Part-2

Akanksha Bansal (akankshabansal90@gmail.com)
Gaurav Nanda (gaurav324@gmail.com)
Prateek Agarwal (prat0318@gmail.com)

Running Instructions:
$ sh ./runscript.sh

Objective:
To understand the way Prolog functions by checking the validity of a known theorem.

Description of PART-2:
As part of this assignment we check the validity of a basic theorem of the bipartite graphs.
In the solution to part (a) of the question we check that the graph provided as the input is actually a bipartite graph. And in the solution for part (b) we check that the cycles that exists in the graph are all of even length.

For part (a) we need to make sure that Unode and Vnode should form a disjoint set. The implementation contains the following constraints :

1. Node's Id should not be null
2. Unique ID
   Every Node being defined should have a unique Id. First we create a single table Node of all the Unodes and Vnodes. Then check that the Id's are unique in the Node table.
3. Check for a bipartite graph
   When an edge is defined between two nodes, then one of the coordinate should belong to the Unode and other end should belong to Vnode.

For part (b) we need to check that there are no cycles of Odd length. Below are the implementation details of the same:

a. Constraint:
   If we are able to find odd length cycle then the graph being defined is incorrect.
b. Cycle check needed only for one set of Nodes
   If a cycle exists then it will have at least one node either in Unode or in Vnode. Therefore if we are able to find all the cycles in the nodes of Unode, we can be sure that all the cycles present in the graph have been detected.
c. Find Cycles first
   In order to calculate the length of the cycle, we first try to find the cycles in the graph. In the process we will store the nodes we have traversed in a list.
d. Length of cycle is equal to no of nodes traversed
   Then the no of edges being traversed is equivalent to the no of nodes present in the list. So we can make sure that the cycle length is not odd by just checking the no of nodes traversed in the path.
e. Recursive Function
   We start off with a Unode and add it to the list of nodes that have been traversed. Then whenever a node is traversed, we add that to the list of visited nodes.
f. Detection of a cycle
   We declare that a cycle has been detected if the node we will now be visiting is the node we started our traversal with.

The tricky part was to make sure that the recursive function does not detect the same cycle again.
Below is the importance of each of the clauses being defined in the cycle determination

function.

a. The clause: *not(X == Z)* makes sure that the same edge is not picked by the recursive function for the next iteration. Without this clause the function tends to get into a infinite loop.
b. The clause: *not(member(Y,Visited))* makes sure that the Vnode we are traveling to has not already been traversed. Without this clause the function tends to traverse the Y nodes multiple times, resulting in unnecessary wastage of resources.
c. The clause : *nth0(0,Visited,Z)* makes sure that we detect a cycle only when the node we are travelling to is the same as the one we started our traversal with. For example without this clause the path: 1,6,5,9.3,8,5 is also a cycle which is incorrect.