also provides useful comments to support key concepts raised in this report.

Question 1

checkGraph/0

For an overall concept, checkGraph examines all possible edges in the graph. It then calls various check functions for each edge examined. If an error is detected, this will be written to the user. I originally had a solution where all the check functions were called the same, similar to the workshop example `check_lives`. I decided against using this construct in the end since I developed this new solution which avoids having to go through all the edges many times, and so would be more efficient computationally. The fail is necessary to enforce backtracking to select a new edge to examine, I also included checkGraph as a fact so that it always returns true.

Question 2

member(X,L)

This function recursively traverses though a list looking at each element, and seeing if it pattern matches to the target using a base case. If this base case is met, the function will return true. On the other hand, if the target is not found and the list becomes empty, no function will match the empty examined list leading to a false being returned.

isSet(L)

By comparing the length of `L` being sorted and non sorted, I can determine if the list is a set or not. This is because sorting a list removes the duplicates, meaning if the lengths are equal then it is a set and will return true, otherwise false.

sameLength(L1, L2)

This function used by isSet/2 and checks if two lists are the same length. It completes this by removing one element from both lists at a time, if both lists become empty simultaneously(base case), then they were of equal length.

lastElement(Z,L)

Empties the list, upon reaching the empty set will attempt to match the last element removed to `Z`.

append(L1, L2, L)

Empties and keeps track of `L1`s elements, when the first list is empty `L2` is unified to `L`. Exit calls add ‘L1`s elements to the front of `L`. The result is `L2` being concatenated (appended) to the end of `L1`.

intersect(L1, L2, L)

Checks elements in `L1` one by one to see if they are a member of `L2`, if so, these elements are added to the result list when backtracking. Cuts avoid unnecessary backtracking, such as inclusion of duplicates and the empty set in the result.

Question 3

wPathRoute(X,Y,L,W)

Inspired by the given path/2 function, this function similarly finds all the possible paths from `X` to `Y`, but also assigns the found path to `L` for each as well as the total path weight. The concept of keeping track of visited vertices and not allowing revisiting is the same here as in path/2(important for a cyclic graph). Therefore upon reaching the destination, all I had to do was assign the already visited vertices list to `L`. Similarly, each time a vertex was added to the visited list(VISITED), I also added the weight of the edge taken to get there to an accumulator(ACC_W). This was used to retrieve the total weight upon reaching the destination.

Note: I am using append here so that the vertices are added to the visited list in the correct order, I also append the destination to the visited list in the base case to complete the path.

pathRoute(X,Y,L)

wPathRoute/4 covers the needed functionality here, therefore I could have simply exploited this via:

```
pathRoute(X,Y,L) :-
    wPathRoute(X,Y,L,_).
```

This would work, however I chose not to do this. The reasoning here is that wPathRoute/4 does more computation than needed (like accumulating the weights also), thus would unnecessarily increase the space and computation complexity of the function. Even though this overhead is minimal, it can definitely make a difference if the function was to be called a large amount of times. Instead I have taken the necessary code from wPathRoute to redefine the function. I believe in prioritising efficiency over repeated code.

wPath(X,Y,W)

Again, instead of using:

```
wPath(X,Y,W) :-
    wPathRoute(X,Y,_,W).
```

Because this would unnecessarily also compute the route. I have taken the necessary code to redefine this function from wPathRoute/4.

path(X,Y)

This was already defined. I could have exploited my already made wPath/3 as so:

```
path(X,Y) :-
    wPath(X,Y,_).
```

But this, again, would make the function unnecessarily less efficient. Therefore I used the pre defined function in the code.

wPathAvoidSetRoute(X,Y,SET,L,W)

Identical to wPathRoute, with the difference being an added check ensuring the next planned vertex is not a member of `SET`. I have included this check before the member check of visited, as I would expect the visited list to normally be longer than `SET`. This means that if a vertex is not in `SET` it saves having to check whether it is a member of the already visited vertices list.

pathAvoidSetRoute(X,Y,SET,L)

Again, instead of:

```
pathAvoidSetRoute(X,Y,SET,L) :-
    wPathAvoidSetRoute(X,Y,SET,L,_).
```

I have taken the relevant code from wPathAvoidSetRoute/5 and redefined it for efficiency.

