INDEX

Name Supriya.S Sub. SE AI

Std.: 5~~th~~ sem CSE Div. E Roll No.

Telephone No. E-mail ID.

Blood Group. Birth Day.

Sr.No.	Title	Page No.	Sign./Remarks
1.	Implementation of Tic-Tac-Toe	1	24/9/2024
2.	Implementation of vacuum world cleaner	6	1/10/2024
3.	Solve 8 puzzle problems using DFS and BFS	10	9/10/2024
~~4.~~	~~Implement depth limited search algorithm and uniform cost search algorithm.~~	~~26~~	
4.5	Implement Iterative deepening search algorithm	23	
5.~~6~~	8-Puzzle problem using Astar, and heuristic approach	15	15/10/2024
6.	Implement simulatedannealing	~~26~~ 29	
7.	Implement Hill climbing algorithm	26	29/10/2024
8.	Implementation of knowledge base		13/11/2024

CS

24-9-24

Algorithm for tic-tac-toe implementation

Algorithm: Print_Board(board)
// to print tic-tac-toe board
for row in board:
    print row
print()// for next line

Is_filled(board):
// checks whether the board is filled
for row in board
    for cell in row
        if cell == '_'
            return False
return True

Check_winner(board):
for i in range(3):
    if board[i] == ['X','X','X']
        or [board[j][i] for j in range(3)
        == ['X','X','X']:
        return 'X'
    if board[i] == ['O','O','O']
        or [board[j][i] for j in range(3)
            == ['O','O','O']:
        return 'O'
    if [board[i][i] == ['X' 'X','X']
        or [board[i][2-i] for i in
            range(3)] == ['X','X','X']
    return 'X'
~~if [board~~
// same for 'O'

```
    return None

Is_full(board):
    for row in board
        for cell in row
            if cell != '_'
                return False
    return True

Start_game():
current_player = input("Enter : ").
                                upper()
while not Is_full(board):
// print board
Enter row and column number
if 0 <= row <3 and 0 <= col <3
        and board[row][col] == '_':
    board[row][col] = current_
                                player
    winner = check_winner(board)
    if winner:
        print_board(board)
        print winner
        return
    current_player = 'O'
    if current_player == 'X'
                        else 'X'
else:
print("Invalid")
print_board(board)
print("Game Draw")
```

24/9/2024

O/P:

1) Enter first player ('X' or 'O'): O
- - -
- - -
- - -
O's turn. Enter row (0-2): 0
O's turn. Enter col (0-2): 0
O - -
- - -
- - -
~~O's~~
X's turn. Enter row (0-2): 0
X's turn. Enter col (0-2): 1
O X -
- - -
- - -
O's turn. Enter row (0-2): 1
O's turn. Enter col (0-2): 0
O X -
O - -
- - -
X's turn. Enter row (0-2): 1
X's turn. Enter col (0-2): 1
O X -
O X -
- - -
O's turn. Enter row (0-2): 2
O's turn. Enter col (0-2): 0
X O - O X -
X O - O X -
X - - O - -
O is the winner

