INTELLIGENT SYSTEMS
ITCS 6150/8150, Fall 2023

PROJECT 1

Solving the 8-puzzle using A* algorithm

Submitted By:

Nikhila Chitneni	801315064
Saran Sai Chava	801308389
Steffy Roselina Eben Judson	801306459

Submitted To:

Dr. Dewan T. Ahmed

TABLE OF CONTENTS

1. ABSTRACT

The classic 8-puzzle problem involves a 3x3 grid with eight numbered tiles and one empty space. The challenge is to rearrange these tiles to achieve a goal state configuration. The objective is to manipulate these tiles within the grid to ultimately achieve the goal arrangement. This report explains the application of the A* search algorithm to effectively address and solve the 8-puzzle problem.

.

2. INTRODUCTION

The A* search algorithm is used to determine the best series of actions to take from the starting point to the desired goal state. The algorithm continuously explores many states, keeps both open and closed lists, and prioritizes states with the best g-cost and h-cost. The value of f(n) is the sum of path cost and heuristic cost. $f(n) = g(n) + h(n)$ g(n) is the cost of the current path. h(n) is a heuristic function that finds the shortest route from current node to the target node.

3. ALGORITHM

3.1 DESCRIPTION :

First , we start by expanding the initial state ie., generating every possible move from that state and place these new nodes on the frontier . We compute heuristics and path cost , h(n) and g(n) respectively for the nodes on the frontier. Being a greedy algorithm , A * expands the node with least f(n) = g(n) + h(n). This is done recursively , excluding the duplicate nodes as we progress and terminates when we reach the goal state.

3.2 HEURISTICS :

Misplaced Tile Heuristic :

In this method we count the number of misplaced tiles as h(n). A * search algorithm chooses the node with lowest f(n). So we have a good chance of expanding the node with fewer misplaced tiles .

Manhattan Distance Heuristic :

In this method instead of just counting the misplaced tiles , we count the number of tiles if each misplaced tile is away from its position in the goal state. So we try to choose for expansion , the node that has fewer steps leading towards the goal state.

4. IMPLEMENTATION

GLOBAL VARIABLES
(i) initial_board - input given by user; contains 3 x 3 grid of values in initial state.
(ii) goal - expected output specified by user. 3 x 3 grid of values in expected goal state

METHODS

manhattan_distance(initial_board, goal_board) -> int: This function calculates the Manhattan distance heuristic between the initial_board and the goal_board. It measures the total number of moves (both horizontal and vertical) required to move each tile from its current position in the initial_board to its correct position in the goal_board. The function returns an integer representing the Manhattan distance.

count_misplaced(initial_board, goal_board) -> int: This function calculates the Misplaced Count heuristic between the initial_board and the goal_board. It counts the number of tiles in the initial_board that are not in their correct positions in the goal_board. The function returns an integer representing the count of misplaced tiles.

boardGood(board) -> bool: This function checks if the board is a valid 8-puzzle board. It ensures that the board contains the numbers 1 to 8 (or 0 for the blank tile) exactly once. It returns True if the board is valid, and False otherwise.

showBoard(board: list[list[int]]) -> None: This function displays the content of the board in a 3x3 format, printing each element row by row.

replaceEmptySpace(board: list[list[int]]) -> list[list[int]]: This function replaces any empty spaces in the board (represented as spaces) with '0' to make it consistent for processing.

findElement(board: list[list[int]], ele) -> list[int]: This function finds the indices of a given element ele in the board and returns them as a list of two integers [row, column].

findEmptySpace(board: list[list[int]]) -> list[int]: This function finds the indices of the blank (0) tile in the board and returns them as a list of two integers [row, column].

cloneBoard(board: list[list[int]]) -> list[list[int]]: This function creates a deep copy of the input board and returns the copy.

goalTest(board: list[list[int]], goal_board: list[list[int]]) -> bool: This function checks if the board matches the goal_board, indicating whether the puzzle has been solved. It returns True if the board is equal to the goal_board, and False otherwise.

attachManhattan(boards: list[list[list[int]]], goal: list[list[int]], count: int) -> list: This function calculates the Manhattan distance for a list of boards and attaches it to each board along with the current count. It returns a list of tuples, where each tuple contains the Manhattan distance plus the count and the corresponding board.

attachMisplaced(boards: list[list[list[int]]], goal: list[list[int]], count: int) -> list: Similar to attachManhattan, this function calculates the Misplaced Count heuristic for a list of boards and attaches it to each board along with the current count. It returns a list of tuples, where each tuple contains the Misplaced Count plus the count and the corresponding board.

actionList(board, prev_boards): This function generates a list of successor boards that can be reached from the given board by making valid moves. It considers moving the blank tile (0) up, down, left, or right. It ensures that the generated boards are not duplicates of previously visited boards (in prev_boards).

aStarSearchManhattan(initial_board, goal_board) -> list: This function performs an A* search using the Manhattan distance heuristic to find the solution to the 8-puzzle problem. It returns a list of tuples, where each tuple contains a board and the corresponding count of moves.

aStarSearchMisplaced(initial_board, goal_board) -> list: Similar to aStarSearchManhattan, this function performs an A* search using the Misplaced Count heuristic to find the solution to the 8-puzzle problem. It also returns a list of tuples, where each tuple contains a board and the corresponding count of moves.

These functions collectively implement the 8-puzzle problem-solving algorithm using two different heuristic methods. The program allows users to input initial and goal states and then selects the desired heuristic to find the solution to the puzzle.

```python
def showBoard(board:list[list[int]])->None:
  for i in range(3):
    for j in range(3):
      print(board[i][j],end=" ")
    print()
  print()

#replaces empty space " ", with zero
def replaceEmptySpace(board:list[list[int]])->list[list[int]]:
  for i in range(3):
    for j in range(3):
      if str(board[i][j]).isspace():
        board[i][j]='0'
  return board

def findElement(board:list[list[int]],ele)->list[int]:
  for i in range(3):
    for j in range(3):
      if board[i][j]==ele:
        return [i,j]

def findEmptySpace(board:list[list[int]])->list[int]:
  for i in range(3):
    for j in range(3):
      if str(board[i][j]).isspace() or board[i][j]=='0':
        return [i,j]

  raise Exception("No space left in board, made an illegal move")
  return [0,0]

def cloneBoard(board:list[list[int]])->list[list[int]]:
  newboard=[[0,0,0],[0,0,0],[0,0,0]]
  for i in range(3):
    for j in range(3):
      newboard[i][j]=board[i][j]

  return newboard

def goalTest(board:list[list[int]],goal_board:list[list[int]])->bool:
  for i in range(3):
    for j in range(3):
      if goal_board[i][j]!=board[i][j]:
        return False

  return True

def attachManhattan(boards:list[list[list[int]]],goal:list[list[int]],count:int)->list:
```


```python
def cloneBoard(board:list[list[int]])->list[list[int]]:
  newboard=[[0,0,0],[0,0,0],[0,0,0]]
  for i in range(3):
    for j in range(3):
      newboard[i][j]=board[i][j]

  return newboard

def goalTest(board:list[list[int]],goal_board:list[list[int]])->bool:
  for i in range(3):
    for j in range(3):
      if goal_board[i][j]!=board[i][j]:
        return False

  return True
```


```python
def attachManhattan(boards:list[list[list[int]]],goal:list[list[int]],count:int)->list:
  d=[]

  for i in boards:
    d.append((manhattan_distance(i,goal) + count,i))

  return d

def attachMisplaced(boards:list[list[list[int]]],goal:list[list[int]],count:int)->list:
  d=[]

  for i in boards:
    d.append((count_misplaced(i,goal) + count,i))

  return d


def actionList(board,prev_boards):
  boards=[]
  x,y = findEmptySpace(board)

  #Check if there is a legal move in the direction - up
  if x>0:
    mvUpBoard = cloneBoard(board)
    mvUpBoard[x][y] = mvUpBoard[x-1][y]
    mvUpBoard[x-1][y] = '0'

    if mvUpBoard not in prev_boards:
      boards.append(mvUpBoard)
  #Check if there is a legal move in the direction - left
  if y>0:
```