shortestPathAvoidSet(X,Y,SET,L,W)

This function exploits wPathAvoidSetRoute/5 to calculate all the possible paths with their corresponding route and weight using findall/3. Therefore each path route with it's weight is put into a list [Path, Weight], nested within a parent list holding all the paths. The longest possible path is calculated for the graph using longestPossibleRoute/1, we add one to this and make it the initial state of the shortest path comparison. I could have hard coded a very large number here in this case, but it would not be scalable if many more vertices were added.
The helper function then recursively evaluates all the paths, if the path is shorter than the currently known shortest, the route and it's weight are recorded. Otherwise the function ignores this path and evaluates the next path. I utilise a cut here to ensure that both cases are not executed for the same path, as this would lead to unnecessary searching and possibly incorrect results.

shortestPath(X,Y,L,W)

Same approach as shortestPathAvoidSet/5, but instead uses wPathRoute to calculate all the possible path combinations.

connectedGraph/0

My logic here was to check that every possible pairing of vertices had a path. Therefore this function uses kSetVert/2 to compute all permutations of 2 vertices, representing all possible pairings. I then check that for each pairing there exists a path between them, using the path/2 function. To ensure this function will only return true if all the pairs have a path (and not just some), I encapsulate this code within forall/2.

Question 4

buildSST(T)

My logic here is to compute all possible permutations which include all the vertices, within the construction of these permutations I disregard the permutations that are not valid tours. The structure of this function vaguely resembles the structure of kSetVert/2.

1. I first find the number of vertices in the graph, this is done by computing a list of all the vertices using listVertices/2 and finding the length of this. I will use this to define how many vertices I need to compute for the tour.
2. I then ensure there is a valid number of vertices, if there are no vertices this function will return an empty set, cutting any further execution.
3. A starting vertex is chosen, since the start can be any vertex, I use `vertex(X)`, backtracking will consider all the other options. The number of vertices needed to be computed is reduced by one.

4. The helper function is then called which takes in the number of vertices left to compute, the original start vertex, the previously added vertex, and the tour route so far. Note here that at this point, the start vertex and previously added vertex are the same (X).
5. If there are vertices still left to compute, the helper function will choose another vertex, it will then ensure that this vertex has an edge from the previous vertex in the current tour route. If so, it will reduce the number of vertices needed to be computed by one and append it to the recorded tour route.
6. The helper function is then called recursively with the new number of vertices left to compute, the new previously added vertex and current tour route.
7. Finally there is a base case of when the number of vertices left to compute is 0, in this case the helper function will do one last check to ensure the end point of the tour has an edge to the start of the tour. If so, this computed permutation is a valid tour, the start vertex is appended to the end. This tour is then assigned to the result.

Originally, I computed all the permutations and then afterwards checked through them all again to see which ones were valid, understandably this was very slow. Therefore checking the permutations during their construction and stopping construction if invalid (as above) was a much more efficient solution that I adopted.

buildSSTWithStart(V,T)

I heavily exploit the above function here to do most of the work. The only difference here is that instead of choosing a random vertex to start, I forcibly choose the inputted start vertex before calling the helper. The helper defined for buildSST(T)/1 is then called, since functionality needed is identical from this point on, as shown in code comments.

Question 5

My approach for this question was to generate all subsets of the vertices, and then examine which of the subsets would be a valid placement for the fire stations.

reachableFireStation(V, L)

This function I defined to check whether a vertex `V` would be safe with fire stations built on vertices listed in `L`. This is done by ensuring there is a path from one of the vertices in `L` with a weight equal to or less than 5. Since only one fire station is needs to be in range, when it has been proven that at least one fire station is within range, further searching is stopped via a cut to save computation.

totalFireStationCost(L, C)

Calculates how much it would cost (`C`) to build a fire station on every vertex listed in `L`.

buildSafeSetFSWithinBudget(B, S)

1. Calculate the subsets of all the vertices using subsetVert/1, examining each subset after computation and evaluating it is a valid fire station placement combination.