2) Enter first player ('X' or 'O'): O

- - -

- - -

- - -

O's turn. Enter row (0-2): 1

O's turn. Enter col (0-2): 1

- - -

- O -

- - -

X's turn. Enter row (0-2): 0

X's turn. Enter col (0-2): 0

X - -

- O -

- - -

O's turn. Enter row (0-2): 1

O's turn. Enter col (0-2): 0

X - -

O O -

- - -

X's turn. Enter row (0-2): 1

X's turn. Enter col (0-2): 2

X - -

O O X

- - -

O's turn. Enter row (0-2): 0

O's turn. Enter col (0-2): 1

X O -

O O X

- - -

X's turn. Enter row (0-2): 2

X's turn. Enter col (0-2): 1

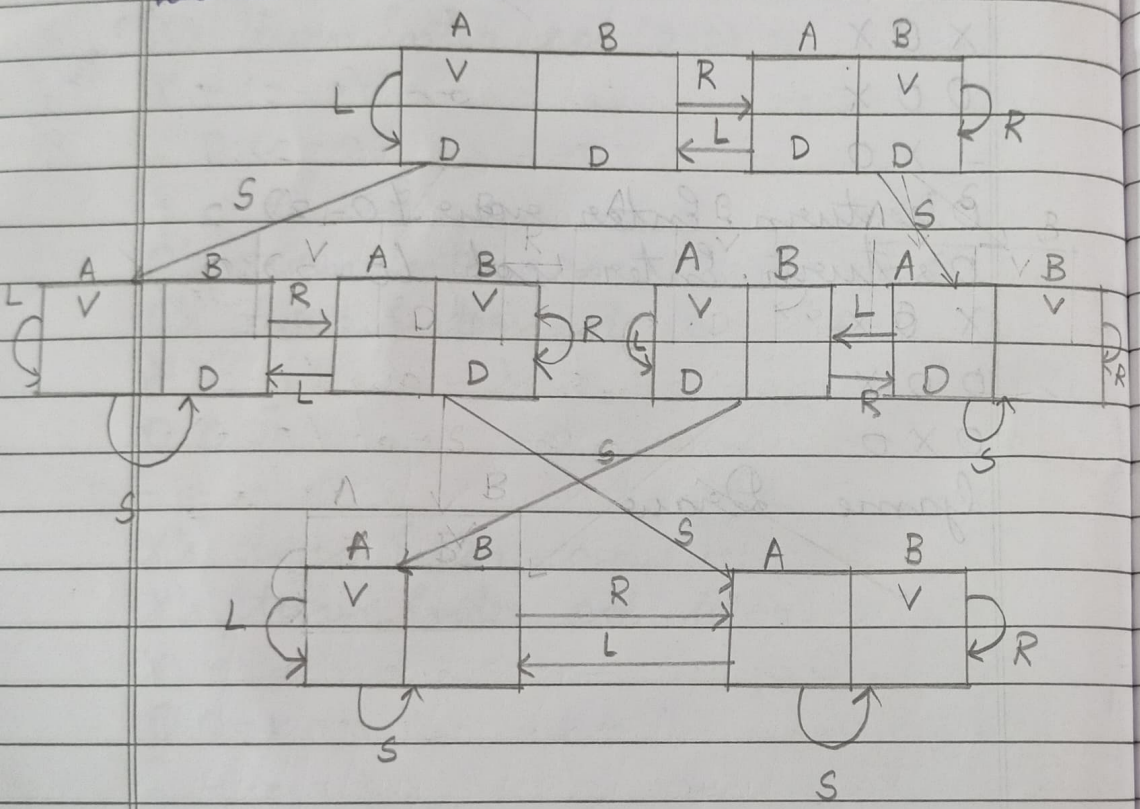
X O -
O O X
- X -
O's turn. Enter row (0-2) : 2
O's turn. Enter col (0-2) : 2
X O -
O O X
- X O
X's turn. Enter row (0-2) : 0
X's turn. Enter col (0-2) : 2
X O X
O O X
- X O
O's turn. Enter row (0-2) : 2
O's turn. Enter col (0-2) : 0
X O X
O O X
O X O
Game Draw

1-10-24

Implement Vacuum World Cleaner

function REFLEX-VACUUM-AGENT ([location, status]) returns an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left

The states space diagram of vacuum world cleaner.



Parameters to ~~the~~ input :-
location, status, otherroomstatus.

Problem Formulation Steps
1. States
2. Initial state
3. Actions
4. Transition model

5. Goal Test
6. Path cost

Algorithm for Two Quaderants

1. Declare goal = [ ]
2. Input room location, status of room location (A or B) as 0 for clean and 1 for dirty and other room status
3. cost ← 0
4. if room location == 'A'
   goal [1] ← status
   goal [3] ← other room
   // display initial goal state
   print "Vacuum is placed in location A"
   if status == 0
   print "Location A is already clean"
   if status == 1
   print "Location A is Dirty"
   cost ← cost + 1
   goal [1] ← 0
   print "Location A has been cleaned"
   print "Moving right to location B"
   print "Vacuum is placed in location B"
   if other room == 1
   print "Location B is dirty"
   cost ← cost + 1
   goal [3] ← 0
   print "Location B has been cleaned"

else
print "Location B is already clean"