main.py M × ⓘ Readme U

main.py > manhattan_distance

You, 6 minutes ago | 1 author (You)

```python
from queue import PriorityQueue

# Manhattan distance
def manhattan_distance(initial_board, goal_board)-> int:

  distance = 0

  for i in range(3):
    for j in range(3):
      You, 18 hours ago • Init+heuristics
      if initial_board[i][j] != '0':

        # Calculate Manhattan distance
        x_diff = abs(j - findElement(goal_board,initial_board[i][j])[0])
        y_diff = abs(i - initial_board.index(initial_board[i]))
        distance += x_diff + y_diff

  return distance


# Misplaced count
def count_misplaced(initial_board, goal_board)-> int:

  count = 0

  for i in range(3):
    for j in range(3):

      if initial_board[i][j] != '0':

        if initial_board[i][j] != goal_board[i][j]:
          count += 1

  return count

#Test statements for heuristics
#initial_board = [[1, 2, 3], [4, 5, 6],[0, 7, 8]]

#goal_board = [[1, 2, 3],[4, 5, 6],[7, 8, 0]]

#print(manhattan_distance(initial_board, goal_board))
#print(count_misplaced(initial_board, goal_board))
def boardGood(board)->bool:
  l=['1','2','3','4','5','6','7','8','9','0']
  l1=[]
  for i in range(3):
    for j in range(3):
      l1.append(board[i][j])
```

```python
def actionList(board,prev_boards):
  boards=[]
  x,y = findEmptySpace(board)

  #Check if there is a legal move in the direction - up
  if x>0:
    mvUpBoard = cloneBoard(board)
    mvUpBoard[x][y] = mvUpBoard[x-1][y]
    mvUpBoard[x-1][y] = '0'

    if mvUpBoard not in prev_boards:
      boards.append(mvUpBoard)
  #Check if there is a legal move in the direction - left
  if y>0:
    mvleftBoard = cloneBoard(board)
    mvleftBoard[x][y] = mvleftBoard[x][y-1]
    mvleftBoard[x][y-1] = '0'
    if mvleftBoard not in prev_boards:
      boards.append(mvleftBoard)
  #Check if there is a legal move in the direction - right
  if y<2:
    mvrightBoard = cloneBoard(board)
    mvrightBoard[x][y] = mvrightBoard[x][y+1]
    mvrightBoard[x][y+1] = '0'
    if mvrightBoard not in prev_boards:
      boards.append(mvrightBoard)
  #Check if there is a legal move in the direction - down
  if x<2:
    mvdownBoard = cloneBoard(board)
    mvdownBoard[x][y] = mvdownBoard[x+1][y]
    mvdownBoard[x+1][y] = '0'
    if mvdownBoard not in prev_boards:
      boards.append(mvdownBoard)

  return boards

def aStarSearchManhattan(initial_board, goal_board)->list:
  path = []
  fringe=[]
  visited=[]
  count=0
  fringe = PriorityQueue()
  fringe.put((0,initial_board))
  while not fringe.empty():
    curr_board=fringe.get_nowait()[1]
    if curr_board in visited:
      pass
    if boardGood(curr_board):
```

```python
def aStarSearchManhattan(initial_board, goal_board)->list:
  path = []
  fringe=[]
  visited=[]
  count=0
  fringe = PriorityQueue()
  fringe.put((0,initial_board))
  while not fringe.empty():
    curr_board=fringe.get_nowait()[1]
    if curr_board in visited:
      pass
    if boardGood(curr_board):
      raise Exception(" Bad board")

    visited.append(curr_board)
    if goalTest(curr_board,goal_board):
      path.append((curr_board,count))
      return path
    prospectiveBoards = actionList(curr_board,visited)
    d = attachManhattan(prospectiveBoards,goal_board,count+1)
    #updating the fringe, if board/node already exists, check if we have a better heuristic to update it in the frontier
    for key in d:
      fringe.put(key)
    path.append((curr_board,count))
    count+=1
  return path


def aStarSearchMisplaced(initial_board, goal_board)->list:
  path = []
  fringe=[]
  visited=[]
  count=0
  fringe = PriorityQueue()
  fringe.put((0,initial_board))
  while not fringe.empty():
    curr_board=fringe.get_nowait()[1]
    if curr_board in visited:
      pass
    if boardGood(curr_board):
      raise Exception(" Bad board")

    visited.append(curr_board)
    if goalTest(curr_board,goal_board):
      path.append((curr_board,count))
      return path
    prospectiveBoards = actionList(curr_board,visited)
    d = attachMisplaced(prospectiveBoards,goal_board,count+1)
```

```python
def aStarSearchMisplaced(initial_board, goal_board)->list:
  path = []
  fringe=[]
  visited=[]
  count=0
  fringe = PriorityQueue()
  fringe.put((0,initial_board))
  while not fringe.empty():
    curr_board=fringe.get_nowait()[1]
    if curr_board in visited:
      pass
    if boardGood(curr_board):
      raise Exception(" Bad board")

    visited.append(curr_board)
    if goalTest(curr_board,goal_board):
      path.append((curr_board,count))
      return path
    prospectiveBoards = actionList(curr_board,visited)
    d = attachMisplaced(prospectiveBoards,goal_board,count+1)
    #updating the fringe, if board/node already exists, check if we have a better heuristic to update it in the frontier
    for key in d:
      fringe.put(key)
    path.append((curr_board,count))
    count+=1
  return path
```

```python
print("###### Welcome to 8 Puzzle Solver ######")

# Input for the initial state
print(f"Enter Initial State Board.")
init_board = []
for i in range(3):
        row = input().split()
        if len(row) != 3:
            print("Invalid input. Enter 3 elements for each row.")
        init_board.append(row)

# Input for the goal state
print(f"Enter Goal State Board.")      You, 1 second ago • Uncommitted changes
goal = []
for i in range(3):
        row = input().split()
        if len(row) != 3:
            print("Invalid input. Enter 3 elements for each row.")
        goal.append(row)
print("Misplaced")
print(aStarSearchMisplaced(init_board,goal))

print("Manhattan")

print(aStarSearchManhattan(init_board,goal))
```

5. RESULTS

<table>
<tr><th>INITIAL</th><th>GOAL</th><th>Misplaced tiles heuristic</th><th>Manhattan distance heuristic</th></tr>
<tr><td>Example 1<table><tr><td>4</td><td>1</td><td>3</td></tr><tr><td></td><td>2</td><td>5</td></tr><tr><td>7</td><td>8</td><td>6</td></tr></table></td><td><table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td></td></tr></table></td><td>12 nodes generated

6 nodes expanded

Solution at depth 5</td><td>12 nodes generated

6 nodes expanded

Solution at depth 5</td></tr>
<tr><td>Example 2<table><tr><td>7</td><td>1</td><td>8</td></tr><tr><td></td><td>6</td><td>2</td></tr><tr><td>5</td><td>4</td><td>3</td></tr></table></td><td><table><tr><td>7</td><td>8</td><td></td></tr><tr><td>6</td><td>1</td><td>2</td></tr><tr><td>5</td><td>4</td><td>3</td></tr></table></td><td>9 nodes generated

4 nodes expanded

Solution at depth 3</td><td>9 nodes generated

4 nodes expanded

Solution at depth 3</td></tr>
<tr><td>Example 3<table><tr><td>1</td><td>0</td><td>3</td></tr><tr><td>4</td><td>2</td><td>6</td></tr><tr><td>7</td><td>5</td><td>8</td></tr></table></td><td><table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td></td></tr></table></td><td>9 nodes generated

4 nodes expanded

Solution at depth 3</td><td>9 nodes generated

4 nodes expanded

Solution at depth 3</td></tr>
<tr><td>Example 4<table><tr><td>4</td><td>6</td><td>8</td></tr><tr><td>2</td><td>5</td><td>3</td></tr><tr><td></td><td>7</td><td>1</td></tr></table></td><td><table><tr><td>4</td><td>8</td><td>0</td></tr><tr><td>2</td><td>6</td><td>3</td></tr><tr><td>7</td><td>5</td><td>1</td></tr></table></td><td>10 nodes generated