2. In order to do this I check that every known vertex can reached by a fire station in under 5 minutes; I call reachableFireStation/2 for this.
3. If this has passed, I then compute the total cost of building a fire station on each of the vertices in the subset. If this is below the budget then it is a valid fire station placement combination and so is added to the result.

highestPossibleCost(TOTAL)

Calculates the cost `TOTAL` of building a fire station on all vertices.

computeMinCostSafety(BMIN)

1. Calculate a list of all possible valid subset fire station placements without any budget restrictions in place using buildAllSafeSetFS/1 and findall/3.
2. Calculate the highest possible cost with highestPossibleCost/1, and add 1 to it. This will be used as a scalable initial cheapest path comparison value. Then call the helper function passing in the list of valid subsets and the highest possible cost.
3. The helper function will then examine each subset, if the subset is cheaper than the known one it will record the subset's cost. Otherwise the subset is ignored, to stop both of these executing for the same subset a cut is needed. (Similar to the structure of shortestPathAvoidSet/5).
4. Finally when all subsets have been considered and the list is empty, the cheapest solution found is assigned to `BMIN`.

buildMinCostSafeFS(BMIN,S)

This can now easily be achieved with:

```
buildMinCostSafeFS(BMIN,S) :-
    computeMinCostSafety(BMIN),
    buildSafeSetFSWithinBudget(BMIN,S).
```