//display final goal state and cost for suck
elif "room location == 'B':
goal[3] ← status
goal[1] ← other room
//display initial goal state
//display "Vacuum is placed in location B"
if status == 0
print "Location B is already clean"
if status == 1
print "Location B is Dirty"
cost ← cost + 1
goal[3] ← 0
print "Location B has been cleaned"
print "Moving left to Location A"
print "Vacuum is placed in Location A"
if other room == 1
print "Location A is Dirty"
cost ← cost + 1
goal[1] ← 0
print "Location A has been cleaned"
if other room == 0
print("Location A is already clean")
//display final goal state and cost for suck
else
//display "Invalid input"
<end of if>

5. Add battery level and check if its greater than zero else print "Battery low", decrement it each time

Output:

1. Enter the room location A or B : B
Enter status of the room 0 for clean or 1 for dirty : 1
Enter other room status 0 or 1 : 0
Initial goal state ['A', 0, 'B', 1]
Vacuum is placed in location B
Location B is Dirty
Location B has been cleaned
Moving left to location A
Vacuum is placed in location A
Location A is already clean
Goal State is ['A', 0, 'B', 0]
Cost for suck is 1

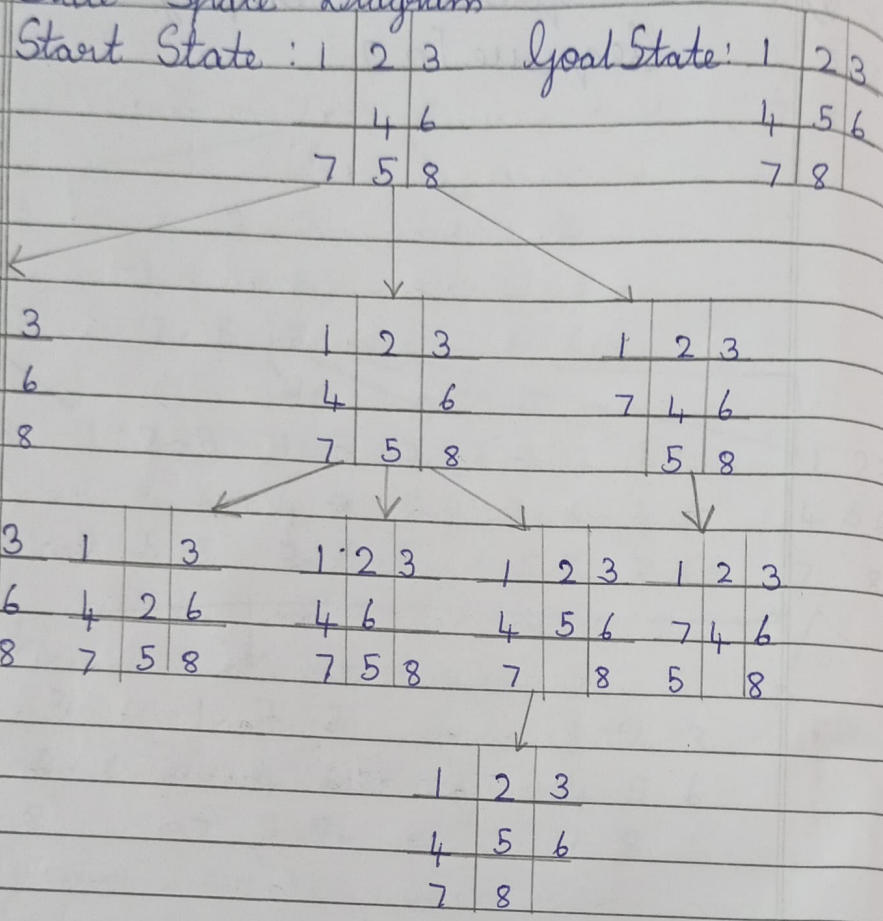
1/10/2024

2. Enter the room location A or B : A
Enter status of the room 0 for clean or 1 for dirty : 0
Enter other room status 0 or 1 : 0
Initial goal state ['A', 0, 'B', 0]
Vacuum is placed in location A
Location A is already clean
Moving left to location B
Location B is already clean
Goal State is ['A', 0, 'B', 0]
Cost for suck is 0