5 nodes expanded

Solution at depth 4</td><td>10 nodes generated

5 nodes expanded

Solution at depth 4</td></tr>
<tr><td>Example 5<table><tr><td>8</td><td>7</td><td>6</td></tr><tr><td>2</td><td>5</td><td>4</td></tr><tr><td></td><td>1</td><td>3</td></tr></table></td><td><table><tr><td>8</td><td>7</td><td>6</td></tr><tr><td>5</td><td>4</td><td>3</td></tr><tr><td>2</td><td>1</td><td></td></tr></table></td><td>10 nodes generated

5 nodes expanded

Solution at depth 4</td><td>10 nodes generated

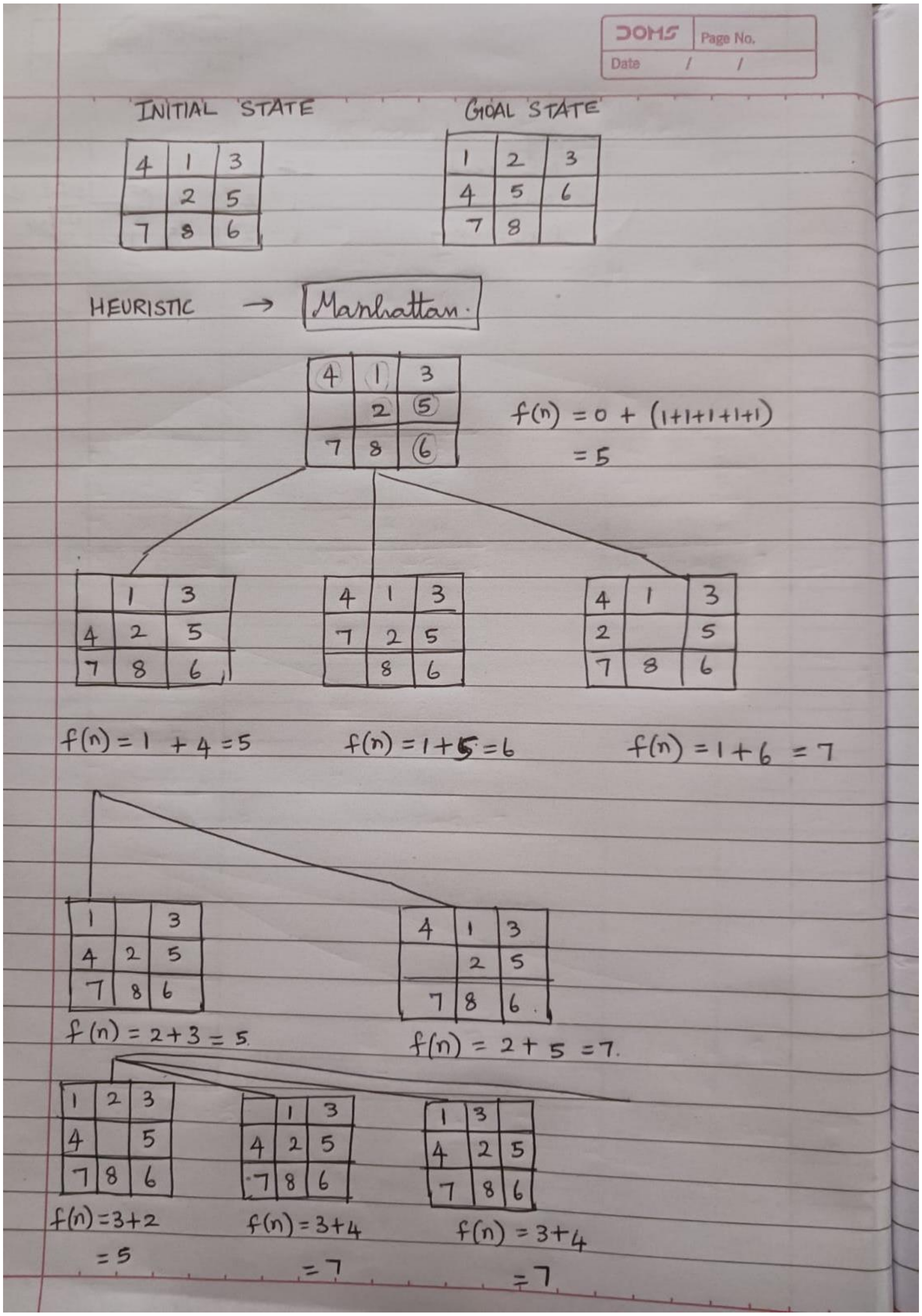
5 nodes expanded

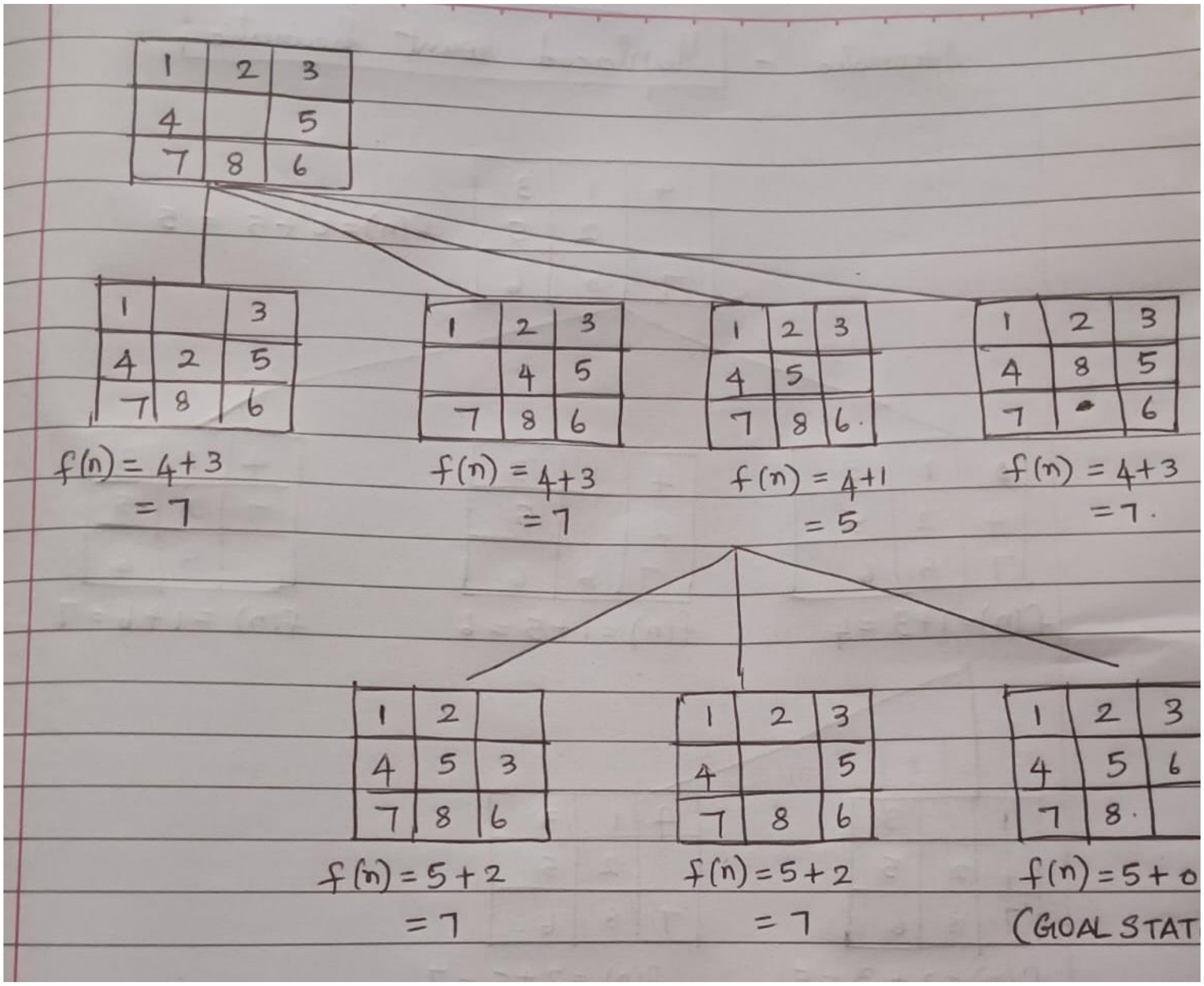