However, this is very inefficient since the subset associated with the cheapest cost can easily be recorded within computeMinCostSafety/1. Therefore I chose to implement this by repeating most of the code from computeMinCostSafety/1 and making the minor change of also recording and returning the subset(LOCATIONS) to the user. This avoids having to call buildSafeSetFSWithinBudget/2 using the `BMIN` unnecessarily which produces a major amount of computational overhead.

```prolog
edge(a,b,5).
edge(b,a,5).
edge(b,c,3).
edge(c,a,2).
edge(c,d,4).
edge(d,b,6).
edge(c,f,4).
edge(f,c,4).
edge(e,c,5).
edge(f,e,7).
edge(g,a,3).
edge(d,g,8).
edge(e,g,2).
edge(f,h,3).
edge(h,i,2).
edge(i,j,3).
edge(j,h,4).
edge(d,h,1).
edge(j,f,6).
edge(l,k,-1).
edge(k,l,4).
edge(a,z,-2).

vertex(a).
vertex(b).
vertex(c).
vertex(d).
vertex(e).
vertex(f).
vertex(g).
vertex(h).
vertex(i).
vertex(j).

costFS(a,20).
costFS(b,10).
costFS(c,5).
costFS(d,8).
costFS(e,12).
costFS(f,18).
costFS(g,9).
costFS(h,7).
costFS(i,14).
costFS(j,2).


%Q1

nonExistent(START, FINISH) :-
    \+vertex(START),
    format('Vertex ~w of edge (~w,~w) is not a valid vertex.',
           [START, START, FINISH]), nl;

    \+vertex(FINISH),
    format('Vertex ~w of edge (~w,~w) is not a valid vertex.',
           [FINISH, START, FINISH]), nl.

weightValueCheck(START, FINISH, WEIGHT) :-
    WEIGHT < 1,
    format('Vertex (~w, ~w) has weight ~w which is less than 1.',
           [START, FINISH, WEIGHT]), nl.

weightConsistencyCheck(START, FINISH, WEIGHT) :-
    edge(FINISH, START, REVERSEWEIGHT),
    WEIGHT =\= REVERSEWEIGHT,
    format('Edge (~w, ~w) has weight ~w and edge (~w, ~w) has weight ~w which is inconsistent.',
           [START, FINISH, WEIGHT, FINISH, START, REVERSEWEIGHT]), nl.
```

```
checkGraph :-
    edge(START, FINISH, WEIGHT),
    (nonExistent(START, FINISH);
    weightValueCheck(START, FINISH, WEIGHT);
    weightConsistencyCheck(START, FINISH, WEIGHT)),
    fail.   %Forces backtracking to ensure all edges are considered.

checkGraph.


%Q2
member(X, [H|T]) :-
    memberHelper(T, X, H).

memberHelper(_, X, X).   %If member found, true fact.
memberHelper([H|T], X, _) :-
    memberHelper(T, X, H).


isSet(L) :-
    is_list(L),
    sort(L, SortedL),   %Removes duplicates.
    sameLength(L, SortedL).

sameLength([], []).     %Both become empty only simultaneoulsy only if same size.
sameLength([_|T1], [_|T2]) :-
    sameLength(T1, T2).


lastElement(Z, [H|T]) :-
    lastHelper(T, H, Z).

lastHelper([], Z, Z).   %Last element found when the list is now empty.
lastHelper([H|T], _, Z) :-
    lastHelper(T, H, Z).


append([], L, L).
append([H|T], L, [H|T2]) :-
    append(T, L, T2).


intersect([], _, []) :- !.
intersect([H|T], B, C) :-
    member(H, B), !,
    C = [H|T2],
    intersect(T, B, T2).

intersect([_|T], B, T2) :-
    intersect(T, B, T2).


%Q3

wPathRoute(X, Y, L, W) :-
    wPathRouteHelper(X, Y, [], 0, L, W).

wPathRouteHelper(X, X, VISITED, ACC_W, L, W) :-
    append(VISITED, [X], L),    %Append the destination to the final returned route.
    W = ACC_W.
wPathRouteHelper(X, Y, VISITED, ACC_W ,L, W) :-
    edge(X, Z, WT),
    \+member(Z, VISITED),
    W1 is ACC_W + WT,
    append(VISITED, [X], P),
    wPathRouteHelper(Z, Y, P, W1, L, W).
```

```prolog
pathRoute(X, Y, L):-
    pathRouteHelper(X, Y, [], L).

pathRouteHelper(X, X, VISITED, L) :-   %Target destination found (Base case).
    append(VISITED, [X], L).
pathRouteHelper(X, Y, VISITED, L):-    %General Case.
    edge(X, Z, _),
    \+member(Z, VISITED),
    append(VISITED, [X], R),
    pathRouteHelper(Z, Y, R, L).


wPath(X, Y, W):-
    wPathHelper(X, Y, [], 0, W).

wPathHelper(X, X, _, ACC_W, W) :-
    W = ACC_W.
wPathHelper(X, Y, VISITED, ACC_W ,W):-
    edge(X, Z, WEIGHT),
    \+member(Z, VISITED),
    W1 is ACC_W + WEIGHT,   %Accumilate weights.
    wPathHelper(Z, Y, [X|VISITED], W1, W).


path(X, Y):-
    pathHelper(X, Y, []).

pathHelper(X, X, _).
pathHelper(X ,Y, VISITED):-
