Program 1

```
def main():
   a=False
   b=True
   print("not operation of a= ",not(a))
   print("or operation of a and b= ",(a or b))
   print("and operation of a and b= ", (a and b))
   print("xor operation of a and b= ", (a ^ b))
   print("xnor operation of a and b= ", not(a ^ b))
   print("implication of a and b= ", imp(a,b))
   print("Bidirectional operation of a and b= ",bidir(a,b))
def imp(a,b):
  return (not(a)) or b
def bidir(a,b):
   return (imp(a,b) and imp(b,a))
if __name__ == '__main__':
   main()
```

Program 2

```
flag=True
count=1
while flag:
    perc=input("enter the percept")
    loc=input("enter the location")
    if loc=="A":
        if perc=="dirty":
            print("action: suck...turn right")
        else:
            print("action: turn right")
    else:
        if perc=="dirty":
            print("action: suck.....turn left")
        else:
            print("action: turn left")
    print("Do you want to continue?")
    Cont=input("Enter Y or N")
    if Cont == 'Y':
        flag= True
    else :
        flag = False
```

Program 3

```
# Given M x N grid(floor) create an agent that moves around the grid until the entire grid is clean
floor = [[1, 0, 0, 0], # '1' represents dirty and '0' represents clean
        [0, 1, 0, 1],
        [1, 0, 1, 1]]
def clean(floor):
    m = len(floor[0]) # no of cols
    n = len(floor)    # no of rows
    no_of_tiles = m * n
    tiles_checked = 0
    row = 0
    col = 0
    while tiles_checked < no_of_tiles:
  # Current position
     print_floor(floor, row, col)
  # Suck if dirty
      if floor[row][col] == 1:
            floor[row][col] = 0
            print('Sucked the dirt')
        else:
            print('Already Clean')
      # Next tile
        if row % 2 == 0:          # Even rows the bot moves right to the next tile
            if col < m-1:
                col += 1
            else:
                row += 1  # Move to next row if we reached the last col
        elif row % 2 == 1:        # Odd rows the bot moves left to the next tile
            if 0 < col:
                col -= 1
            else:
```

```
 row += 1  # Move to next row if we reached the last col
 tiles_checked += 1
 print('---------------')
 print('Cleaned!!!')
def print_floor(floor, row, col):
temp = floor[row][col]
floor[row][col] = 'VC'
for x in floor:
print(x)
floor[row][col] = temp
```
```
# Call the function
clean(floor)
```

Program 4

```
N = 4
def main():
 s = [1,0,1,0]
 t = [1,1,0,0]
    a=[]
    b=[]
    c=[]
    d=[]
    e=[]
 for i in range(N):
    a.append(not(s[i] or t[i]))
    b.append(bool(s[i] and t[i]))
    c.append(bool(t[i] or(not(t[i]))))
    d.append(not(bidir(s[i],s[i])))
    e.append(imp((not(s[i])),(not(t[i]))))
    print("Truth table of a: ",a)
    print("Truth table of b: ", b)
    print("Truth table of c: ", c)
    print("Truth table of d: ", d)
    print("Truth table of e: ", e)
    p=entails(a, b)
    q=entails(a,c)
    r=entails(a, d)
    s=entails(a, e)
    print("a entails b: ",p)
    print("a entails c: ", q)
    print("a entails d: ", r)
    print("a entails e: ", s)
   def imp(j,k):
   return (not(j)) or k
```

```
def bidir(j,k):
 return (imp(j,k) and imp(j,k))
def entails(m,n):
  #for i in j:
  for i in range(N):
  for j in range(N):
            if (m[i] and n[j]== 1):
               if(i==j):
                  return "yes"
                  break
  return "NO"
if __name__ == '__main__':
   main()
```

Program 5

```
from copy import deepcopy

import numpy as np

import time

# takes the input of current states and evaluvates the best path to goal state

def bestsolution(state):

   bestsol = np.array([], int).reshape(-1, 9)

   count = len(state) - 1

   while count != -1:

   bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)

   count = (state[count]['parent'])

   return bestsol.reshape(-1, 3, 3)

# this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.

def all(checkarray):

   set=[]

   for it in set:

   for checkarray in it:

       return 1

    else:

       return 0

# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state

def manhattan(puzzle, goal):

   a = abs(puzzle // 3 - goal // 3)

   b = abs(puzzle % 3 - goal % 3)

   mhcost = a + b

   return sum(mhcost[1:])

# will calcuates the number of misplaced tiles in the current state as compared to the goal state

def misplaced_tiles(puzzle,goal):

   mscost = np.sum(puzzle != goal) - 1

   return mscost if mscost > 0 else 0
```

```
#3[on_true] if [expression] else [on_false]

# will indentify the coordinates of each of goal or initial state values

def coordinates(puzzle):

    pos = np.array(range(9))

    for p, q in enumerate(puzzle):

    pos[q] = p

    return pos

# start of 8 puzzle evaluvation, using Manhattan heuristics

def evaluvate(puzzle, goal):

    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],

    dtype =  [('move',  str, 1),('position', list),('head', int)])

   dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

   # initializing the parent, gn and hn, where hn is manhattan distance function call

    costg = coordinates(goal)

    parent = -1

    gn = 0

    hn = manhattan(coordinates(puzzle), costg)

    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.

    dtpriority = [('position', int),('fn', int)]

    priority = np.array( [(0, hn)], dtpriority)

     while 1:

        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

        position, fn = priority[0]

        priority = np.delete(priority, 0, 0)

        # sort priority queue using merge sort,the first element is picked for exploring remove from
queue what we are exploring

        puzzle, parent, gn, hn = state[position]

        puzzle = np.array(puzzle)

        # Identify the blank square in input

        blank = int(np.where(puzzle == 0)[0])
```

```
gn = gn + 1
c = 1
start_time = time.time()
for s in steps:
c = c + 1
if blank not in s['position']:
# generate new state as copy of current
 openstates = deepcopy(puzzle)
openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]
# The all function is called, if the node has been previously explored or not
if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
end_time = time.time()
if (( end_time - start_time ) > 2):
print(" The 8 puzzle is unsolvable ! \n")
exit
# calls the manhattan function to calcuate the cost
hn = manhattan(coordinates(openstates), costg)
# generate and add new state in the list
q = np.array([(openstates, position, gn, hn)], dtstate)
state = np.append(state, q, 0)
# f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state
fn = gn + hn
q = np.array([(len(state) - 1, fn)], dtpriority)
 priority = np.append(priority, q, 0)
 # Checking if the node in openstates are matching the goal state.
 if np.array_equal(openstates, goal):
  print(' The 8 puzzle is solvable ! \n')
 return state, len(priority)
 return state, len(priority)
```

Program 6

```
def pour(jug1, jug2):
 max1, max2, fill = 3, 4, 2  #Change maximum capacity and final capacity
 print("%d\t%d" % (jug1, jug2))
  if jug2 is fill:
      return
  elif jug2 is max2:
      pour(0, jug1)
   elif jug1 != 0 and jug2 is 0:
      pour(0, jug1)
   elif jug1 is fill:
      pour(jug1, 0)
   elif jug1 < max1:
      pour(max1, jug2)
   elif jug1 < (max2-jug2):
      pour(0, (jug1+jug2))
    else:
      pour(jug1-(max2-jug2), (max2-jug2)+jug2)
 print("JUG1\tJUG2")
pour(0, 0)
```