Solution at depth 4</td></tr>
</table>

<table>
<tr>
<td>Example 6
<table><tr><td>4</td><td>7</td><td>5</td></tr><tr><td>8</td><td>1</td><td>3</td></tr><tr><td>2</td><td></td><td>6</td></tr></table></td>
<td><table><tr><td>4</td><td>7</td><td>5</td></tr><tr><td>8</td><td>3</td><td>6</td></tr><tr><td>2</td><td>1</td><td></td></tr></table></td>
<td>9 nodes generated
4 nodes expanded
Solution at depth 3</td>
<td>9 nodes generated
4 nodes expanded
Solution at depth 3</td>
</tr>
</table>

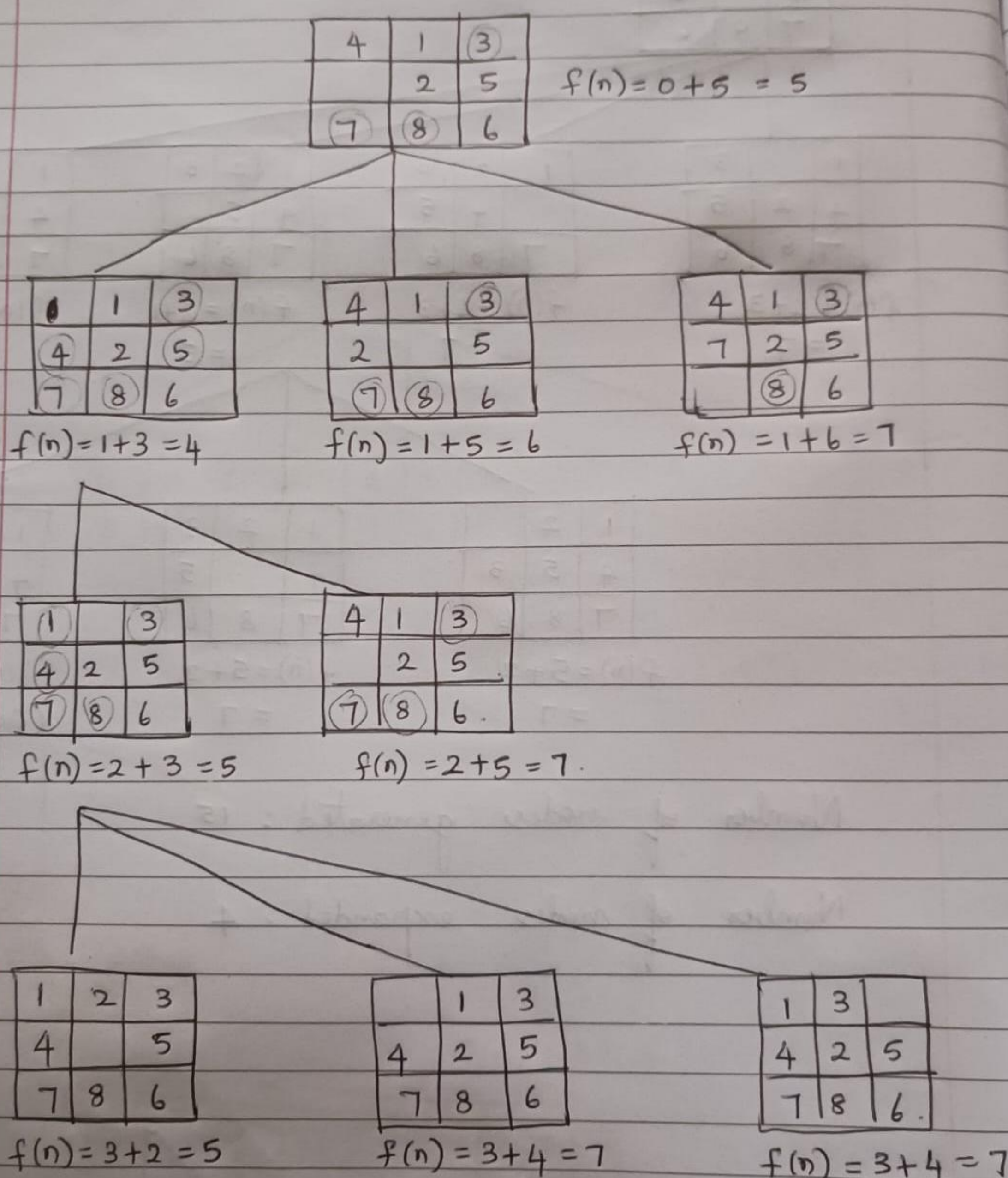
6. EXAMPLES

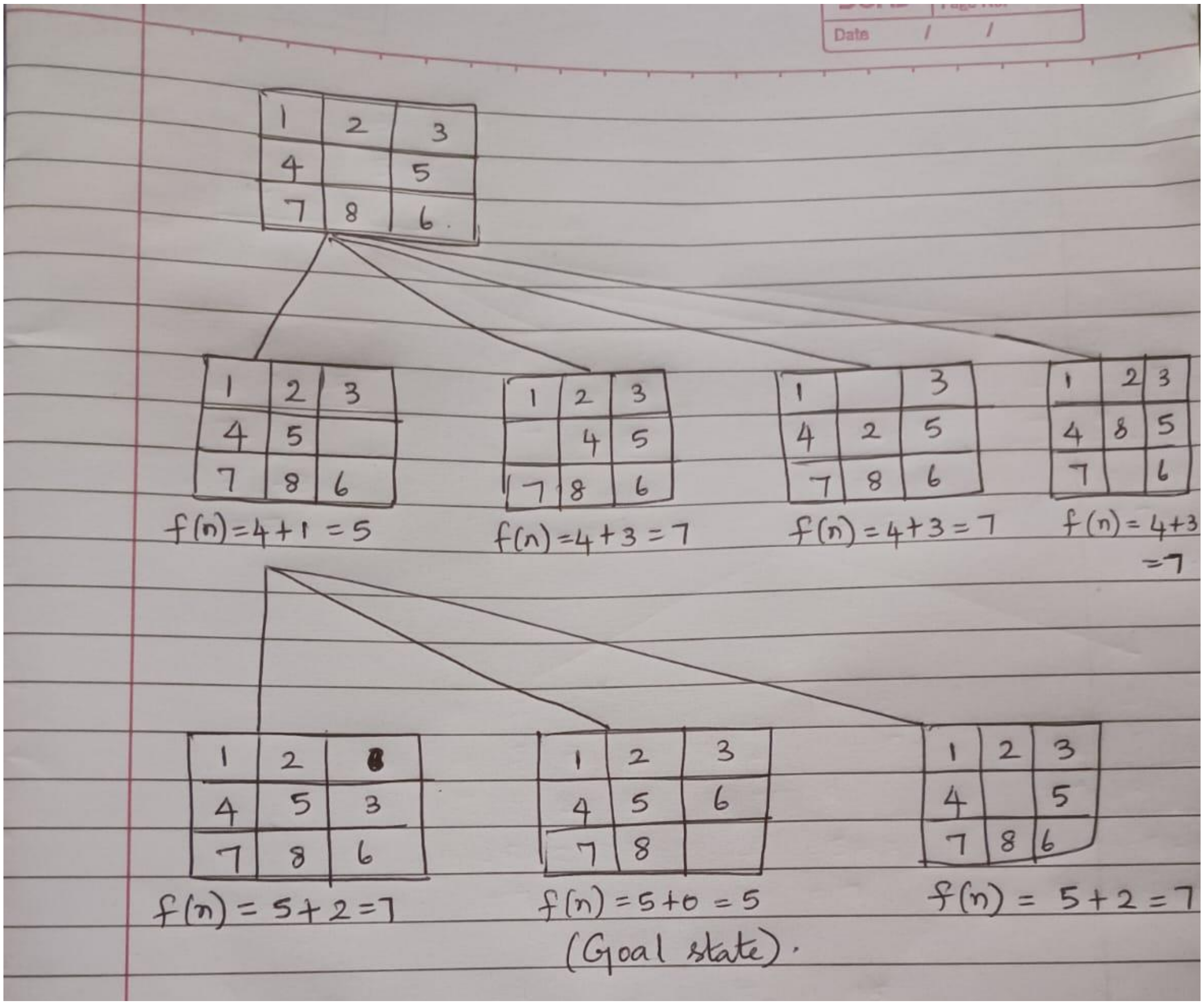
Example 1 :



