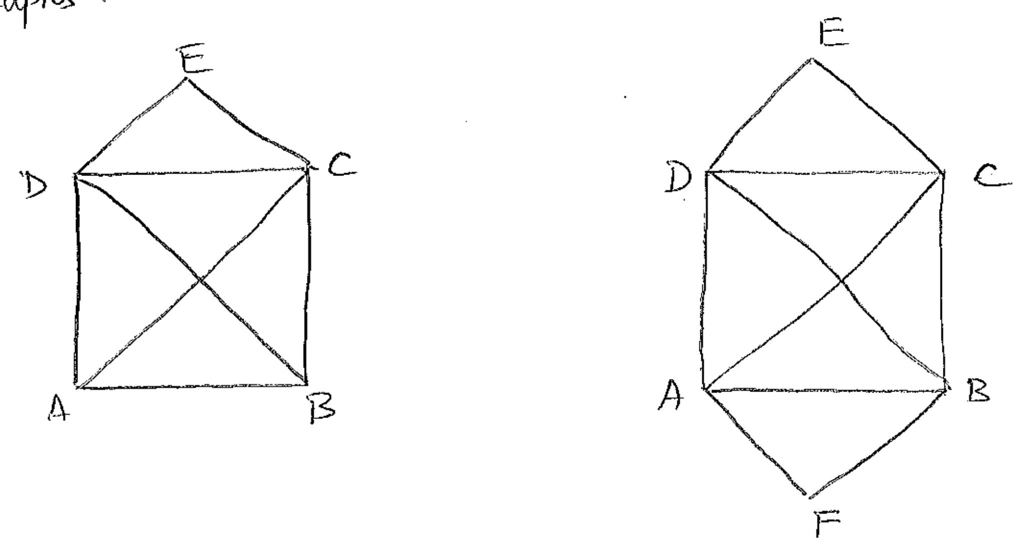
Homework 1

1. Problem Formulation.

Graphs :



The primary data structure that we check here is EdgeVisited.

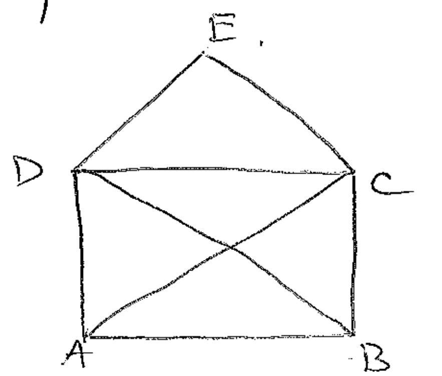
Initial State: EdgeVisited = [ ]

Goal State: EdgeVisited = [(A,B), (B,A), ---- <all edges visited> ---]

At the goal state, we can verify if we reach back to the source node, then we have the cycle; or else we have the path. This calculation can be pre-empted to know what to expect.