8-10-24

Implement 8-puzzle problem Using non-heuristic method Using BFS and DFS method

State Space Diagram



Algorithm for using BFS method

```
class BoardNode:
    initialize(board, parent = None, move = None
    is_goal(goal_state)
    __hash__()
    __eq__(other)
    __repr__()
```

```
class BFS:
    initialize (initial board, goal state)
    search():
        visited = empty set
        queue = queue containing initial node
        solutions = empty list
        while queue is not empty:
            node = queue.popleft()
            if node is goal state:
                path = get solution path (node)
                append path to solutions
                continue
            add node to visited
            for each neighbor in get successors (node):
                if neighbor not in visited
                and neighbor not in queue:
                    queue.append(neighbor)

        for each solution in solutions:
            print (solution)
        print (total number of solutions)
    get successors(node):
        locate zero (empty tile) in board
        for each move in [UP, DOWN,
                          LEFT, RIGHT]:
            calculate new position
            if new position is valid:
                create new board by
                    swapping tiles
                append new BoardNode to
                    successors
```

```
find_zero(board):
    for each cell in board:
        if cell is zero:
            return (row, column)
get_solution_path(node):
    path = empty list
    while node.parent is not None:
        append node.move to path
        node = node.parent
    return reversed(path)
function get_user_input(board type):
    print("Enter 3x3 board:")
    return board configuration

main():
    initial_board = get_user_input("initial")
    goal_state = get_user_input("goal")
    bfs_solver = BFS(initial_board, goal_state)
    bfs_solver.search()
```

O/P:-

Enter the initial 8-puzzle board (3x3)

1 2 3
0 4 6
7 5 8

Enter the goal 8-puzzle board

1 2 3
4 5 6
7 8 0

Solution: 1
Moves: ['RIGHT', 'DOWN', 'RIGHT']
Final Board State:
1 2 3
4 5 6
7 8 0
Number of moves: 3

Algorithm for using DFS method
class BoardNode:
initialize (board, parent = None, move = None)
is_goal (goal_state)
__hash__()
__eq__(other)
__repr__()
class DFS:
initialize (initial_board, goal_state)
search():
visited = empty set
call dfs (initial_node, visited)
if solutions found:
print solutions
print total unique states, total permutations
_dfs (node, visited):
if node is goal state:
append solution path to found solutions
return True
mark node as visited
add current board to unique states
increment permutation count

```
    for each successor in get_successors(node):
        if successor not in visited:
            call dfs(successor, visited)
get_successors(node):
    for each move in [UP, DOWN, LEFT, RIGHT]:
        if valid move:
            create new board
            append new BoardNode (new_board, node, move) to successors
find_zero(board):
    return coordinates of zero
get_solution_path(node):
    while node has a parent:
        append move to path
        move to parent node
    return reversed(path)
print_solution(path):
    print solution path
function get_user_input():
    read & return 3x3 board
main():
    get initial & goal states
```

O/P:

Enter initial state: 1 2 3
0 4 6
7 5 8

Enter goal state: 1 2 3
4 5 6
7 8 0

Total unique states encountered: 852

15-10-24

Lab Program No. 3

For 8 Puzzle problem using A star implementation to calculate f(n) using a) g(n) = depth of a node
h(n) = heuristic value (no. of misplaced tiles)
f(n) = g(n) + h(n)

b) g(n) = depth of a node
h(n) = heuristic value (manhattan distance)