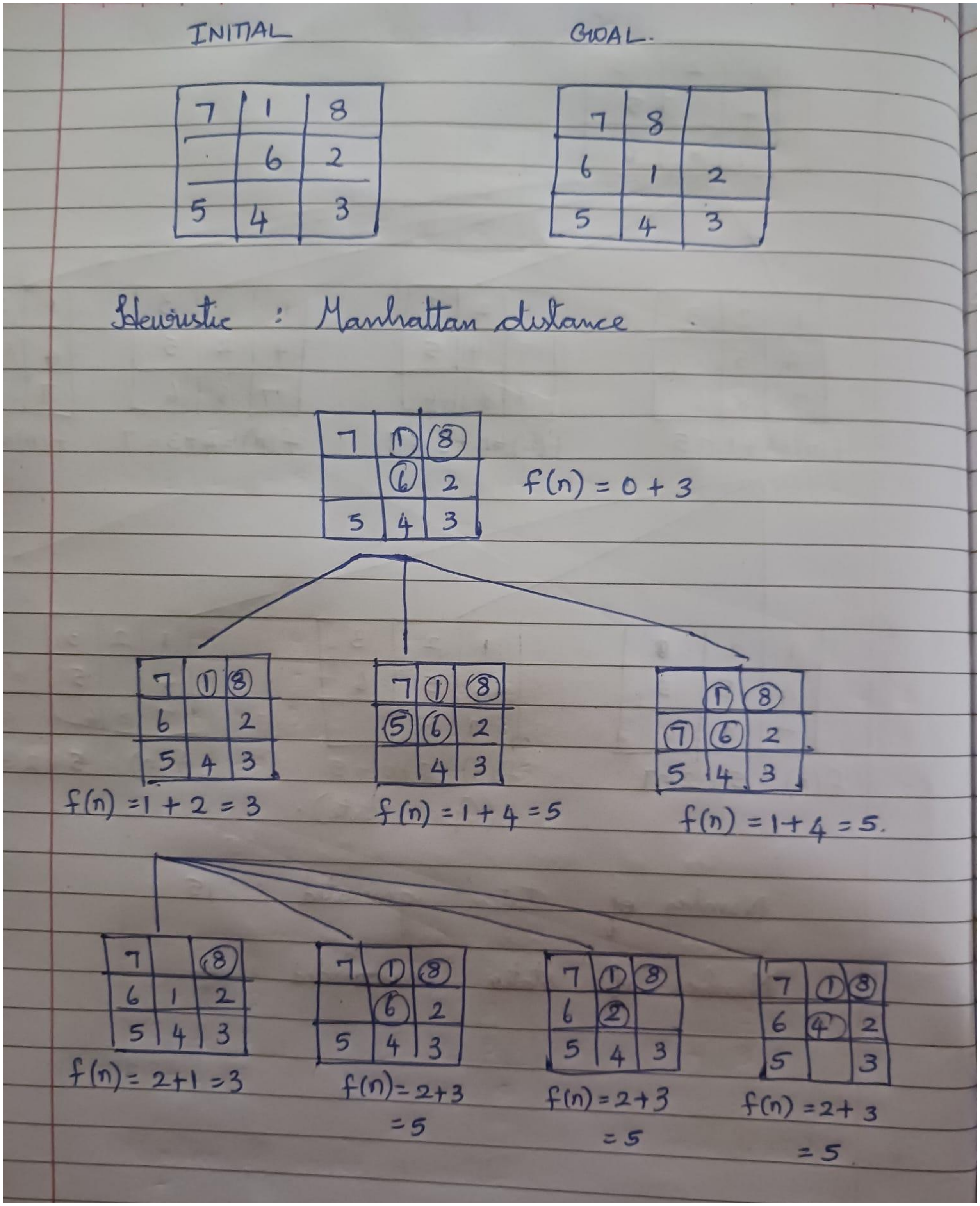

f(n) = 4+3
= 7
f(n) = 4+3
= 7
f(n) = 4+1
= 5
f(n) = 4+3
= 7.
f(n) = 5+2
= 7
f(n) = 5+2
= 7
f(n) = 5+0
(GOAL STAT

Heuristic - Misplaced count heuristic




Date / /
1 2 3
4 _ 5
7 8 6
1 2 3
4 5 _
7 8 6
f(n)=4+1 =5
1 2 3
_ 4 5
7 8 6
f(n) =4+3 = 7
1 _ 3
4 2 5
7 8 6
f(n) = 4+3 = 7
1 2 3
4 8 5
7 _ 6
f(n) = 4+3
=7
1 2 _
4 5 3
7 8 6
f(n) = 5+2=7
1 2 3
4 5 6
7 8 _
f(n) = 5+0 = 5
(Goal state).
1 2 3
4 _ 5
7 8 6
f(n) = 5+2 = 7

Example 2 :


INITIAL
GOAL.
Heuristic : Manhattan distance
f(n) = 0 + 3
f(n) = 1 + 2 = 3
f(n) = 1 + 4 = 5
f(n) = 1+4 = 5.
f(n) = 2+1 = 3
f(n) = 2+3
= 5
f(n) = 2+3
= 5
f(n) = 2+ 3
= 5

7		8
6	1	2
5	4	3

7	8	
6	1	2
5	4	3

$f(n) = 3 + 0 = 3$

(Goal state)

	7	8
6	1	2
5	4	3

$f(n) = 3 + 2 = 5$

7	1	8
6		2
5	4	3

$f(n) = 3 + 2 = 5.$

Heuristic : Misplaced cost.

7	1	8
	6	2
5	4	3

$f(n) = 0 + 3 = 3$

	1	8
7	6	2
5	4	3

$f(n) = 1 + 4 = 5$

7	1	8
6		2
5	4	3

$f(n) = 1 + 2 = 3$

7	1	8
5	6	2
	4	3

$f(n) = 1 + 4 = 5$.