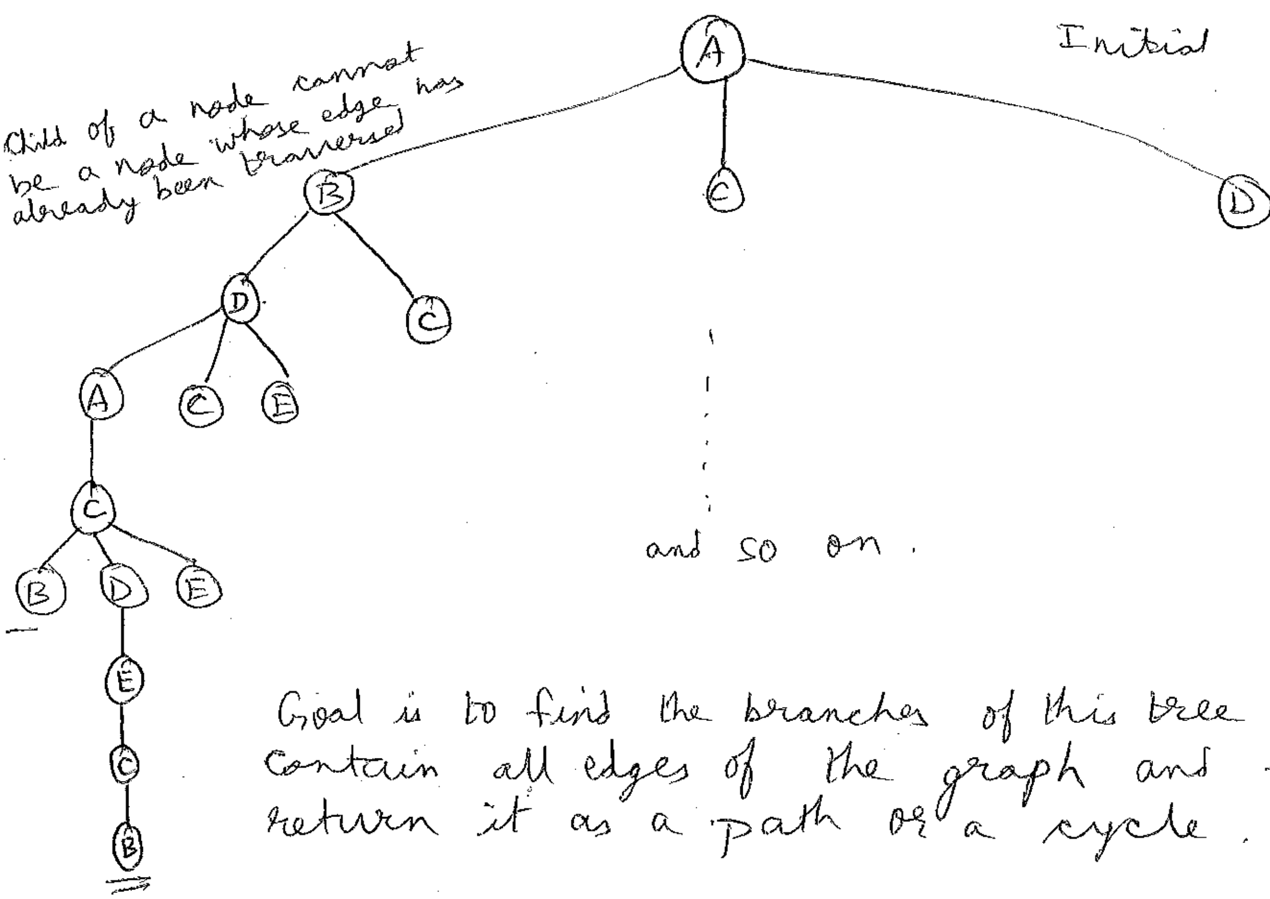
At any arbitrary state : EdgeVisited = [--- (a_i, b_i), (b_i, a_i), ---]
a_i and b_i are the nodes where b_i ~~was~~ is the node just entered from a_i.

We can ~~you~~ use BFS or DFS policy on the search tree for the graphs.
To show an example; consider the first graph :

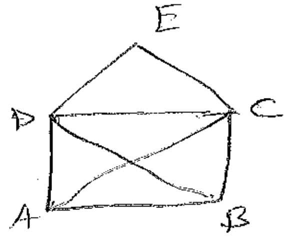


A → B, C, D
B → A, D, C
C → A, B, D, E
D → A, B, C, E
E → C, D

The search tree will look as follows :



Goal is to find the branches of this tree which contain all edges of the graph and finally return it as a path or a cycle.



A sample implementation using DFS would be the following:

Stack	Node_Visited	Edge_Visited	Frequencies	Commands
A	–	–	[3,3,4,4,2]	Push(A)
–	A	–	[3,3,4,4,2]	Pop()
B C D	A	–	[3,3,4,4,2]	Push(A.available Children)
B C	A, D	AD, DA	[2,3,4,3,2]	1. Pop() 2. If node_visited has entry, look for last entry (A) and combine with popped node to form edges and add to visited_edges list.
B C B C E	A, D	~~a~~ AD, DA	[2,3,4,3,2]	Push(D.available Children)

This can go on until Edge_Visited has all ~~[illegible]~~ the edges of the graph, and since the given graphs are undirected, we add both front and back edges to the data structure.