a) No. of misplaced tiles

Draw the states space diagram for

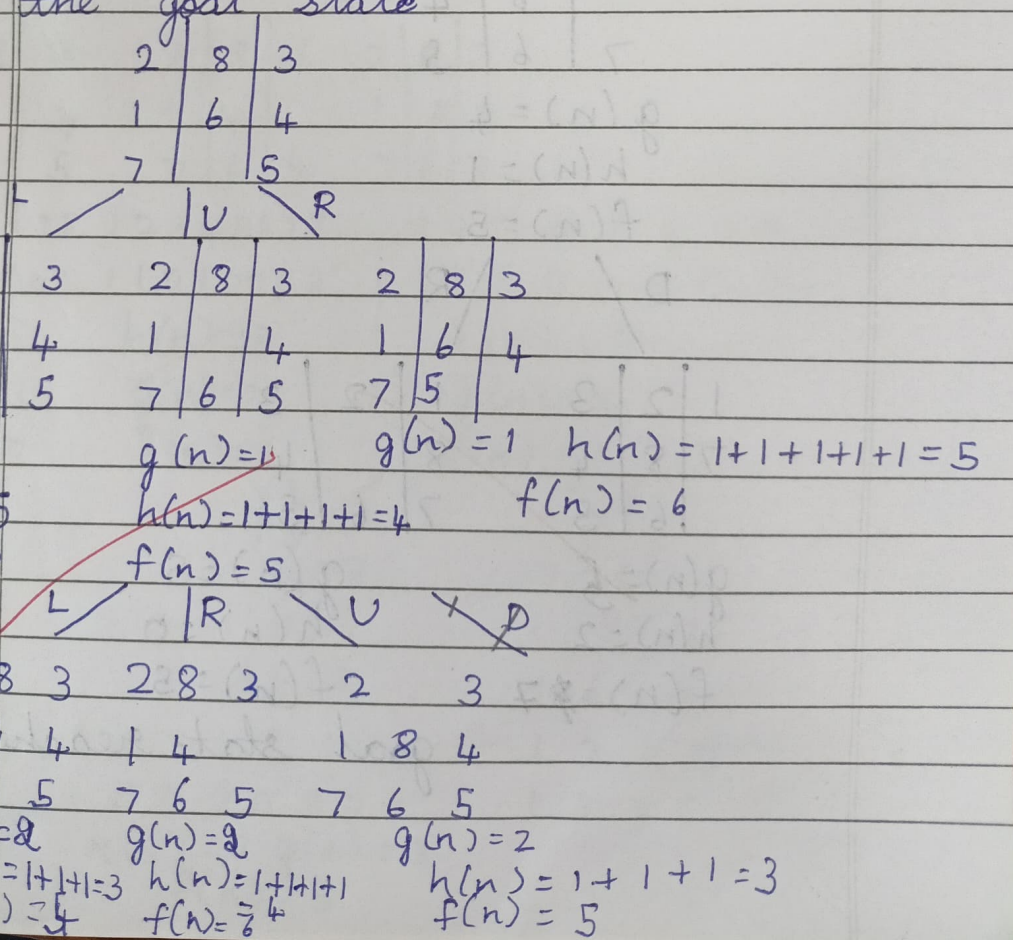
2	8	3
1	6	4
7		5

Initial

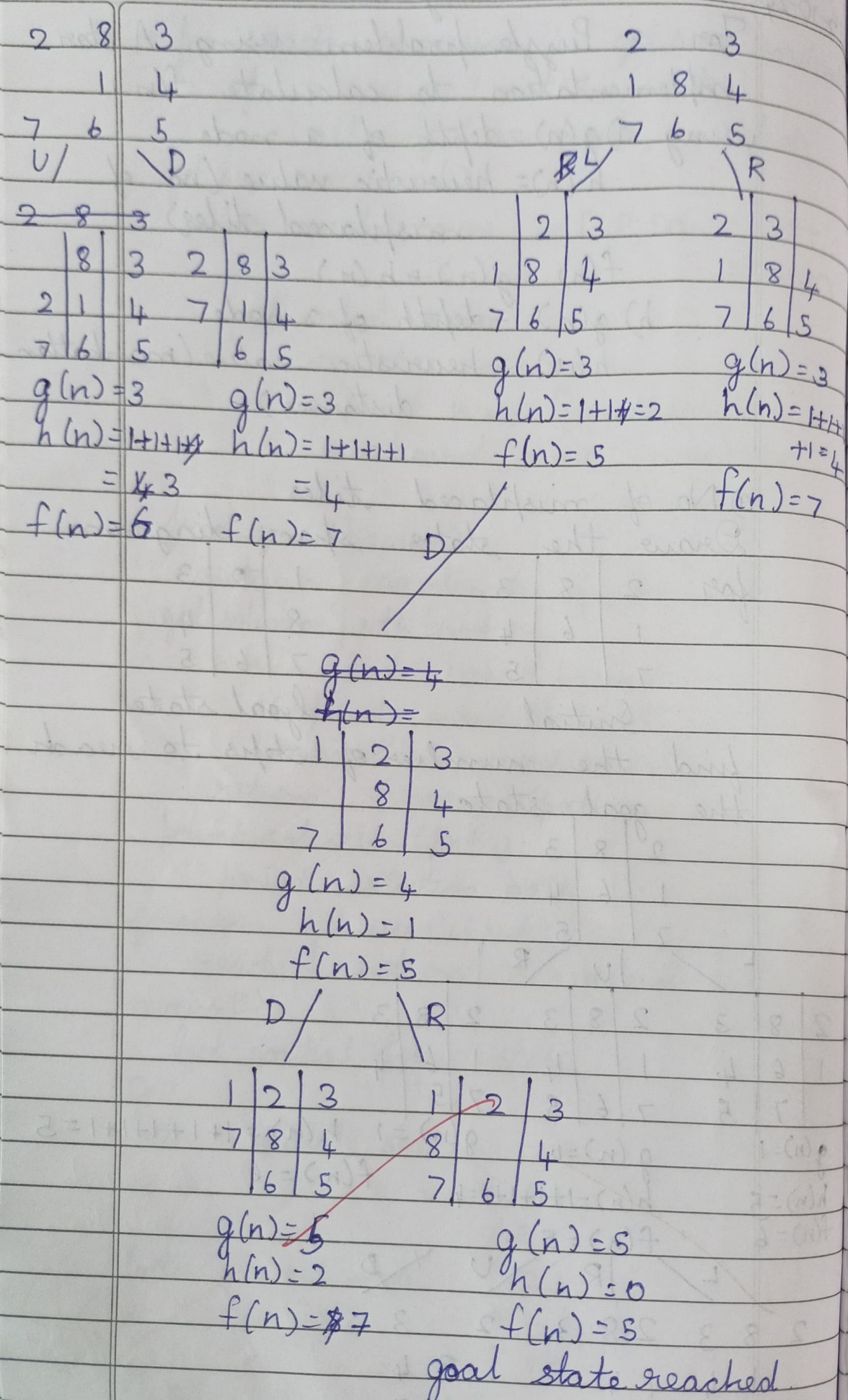
1	2	3
8		4
7	6	5

Goal state

find the number of steps to reach the goal state



Date Page


2 8 3
1 4
7 6 5
U/ \D
g(n)=3
h(n)=1+1+1+1
=4
f(n)=6
g(n)=3
h(n)=1+1+1+1
=4
f(n)=7
2 3
1 8 4
7 6 5
\R
g(n)=3
h(n)=1+1+1=2
f(n)=5
g(n)=3
h(n)=1+1+1+1=4
f(n)=7
D
g(n)=4
h(n)=
g(n)=4
h(n)=1
f(n)=5
D/ \R
g(n)=5
h(n)=2
f(n)=7
g(n)=5
h(n)=0
f(n)=5
goal state reached