7		8
6	1	2
5	4	3

$f(n) = 2 + 1 = 3$

7	1	8
	6	2
5	4	3

$f(n) = 2 + 3 = 5$

7	1	8
6	2	
5	4	3

$f(n) = 2 + 3 = 5$

7	1	8
6	4	2
5		3

$f(n) = 2 + 3 = 5$

7	1	8
6		2
5	4	3

$f(n) = 3 + 2 = 5$

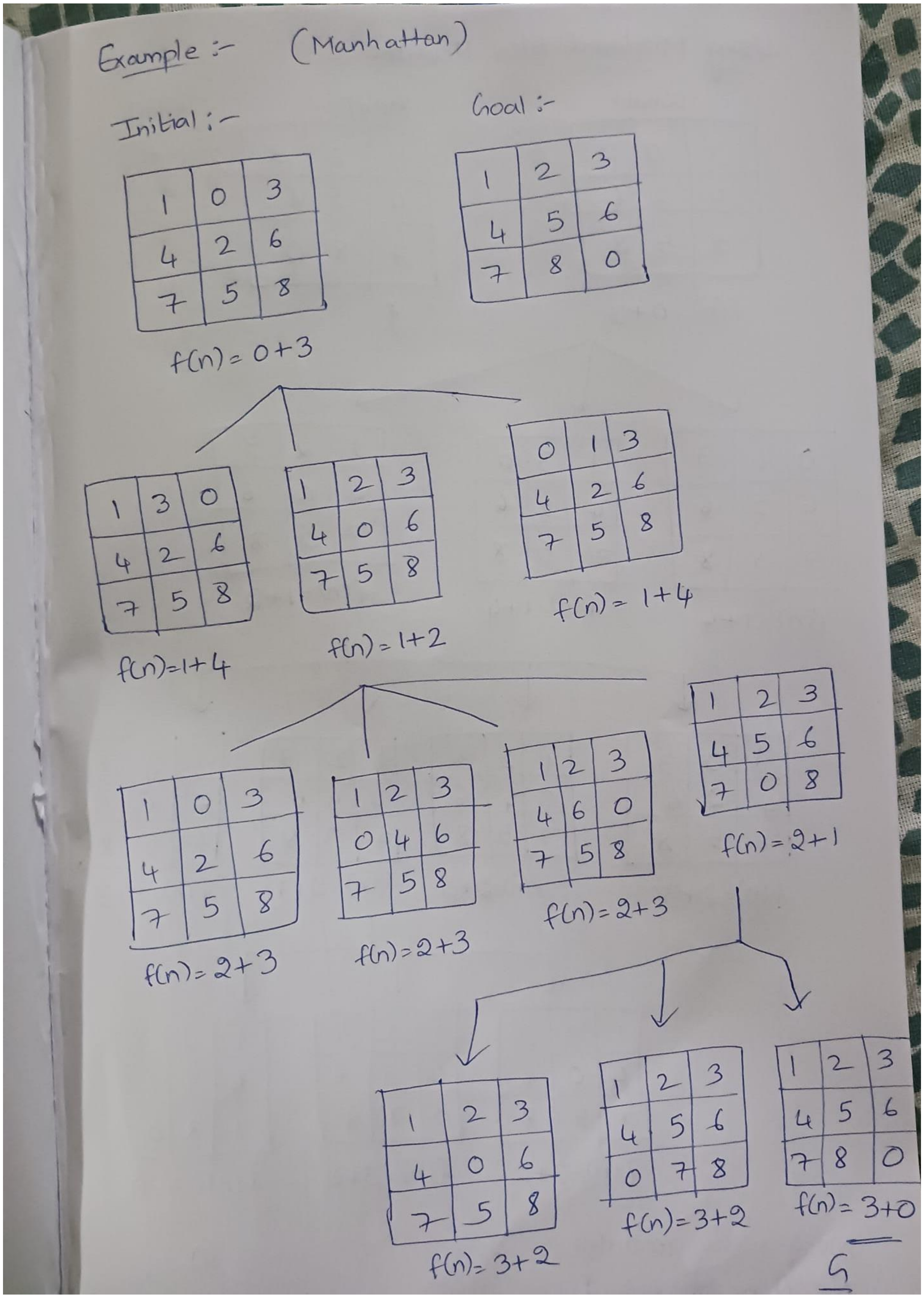
	7	8
6	1	2
5	4	3

$f(n) = 3 + 2 = 5$

7	8	
6	1	2
5	4	3

$f(n) = 3 + 0 = 3$
(Goal state).

Example 3


Example :- (Manhattan)
Initial :-
Goal :-
f(n) = 0+3
f(n)=1+4
f(n) = 1+2
f(n) = 1+4
f(n) = 2+3
f(n) = 2+3
f(n) = 2+3
f(n) = 2+1
f(n) = 3+2
f(n) = 3+2
f(n) = 3+0
G