    edge(X, Z, _),
    \+member(Z, VISITED),
    pathHelper(Z, Y, [X|VISITED]).


wPathAvoidSetRoute(X, Y, SET, L, W) :-
    wPathAvoidSetRouteHelper(X, Y ,SET, [], 0, L, W).

wPathAvoidSetRouteHelper(X, X, _, VISITED, ACC_W, L, W) :-
    append(VISITED, [X], L),
    W = ACC_W.

wPathAvoidSetRouteHelper(X, Y, SET, VISITED, ACC_W ,L, W) :-
    edge(X, Z, WEIGHT),
    \+member(Z, SET),     %Check the vertex is not within the SET before continuing.
    \+member(Z, VISITED),
    W1 is ACC_W + WEIGHT,
    append(VISITED, [X], P),
    wPathAvoidSetRouteHelper(Z, Y, SET, P, W1, L, W).


pathAvoidSetRoute(X, Y, SET, L) :-
    pathAvoidSetRouteHelper(X, Y, SET, [], L).

pathAvoidSetRouteHelper(X, X, _, VISITED, L) :-
    append(VISITED, [X], L).

pathAvoidSetRouteHelper(X, Y, SET, VISITED, L) :-
    edge(X, Z, _),
    \+member(Z, SET),
    \+member(Z, VISITED),
    append(VISITED,[X], P),
    pathAvoidSetRouteHelper(Z, Y, SET, P, L).


%Calculates the longest possible path.
longestPossibleRoute(V) :-
    findall(W,edge(_,_,W), WEIGHTS),    %Retrieve all edge weights as a list.
    sum_list(WEIGHTS, V).
```

```
shortestPathAvoidSet(X, Y, SET, L, W) :-
    %Obtain all possible paths to begin with.
    findall([PATH, WEIGHT], wPathAvoidSetRoute(X, Y, SET, PATH, WEIGHT), PATHS),
    PATHS \= [],
    longestPossibleRoute(LONG),
    LONG1 is LONG + 1,    %Used for initial CUR_W (current shortest path known).
    shortestPathAvoidSetHelper(PATHS, [], LONG1, L, W).

shortestPathAvoidSetHelper([], FIN_L, FIN_W, L, W) :-
    L = FIN_L,
    W = FIN_W.
shortestPathAvoidSetHelper([[PATH, WEIGHT]|T], CUR_L, CUR_W, L, W) :-
    (WEIGHT < CUR_W,
        %If path weight is shortest known, update this by recording path below.
        shortestPathAvoidSetHelper(T, PATH, WEIGHT, L, W), !);
    %Else, keep the current known shortest path, and examine the next path.
    shortestPathAvoidSetHelper(T, CUR_L, CUR_W, L, W).


%Simply uses `wPathRoute` instead.
shortestPath(X, Y, L, W) :-
    %Obtain all possible paths to begin with.
    findall([PATH, WEIGHT], wPathRoute(X, Y, PATH, WEIGHT), PATHS),
    PATHS \= [],
    longestPossibleRoute(LONG),
    LONG1 is LONG + 1,
    shortestPathHelper(PATHS, [], LONG1, L, W).

shortestPathHelper([], FIN_L, FIN_W, L, W) :-
    L = FIN_L,
    W = FIN_W.
shortestPathHelper([[PATH,WEIGHT]|T], CUR_L, CUR_W, L, W) :-
    (WEIGHT < CUR_W,
        shortestPathHelper(T, PATH, WEIGHT, L, W), !);
    shortestPathHelper(T, CUR_L, CUR_W, L, W).

%Compute all possible permutations of K vertices.
kSetVert(K, V):-
    K = 0, !, V = [].
kSetVert(K, V):-
    K > 0,
    kSetVertHelper(K, [], V).

kSetVertHelper(K, ACC, RES):-
    K > 0,
    vertex(X),
    \+member(X, ACC),
    K1 is K - 1,
    kSetVertHelper(K1, [X|ACC], RES).
kSetVertHelper(0,ACC, RES):-
    RES = ACC.

%Test every possible verticy pairing, ensuring there is a path between each.
connectedGraph :-
    forall(kSetVert(2, [START|[FINISH|_] ]),
        path(START, FINISH)).


%Q4

%returns all the known graph vertices as a list L.
listVertices(L):- listVerticesHelper([], L).
listVerticesHelper(ACC, RES):-
    vertex(X),
    \+ member(X, ACC), !,
    listVerticesHelper([X|ACC], RES).
listVerticesHelper(ACC, RES):-
    RES = ACC.
```

```prolog
buildSST(T):-
    listVertices(L),
    length(L, N),
    (N = 0, !, T = [];  %If there are no vertices only tour is [].
    N > 0,
    vertex(X),  %Otherwise, choose a starting vertex.
    N1 is N - 1,
    buildSSTHelper(N1, X, X, [X], T)).   %Search for possible tours.

%Called when the tour is complete.
buildSSTHelper(0, START, FINISH, ACC, RES):-
    edge(FINISH, START, _),   %Ensure we can return back to the start of the tour.
    append(ACC, [START], NEW_ACC),
    RES = NEW_ACC.

%Called mid tour,
%`PREV` represents the last vertex added to the tour.
%`START` represents the original starting vertex of the tour.
buildSSTHelper(N, START, PREV, ACC, RES) :-
    vertex(X),
    \+member(X, ACC),
