. 10.24.

DATE: PAGE:

LABORATORY PROGRAM - 2(B)

1. Pseudocode for 8 puzzle problem:

```
function FIND-ZERO(state) returns (row, col)
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] == 0 then
                return (i, j)

function MOVE (state, direction) return new_state
    new_state ← copy of state
    (i,j) ← FIND-ZERO(state)

    if direction == "up" and i>0 then
        swap new_state[i][j] with new_state[i-1][j]
    else if direction == "down" and i<2 then
        swap new_state[i][j] with new_state[i+1][j]
    else if direction == "left" and j>0 then
        swap new_state[i][j] with new_state[i][j-1]
    else if direction == "right" and j<2 then
        swap new_state[i][j] with new_state[i][j+1]
return new_state

function IS-GOAL(state) returns boolean
    return state == goal_state

function PRINT-STATE (state)
    for each row in state do
        print row.
```

```
function DFS(initial_state) returns path or failure
    stack ← [(initial_state, [])]
    visited ← empty set
    while not IS-EMPTY(stack) do
        (state, path) ← POP(stack)
        PRINT-STATE(state)

        if IS-GOAL(state) then
            return path

        visited.add(str(state))
        for direction in [up", "down", "left", "right"] do
            new_state ← MOVE(state, direction)
            if new_state is not null and str(new_state) not
              in visited then
                PUSH(stack, (new_state, path + [direction]))
return failure

function GET-INITIAL-STATE() returns initial_state
    print "Enter the initial state of the 8-puzzle
      (0 for empty space):"
    initial_state ← empty list
    for i from 0 to 2 do
        row ← input row as list of integers
        initial_state.append(row)
return initial_state

main
    initial_state ← GET-INITIAL-STATE()
    ~~po~~ PRINT-STATE(initial_state)
    start_time ← current time
    print "Solving using DFS:"
```

```
dfs_solution ← DFS(initial_state)
end_time ← current time
if dfs_solution is not null then
    print "DFS Solution :", dfs_solution.
else
    print "No solution found."
print "Time taken by DFS :", end_time - start_tim
```

2 Pseudocode for iterative deepening search algorithm.

```
class NODE
  function __INIT__(state, parent)
    self.state ← state
    self.parent ← parent

  function PATH() returns list
    node ← self.
    result ← empty list
    while node is not null do
      append node.state to result
      node ← node.parent
  return REVERSE(result)

function I-D-S(problem) returns path
  depth ← 0
  while true do
    print "Exploring depth:", depth.
    (result, _) ← D-F-S(problem, depth)
    if result is not null and result is not "cutoff" then
      return result
    depth ← depth + 1.
```

```
function D-F-S(problem, limit) returns path or "cutoff"
    frontier ← [NODE(problem.initial_state)]
    explored ← empty_set
    cutoff_occurred ← false
    while frontier is not empty do
        node ← POP(frontier)
    if problem.IS-GOAL(node.state) then
        return node.PATH(), explored
    if node.state not in explored then
        explored.add(node.state)
        if LENGTH(node.PATH()) - 1 < limit then
            for child in problem.EXPAND(node.state) do
                APPEND frontier with NODE(child, node)
        else
            cutoff_occurred ← true
return "cutoff" if cutoff_occurred else null, explored

class GRAPH-PROBLEM
    function __INIT__(initial_state, goal_state, adjacency_list)
        self.initial_state ← initial_state
        self.goal_state ← goal_state
        self.adjacency_list ← adjacency_list

    function IS-GOAL(state) returns boolean
        return state == self.goal_state

    function EXPAND(state) returns list
        return [neighbor for neighbor in self.adjacency
            _list.get(state, [])].
```

```
function GET-GRAPH-FROM-INPUT() returns GRAPH-PROBLE
   adjancency_list ← empty_dictionary
   initial_state ← INPUT("Enter the initial state:").strip
   goal_state ← INPUT("Enter the goal state:").strip
   print "Enter the ~~AD~~ list for graph."
   print "Type 'done' when finished."
   while true do
      node ← INPUT("Enter node:").strip()
      if node.lower() == "done" then
         break
     neighbars_input ← INPUT("Enter N of "+Node+"
      separated by spaces:").strip()
     neighbars ← neighbars_input.split()
     adjacency_list[node] ← [neighbar.strip() for
        neighbar in neighbars]
return GRAPH-PROBLEM(initial_state, goal-state,
                      adjacency_list).

main
   problem ← GET-GRAPH-FROM-INPUT()
   solution ← I-D-S(problem)
   if solution is not null then
      print "Solution Path:", solution
   else
      print "No solution found."
```