Name: Pratik Suryawanshi

ASU ID: 1213231238

Problem 1

Find a path in a weighted graph

For this problem, we can store the edges as facts and rules to check if path exists between two nodes or if they are adjacent to each other, helper function to recursively check for path and do weight sum.

Code:

```
%Find a weight and path in weighted graph.

path(a,b,1).

path(a,c,6).

path(b,c,4).

path(b,d,3).

path(b,e,1).

path(c,d,1).

path(d,e,1).

%rule for the edge between two nodes.

adjacent(Source,Destination,Weight):-

            path(Source,Destination,Weight);

            path(Destination,Source,Weight).

findpath(Source,Destination,Weight,Path):-

            helperFunction(Source,Destination,[Source],Weight,Queue),

   reverse(Queue,Path).

%Basecase when there are only two nodes.

helperFunction(Source,Destination,VisitedNodes,Weight,[Destination|VisitedNodes]):-

            adjacent(Source,Destination,Weight).

helperFunction(Source,Destination,VisitedNodes,FinalWeight,Path):-

            adjacent(Source,MidNode,W1),

            MidNode \== Destination,
            not(member(MidNode,VisitedNodes)),

            helperFunction(MidNode,Destination,[MidNode|VisitedNodes],W2,Path),

            FinalWeight is W1 + W2.
```

Output :

```
?- findpath(a,e,Weight,Path).

Weight = 2,

Path = [a, b, e] ;

Weight = 7,

Path = [a, b, c, d, e] ;

Weight = 5,

Path = [a, b, d, e] ;

Weight = 8,

Path = [a, c, d, e] ;

Weight = 11,

Path = [a, c, d, b, e] ;

Weight = 11,

Path = [a, c, b, e] ;

Weight = 14,

Path = [a, c, b, d, e] ;
```

Problem 2:

Tower of Hanoi

In this problem, we can write base condition for single transaction of transferring disk from one peg to other and main predicate which does all transfers.

Code:

```
%Problem 2

%Tower of Hanoi

%input is num, source, destination and intermediate peg.

%base case.
```

```
hanoi(1,Source,Destination,_):-
        write('Move '),
        write(Source),
        write(' to '),
        write(Destination),
        nl.
hanoi(Num,Source,Destination,Inter):-
        Num > 1,
        NewNum is Num-1,
        hanoi(NewNum,Source,Inter,Destination),
        hanoi(1,Source,Destination,Inter),
        hanoi(NewNum,Inter,Destination,Source).
```

Output:

```
?- hanoi(3,a,c,b).
Move a to c
Move a to b
Move c to b
Move a to c
Move b to a
Move b to c
Move a to c
true.
```

Problem 3

Write in full words:

In this problem, we can use facts to store info related to mapping of integer and its word representation. Later we can recursively get the remainder of number and form a string of words concatenated with hyphen. This recursive logic is covered in helper function.

Code:

```
%Problem 3 full words %

% db to store map of words


word(0,zero).

word(1,one).

word(2,two).

word(3,three).

word(4,four).

word(5,five).

word(6,six).

word(7,seven).

word(8,eight).

word(9,nine).


full_words(Num) :-

        Quotient is div(Num,10),

        helperFunction(Quotient),

        Remainder is mod(Num,10),

        word(Remainder,Word),

        write(Word).


%this is base caee.

helperFunction(0).


%this function will get the remainder and print it with dash and recursively divide number by 10 to get
next number

helperFunction(Num) :-
```

```
        Num > 0,
        Quotient is div(Num,10),
        helperFunction(Quotient),
        Remainder is Num mod 10,
        word(Remainder,Word),
        dash(Word).

%function to print -
dash(Word) :-
        write(Word),
        write('-').
```

Output:

```
?-
|   full_words(123).
one-two-three
true .
?-
|    full_words(2018).
two-zero-one-eight
true .
```

Problem 4

Generate the combinations of K distinct objects chosen from the N elements of a list

In this problem we can use backtracking logic to get possible combination of number in list.

Code:

```
%Problem4
%Generate the combinations of K distinct objects chosen from the N elements of a list.

combination(Num, Input, Output) :-
```

```
        length(Input, InLength),
        length(Output, Num),
        Num =< InLength,
        helperFunction(Output, Input).

%base case to stop the recursive call.
helperFunction([], _).

helperFunction([Head|Tail], Input) :-
        select(Head, Input, Output),
        helperFunction(Tail, Output).
```

Output:

```
?- combination(3,[a,b,c,d],L).
L = [a, b, c] ;
L = [a, b, d] ;
L = [a, c, b] ;
L = [a, c, d] ;
L = [a, d, b] ;
L = [a, d, c] ;
L = [b, a, c] ;
L = [b, a, d] ;
L = [b, c, a] ;
L = [b, c, d] ;
L = [b, d, a] ;
L = [b, d, c] ;
L = [c, a, b] ;
L = [c, a, d] ;
L = [c, b, a] ;
L = [c, b, d] ;
```

```
L = [c, d, a] ;

L = [c, d, b] ;

L = [d, a, b] ;

L = [d, a, c] ;

L = [d, b, a] ;

L = [d, b, c] ;

L = [d, c, a] ;

L = [d, c, b] ;

false.
```

Problem 5

Map Coloring (Incomplete implementation submitted for partial credit)

In this problem, we can use edges, color and paths as facts to store data. Then to check for conflict i.e. if adjacent nodes have same color we can use a rule. The main predicate will recursively form combination/ fill colors of nodes and check if there is conflict. If at end of covering all nodes there is still no conflict then result is found.