b) Using Manhattan Distance

2	8	3
1	6	4
7		5

Initial

1	2	3
8		4
7	6	5

Final

2	8	3
1	6	4
7		5

U L R

2	8	3
1		4
7	6	5

2	8	3
1	6	4
7	5	

2	8	3
1	6	4
7	5	

1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 2

$d = h(n) = 4$
$g(n) = 1$
$f(n) = 5$

1 2 3 4 5 6 7 8
1 1 0 0 0 1 1 2

$h(n) = 6$
$g(n) = 1$
$f(n) = 7$

1 2 3 4 5 6 7 8
1 1 0 0 1 1 0 2

$h(n) = 6$
$g(n) = 1$
$f(n) = 7$

L R ~~D~~ U

2	8	3
	1	4
7	6	5

2	8	3
1	4	
7	6	5

2	8	3
1	8	4
7	6	5

1 2 3 4 5 6 7 8
1 2 0 0 0 0 0 2

$h(n) = 5$
$g(n) = 2$
$f(n) = 7$

1 2 3 4 5 6 7 8
1 1 0 1 0 0 0 2

$h(n) = 5$
$g(n) = 2$
$f(n) = 7$

1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 1

$h(n) = 3$
$g(n) = 2$
$f(n) = 5$

L R

	2	3
1	8	4
7	6	5

2	3	
1	8	4
7	6	5

1 2 3 4 5 6 7 8
1 0 0 0 0 0 0 1

$h(n) = 2$ $g(n) = 3$
$f(n) = 5$

1 2 3 4 5 6 7 8
1 1 1 0 0 0 0 1

$h(n) = 4$ $g(n) = 3$
$f(n) = 7$

	2	3
1	8	4
7	6	5

D ✗

1	2	3
	8	4
7	6	5

$h(n) = 1$ $g(n) = 4$
$f(n) = 5$

1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	1

L D

1	2	3
8		4
7	6	5

$h(n) = 0$
$g(n) = 5$
$f(n) = 5$

1	2	3
7	8	4
	6	5

$h(n) = 2$
$g(n) = 5$
$f(n) = 7$

goal state reached

a) No. of misplaced tiles

Algorithm: Astarmisplacedtiles (start, goal)
// to find the no. of misplaced tiles
h = 0
for i in range(3):
    for j in range(3):
        if (start[i][j] != goal[i][j])
            h = h+1
return h-1

Algorithm: main()
//Input start[][] and goal[][]
g = 0
visited = []
f = 999
while (start != goal):
    if start not in visited:
        g = g+1
        emptytile = locateemptytile(start)
        for each move, the emptytile UP, DOWN, RIGHT, LEFT and update in start,
        h = Astarmisplacedtiles (start, goal)
        if ((g+h) < f):
            f = g+h
            visited.append(start[][])
// print f

Algorithm: locateemptytile (start)
    for i in range (len(start)):
        if start[i][j] == 0:
            return i, j

b) Manhattan distance

Algorithm: Manhattan distance (start, goal)

```
h=0
for i in [1, 2, 3, 4, 5, 6, 7, 8]:
    a = indexof
    store index of a i in start and store in a
    store index of j in goal and store in b
    h = h + abs(a - b)
    return h
return h
```

Algorithm: main()

```
// Input start and goal
g=0
visited = []
f=999
while (start != goal):
    if start not in visited
        g = g+1
        emptytile = locateemptytile(start)
        for each move UP, DOWN, LEFT, RIGHT, update in start
            h = Astarmisplacedtiles(start, goal)
            if ((g+h) < f)
                f = g+h
                visited.append(start[][])
// printf
```

Algorithm

SURYA Gold

Date Page

Output

a) Enter initial state (3 x 3 grid, use 0 for empty tile):

2 8 3
1 6 4
7 0 5

Enter goal state (3 x 3 grid, use 0 for empty tile):

1 2 3
8 0 4
7 6 5

Level: 1, Direction: Up, Heuristic: 3
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Level: 1, Direction: Left, Heuristic: 5
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Level: 1, Direction: Right, Heuristic: 5
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Level: 2, Direction: Up, Heuristic: 3
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Level: 2, Direction: Left, Heuristic: 3
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]

22-10

Level : 3, Direction: Left, Heuristic: 2
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Level: 4, Direction : Down, Heuristic: 1
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Level: 5, Direction : Right, Heuristic: 0
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Goal state reached