    edge(PREV,X,_),    %Ensures there is an edge to this next vertex.
    N1 is N - 1,
    append(ACC, [X], NEW_ACC),
    buildSSTHelper(N1, START, X, NEW_ACC, RES).


%BuildSST with the added functionality of inputting a starting vertex.
buildSSTWithStart(V, T):-
    listVertices(L),
    length(L, N),
    (N = 0, !, T = [];
    N > 0,
    N1 is N - 1,
    %Required functionality from here on is identical to buildSST.
    buildSSTHelper(N1, V, V, [V], T)).  %Call buildSST pre defined helper function.


%Q5
%If X is not the least vertex in terms of lexographical ordering.
nonMinLexVertex(X):-
    vertex(X),
    vertex(Y),
    X @> Y.

%Compute the least vertex X in terms of lexographical ordering.
minLexVertex(X):-
    vertex(X),
    \+nonMinLexVertex(X).

vertexInBetween(X, Y, Z):-
    vertex(X),
    vertex(Y),
    vertex(Z),
    Y @> Z,
    Z @> X.

%Calculate the immediate successor of X (Y) in the order.
succVertex(X, Y):-
    vertex(X),
    vertex(Y),
    Y @> X,
    \+ vertexInBetween(X, Y, _).

%Calculate all the subsets.
subsetVert(S):-
    minLexVertex(X),
    subsetVertHelper([], X, S).
```

```prolog
subsetVertHelper(ACC, LAST_CONSIDERED, S):-
    succVertex(LAST_CONSIDERED, X),
    subsetVertHelper(ACC, X, S).
subsetVertHelper(ACC, LAST_CONSIDERED, S):-
    succVertex(LAST_CONSIDERED, X),
    subsetVertHelper([X|ACC], X, S).
subsetVertHelper(ACC, LAST_CONSIDERED, S):-
    \+ succVertex(LAST_CONSIDERED, _), S = ACC.

%Determine whether vertex V can reach a fire station at any of the vertices given
%(In under or equal to 5 minutes).
reachableFireStation(V, [H|T]) :-
    (wPath(H,V,W), W =< 5, !);  %If one is reachable, no need to check the rest.
    reachableFireStation(V,T).

%Take list of nodes, calculate total fire station cost.
totalFireStationCost(L, C) :-
    totalFireStationCostHelper(L, 0, C).

totalFireStationCostHelper([H|T], ACC, C) :-
    costFS(H,P),
    NEW_ACC is ACC + P,
    totalFireStationCostHelper(T, NEW_ACC, C).
totalFireStationCostHelper([], ACC, C) :-
    C is ACC.

buildSafeSetFSWithinBudget(B,S) :-
    subsetVert(SV),    %Calculate all subsets of vertices.
    forall(vertex(X),
        reachableFireStation(X, SV)), %Ensure every vertex can reach a Fire Station.
    totalFireStationCost(SV, C),
    C =< B,    %Ensure the solution is within budget.
    S = SV.

%Finds all possible Fire Station placements without a budget concerned.
buildAllSafeSetFS(S) :-
    subsetVert(Ss),
    forall(vertex(X),
        reachableFireStation(X, Ss)),
    S = Ss.

%Find the cost of building Fire Stations at every vertex.
highestPossibleCost(TOTAL) :-
    findall(C,costFS(_,C),COSTS),
    sum_list(COSTS, TOTAL).


computeMinCostSafety(BMIN) :-
    findall(SV,buildAllSafeSetFS(SV),LSV),  %Find all valid station placements.
    LSV \= [],
    highestPossibleCost(T),    %Use highest possible cost + 1 as initial cheapest.
    T1 is T + 1,
    computeMinCostSafetyHelper(LSV, T1, BMIN).

computeMinCostSafetyHelper([], CUR_MIN, BMIN) :-
    BMIN = CUR_MIN.
computeMinCostSafetyHelper([H|T], CUR_MIN, BMIN) :-
    totalFireStationCost(H, C),
    (C < CUR_MIN,
        %If placement is cheaper, record this placement.
        computeMinCostSafetyHelper(T, C, BMIN), !);
    %Otherwise, keep current placement recorded and examine next placement.
    computeMinCostSafetyHelper(T, CUR_MIN, BMIN).
```

```
%Same as computeMinCostSafety, but retains the placement of the cheapest solution.
buildMinCostSafeFS(BMIN,S) :-
    findall(SV,buildAllSafeSetFS(SV),LSV),
    LSV \= [],
    highestPossibleCost(T),
    T1 is T + 1,
    buildMinCostSafeFSHelper(LSV, T1, BMIN, [], S).

buildMinCostSafeFSHelper([], CUR_MIN, BMIN, LOCATIONS, S) :-
    BMIN = CUR_MIN,
    S = LOCATIONS.
buildMinCostSafeFSHelper([H|T], CUR_MIN, BMIN, LOCATIONS, S) :-
    totalFireStationCost(H, C),
    (C < CUR_MIN,
        buildMinCostSafeFSHelper(T, C, BMIN, H, S), !);
    buildMinCostSafeFSHelper(T, CUR_MIN, BMIN, LOCATIONS, S).
```