Using Misplaced tiles Heuristic :-

Initial:-

1	0	3
4	2	6
7	5	8

$f(n) = 0+3$

Goal :-

1	2	3
4	5	6
7	8	0

0	1	3
4	2	6
7	5	8

$f(n) = 1+4$

1	3	0
4	2	6
7	5	8

$f(n) = 1+4$

1	2	3
4	0	6
7	5	8

$f(n) = 1+2$

1	0	3
4	2	6
7	5	8

$f(n) = 2+3$

1	2	3
0	4	6
7	5	8

$f(n) = 2+3$

1	2	3
4	5	6
7	0	8

$f(n) = 2+1$

1	2	3
4	6	0
7	5	8

$f(n) = 2+3$

1	2	3
4	0	6
7	5	8

$f(n) = 3+2$

1	2	3
4	5	6
0	7	8

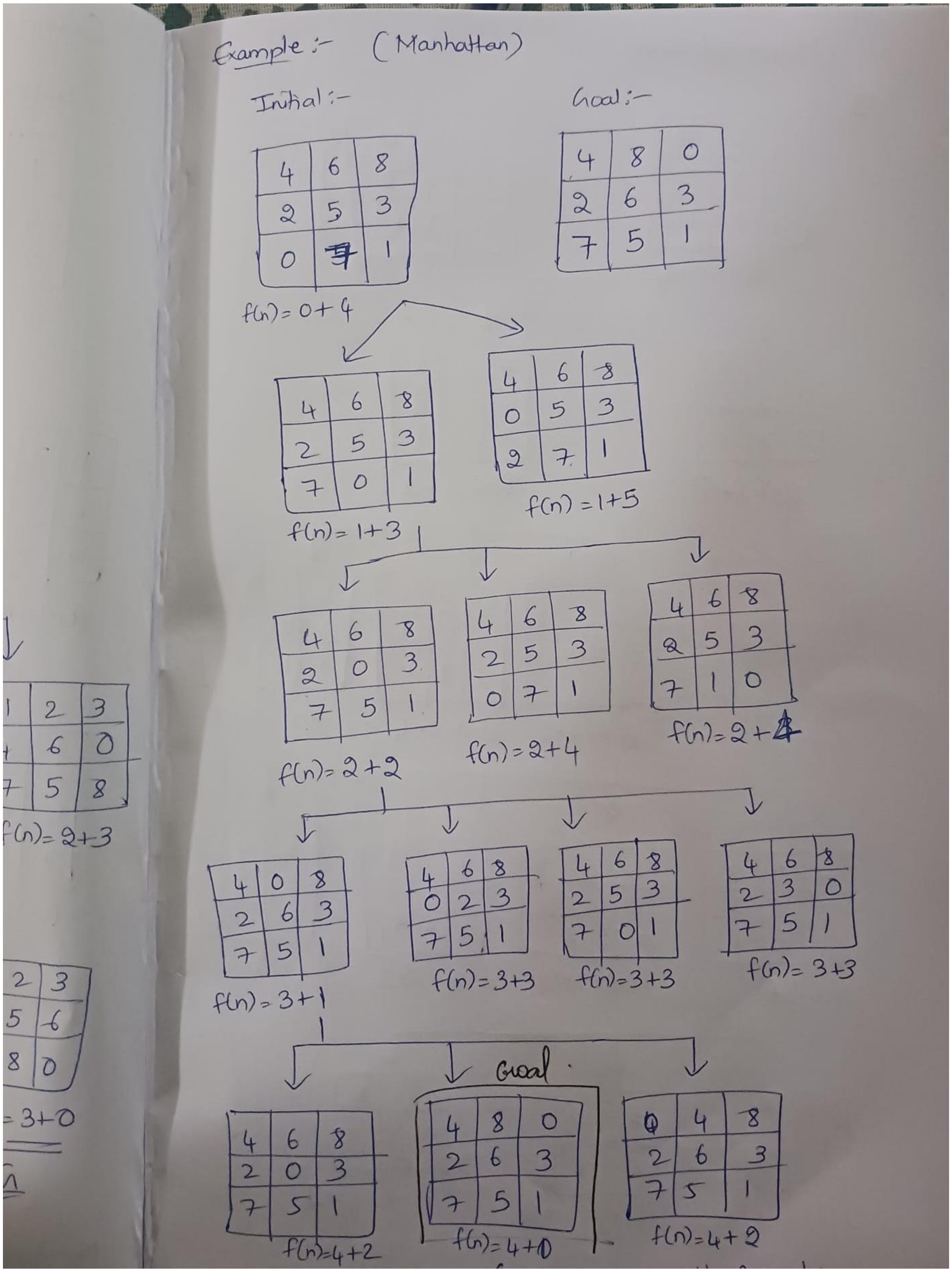
$f(n) = 3+2$

1	2	3
4	5	6
7	8	0

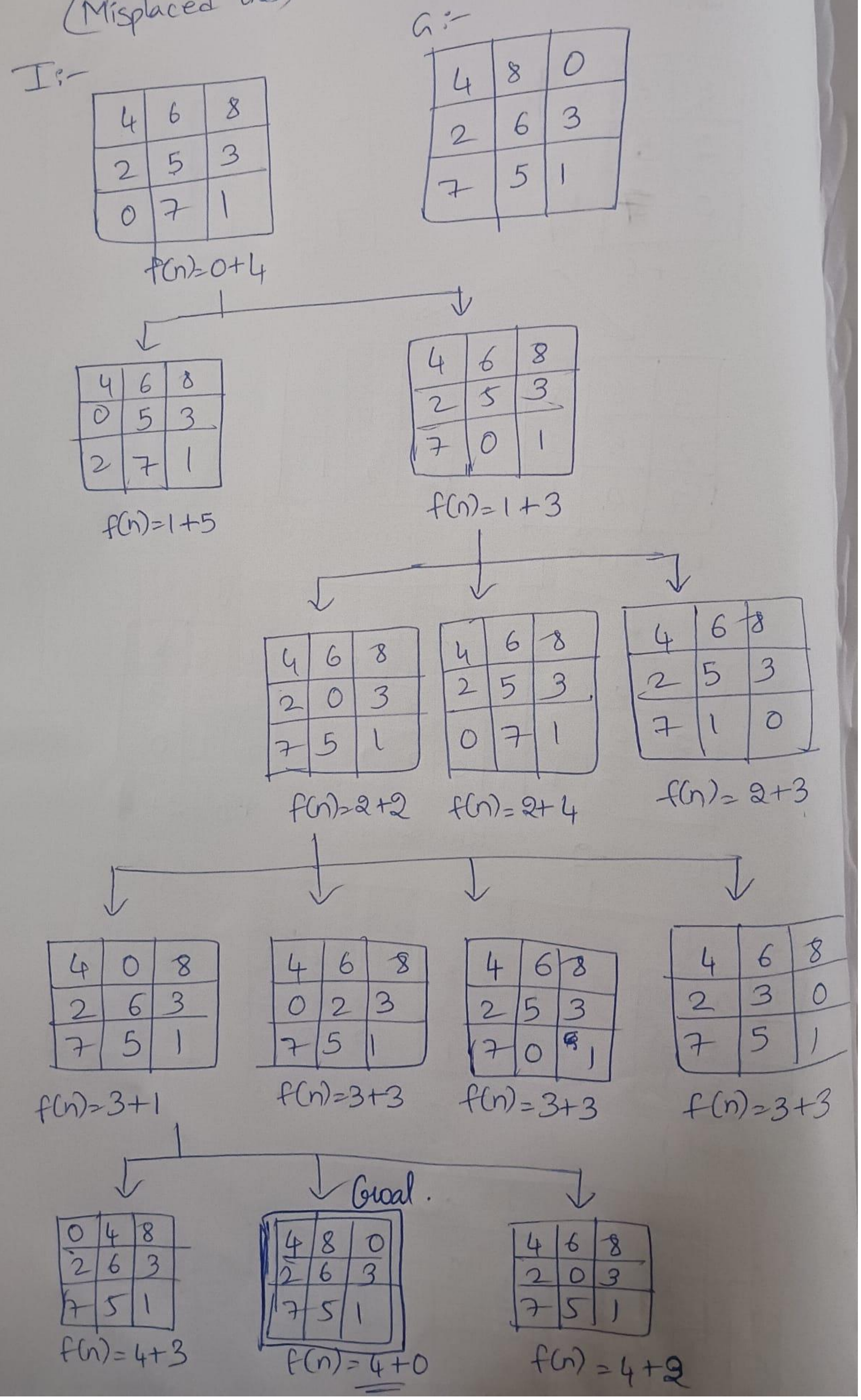
$f(n) = 3+0$

G

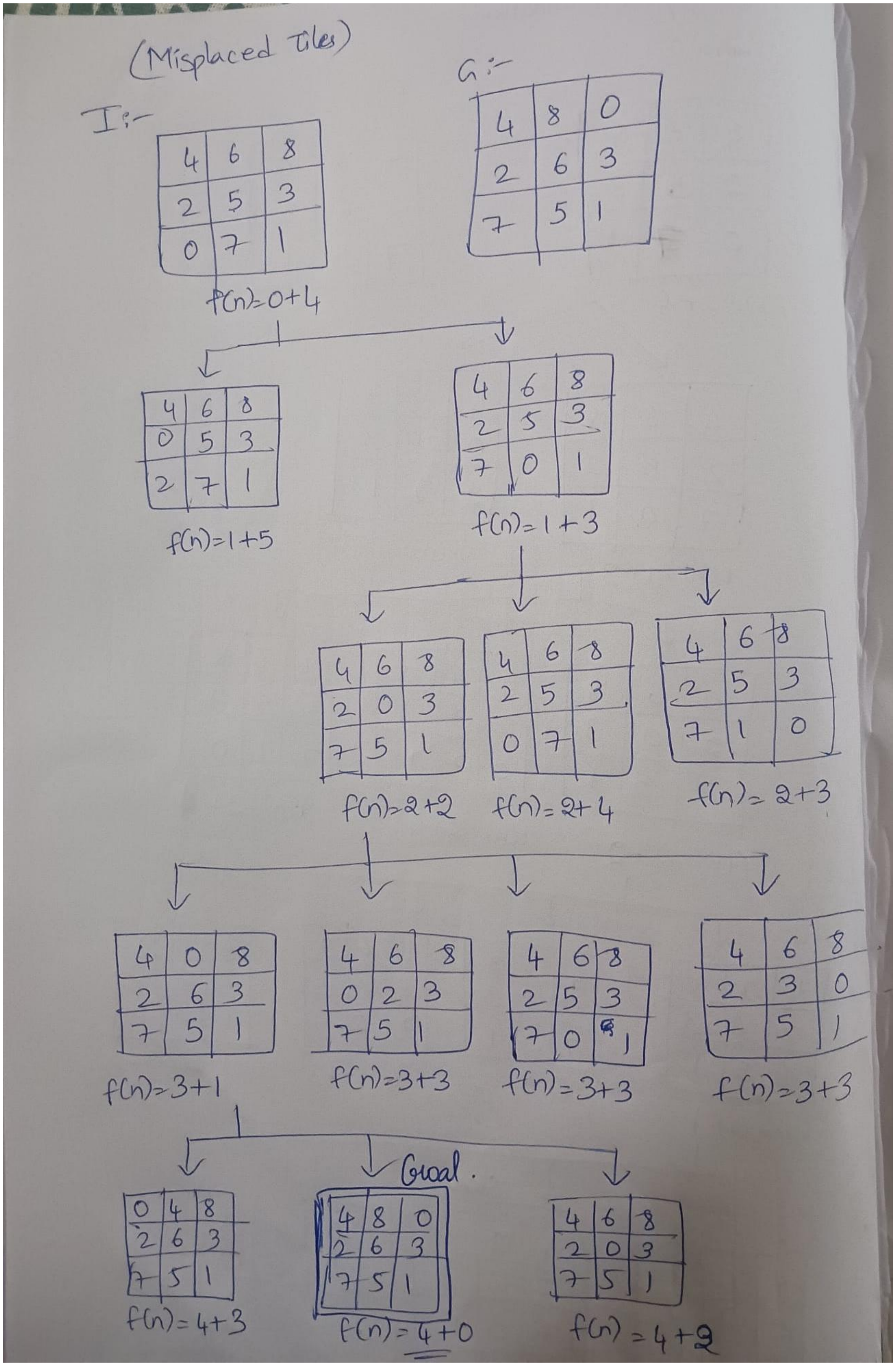
Example 4


Example :- (Manhattan)
Initial :-
Goal :-
f(n) = 0+4
f(n) = 1+3
f(n) = 1+5
f(n) = 2+2
f(n) = 2+4
f(n) = 2+4
f(n) = 2+3
f(n) = 3+1
f(n) = 3+3
f(n) = 3+3
f(n) = 3+3
= 3+0
Goal
f(n)=4+2
f(n)=4+0
f(n)=4+2