Code:

```
edge(1,2).

edge(1,3).

edge(1,4).

edge(1,6).

edge(2,3).

edge(2,5).

edge(3,4).

edge(3,5).

edge(3,6).

edge(3,5).

edge(3,6).


vertex(1).
vertex(2).

vertex(3).

vertex(4).

vertex(5).

vertex(6).


path(X,Y):-

        edge(X,Y);

        edge(Y,X).


color(red).

color(blue).

color(green).

color(yellow).


% very first time with one vertex and one color

color_map(L):-

        color(Color),

        vertex(Vertex),

        append([Vertex|[Color]],[],Row),

        RowNext = [Row],

        VertexNext is Vertex + 1 ,

        helperFunction(VertexNext,RowNext,L).


%helper function for coloring

helperFunction(Vertex,Row,L):-

        color(Color1),

        vertex(Vertex),

        check_for_conflict(Vertex,Color1,Row),
        append(Row,[Vertex,Color1],Row1),
        VertexNext is Vertex+1,
        helperFunction(VertexNext,Row1,L).

helperFunction(_,Row,L):-
        length(Row,N),
        N = 6,
        L = Row.

check_for_conflict(_,_,[]).

check_for_conflict(Vertex,Color,[[V|_]|T]):-
        \+(path(Vertex,V)),
        check_for_conflict(Vertex,Color,T).

check_for_conflict(Vertex,Color,[[V3|[ColorNext]]|T]):-
        path(Vertex,V3),
        Color \== ColorNext,
        check_for_conflict(Vertex,Color,T).

%helper function for length
lenList([],0).
lenList([_|Tail],Num):-
        lenList(Tail,Next),
        Next is Num + 1.
```

Problem 6

N queens problem:

In this problem, we are trying to find positions of N queens in N * N board so that no 2 queens will be on position that other queen can attack. This is achieved through using backtracking logic. We put a queen on certain position which is within boundary of board. If that position is safe then we move to put next queen on adjacent position recursively by checking if that position is safe until we find a possible combination where n queens are placed and on queen is under attack by other queen.

Code:

```
%n queens problem to show place of queens without any conflict

queens(Num,Queens):-

        generateList(1,Num,Ns),

        queens(Ns,[],Queens).

queens(QueenList,SafeQueens,Queens):-

        select(Queen,QueenList,QueenList1),

        not(attack(Queen,SafeQueens)),

        queens(QueenList1,[Queen|SafeQueens],Queens).

queens([],Queens,Queens).


attack(A,As):-

        attack(A,1,As).


%predicate to calculate safe rows and columns

attack(A,N,[B|Ys]):-

        A is B + N;

        A is B - N.


attack(A,N,[B|Ys]):-

        Next is N + 1,

        attack(A,Next,Ys).


%helper function to generate list.
```

```
generateList(P,P,[P]).

generateList(A,B,[A|L]) :-
        A < B,
        Next is A + 1,
        generateList(Next,B,L).
```

Output:

```
?- queens(4,Qs).
Qs = [3, 1, 4, 2] ;
Qs = [2, 4, 1, 3] ;
false.

?- queens(5,Qs).
Qs = [4, 2, 5, 3, 1] ;
Qs = [3, 5, 2, 4, 1] ;
Qs = [5, 3, 1, 4, 2] ;
Qs = [4, 1, 3, 5, 2] ;
Qs = [5, 2, 4, 1, 3] ;
Qs = [1, 4, 2, 5, 3] ;
Qs = [2, 5, 3, 1, 4] ;
Qs = [1, 3, 5, 2, 4] ;
Qs = [3, 1, 4, 2, 5] ;
Qs = [2, 4, 1, 3, 5] ;
```

Problem 7

Goldbach's Conjecture

In this problem we are trying to find a pair of prime numbers to obtain given input sum. We start with 2 prim numbers as facts 2 and 3. Then we can substrct the sum with these primes to check if result is prime. Predicate prime() is used to determine if number is valid prime number by checking if it is not even and it does not have factors. If number obtained is also prime then we found a pair of prim which add upto required result. Then move to next pair.

Code:

```
%Finding the pairs of prime number addition so result is even number


% base for prime

prime(2).

prime(3).


% predicate to check if number Num is prime number

% avoid even numbers and those with factors.

prime(Num) :-

        integer(Num),

        Num > 3,

        Num mod 2 =\= 0,

        \+ factor(Num,3).


% factor used while checking for prime

factor(Number,List) :-

        Number mod List =:= 0.


factor(Number,List) :-

        List * List < Number,

        L2 is List + 2,

        factor(Number,L2).
```

```
goldbach(Number,List) :-
        Number mod 2 =:= 0,
        Number > 4,
        goldbach(Number,List,3).
goldbach(Number,[CurrentNumber,N],CurrentNumber) :-
        N is Number - CurrentNumber,
        prime(N), CurrentNumber < N.
goldbach(Number,List,CurrentNumber) :-
        CurrentNumber < Number,
        getnext_prime(CurrentNumber,NextNumber),
        goldbach(Number,List,NextNumber).
%predicate to get next prime number
getnext_prime(CurrentNumber,NextNumber) :-
        NextNumber is CurrentNumber + 2,
        prime(NextNumber).
getnext_prime(CurrentNumber,NextNumber) :-
        P2 is CurrentNumber + 2,
        \+ prime(P2),
        getnext_prime(P2, NextNumber).
```

Output:

```
?- goldbach(28,L).
L = [5, 23] ;
L = [11, 17] .

?- goldbach(30,L).
L = [7, 23] ;
L = [11, 19] ;
L = [13, 17] ;
```