(Misplaced Tiles)



Example 5



(Misplaced Tiles)
I:-
G:-
f(n)=0+4
f(n)=1+5
f(n)=1+3
f(n)=2+2
f(n)=2+4
f(n)=2+3
f(n)=3+1
f(n)=3+3
f(n)=3+3
f(n)=3+3
Goal.
f(n)=4+3
f(n)=4+0
f(n)=4+2

Example :- (Manhattan)

Initial :-

4	6	8
2	5	3
0	7	1

$f(n) = 0 + 4$

Goal :-

4	8	0
2	6	3
7	5	1

4	6	8
2	5	3
7	0	1

$f(n) = 1 + 3$

4	6	8
0	5	3
2	7	1

$f(n) = 1 + 5$

4	6	8
2	0	3
7	5	1

$f(n) = 2 + 2$

4	6	8
2	5	3
0	7	1

$f(n) = 2 + 4$

4	6	8
2	5	3
7	1	0

$f(n) = 2 + 4$

4	0	8
2	6	3
7	5	1

$f(n) = 3 + 1$

4	6	8
0	2	3
7	5	1

$f(n) = 3 + 3$

4	6	8
2	5	3
7	0	1

$f(n) = 3 + 3$

4	6	8
2	3	0
7	5	1

$f(n) = 3 + 3$

4	6	8
2	0	3
7	5	1

$f(n) = 4 + 2$

Goal

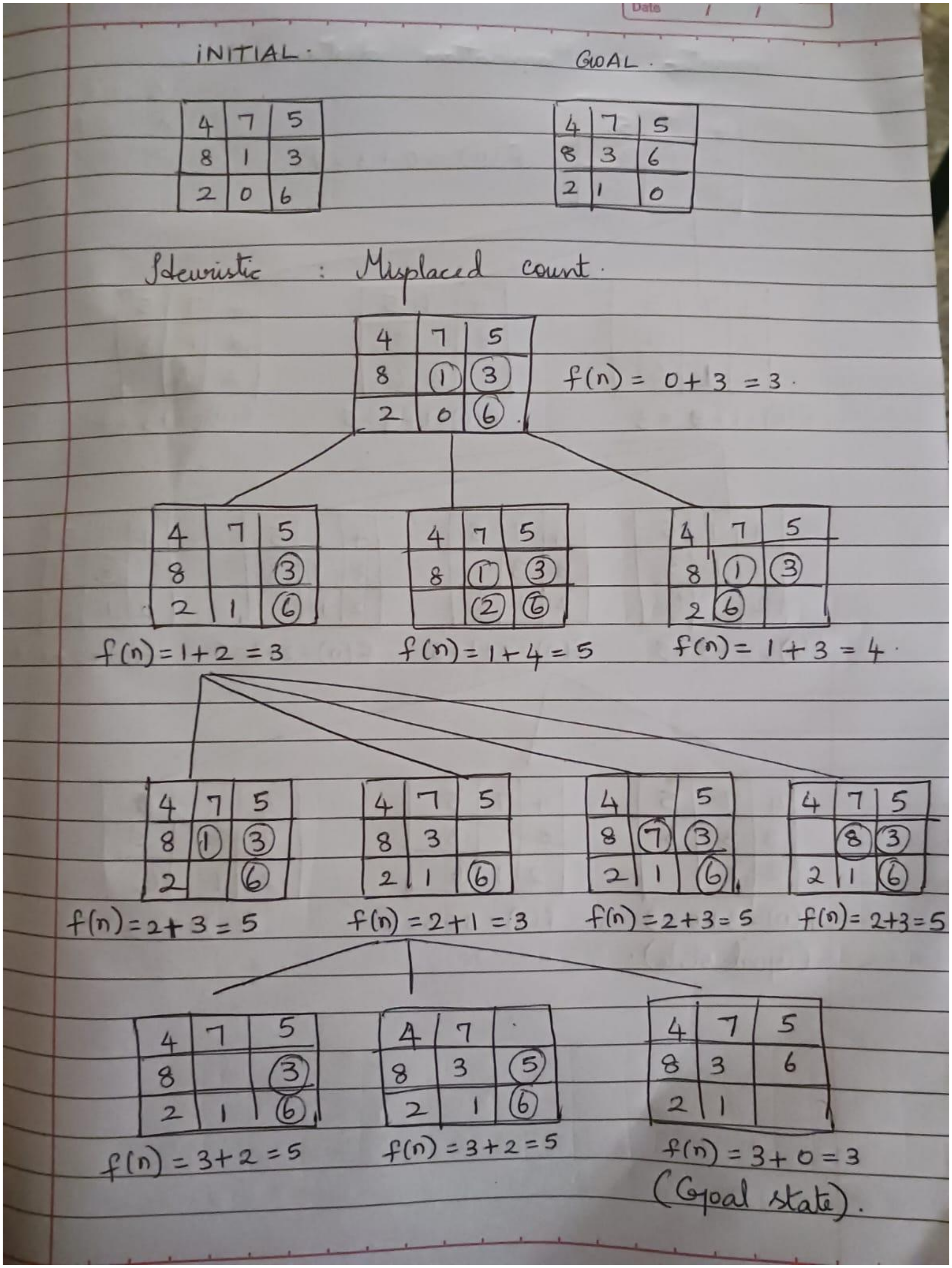
4	8	0
2	6	3
7	5	1

$f(n) = 4 + 0$

0	4	8
2	6	3
7	5	1

$f(n) = 4 + 2$

Example 6


INITIAL.
GOAL.
Heuristic : Misplaced count.
f(n) = 0 + 3 = 3.
f(n) = 1 + 2 = 3
f(n) = 1 + 4 = 5
f(n) = 1 + 3 = 4.
f(n) = 2 + 3 = 5
f(n) = 2 + 1 = 3
f(n) = 2 + 3 = 5
f(n) = 2+3 = 5
f(n) = 3 + 2 = 5
f(n) = 3 + 2 = 5
f(n) = 3 + 0 = 3
(Goal state).

Heuristic : Manhattan distance

Goal

4	7	5
8	3	6
2	1	

4	7	5
8	1	3
2		6

$f(n) = 0 + 3 = 3.$

4	7	5
8		3
2	1	6

$f(n) = 1 + 2 = 3$

4	7	5
8	1	3
	2	6

$f(n) = 1 + 4 = 5$

4	7	5
8	1	3
2	6	

$f(n) = 1 + 3 = 4.$

4	7	5
8	1	3
2		6

$f(n) = 2 + 3 = 5$

4	7	5
8	3	
2	1	6

$f(n) = 2 + 1 = 3$

4		5
8	7	3
2	1	6

$f(n) = 2 + 3 = 5$

4	7	5
	8	3
2	1	6

$f(n) = 2 + 3 = 5$

4	7	5
8	3	6
2	1	

$f(n) = 3 + 0 = 3$
(Goal state).

4	7	5
8		3
2	1	6

$f(n) = 3 + 3 = 6$

4	7	
8	3	5
2	1	6

$f(n) = 3 + 2 = 5.$

7 . Conclusion :

The A * search algorithm has been implemented and applied to solve 8 puzzle problem , using two different heuristic functions - Manhattan distance and Misplaced tiles count.