Artificial Intelligence Laboratory

Gourav Kumar Shaw (2020CSB010)

Assignment-3

---

1. **Develop a program that efficiently determines the next higher permutation of a positive integer represented as a list of decimal digits. The next higher permutation is defined as the smallest integer greater than the given integer, while maintaining the same set of digits. Ensure that the program returns the result as a list of decimal digits. This task requires a careful consideration of algorithmic efficiency and precision in handling digit permutations, elevating the difficulty level of the problem. For example, the next higher permutation of 123542 is 124235.**

The code for finding the next permutation is given below:

```
def next_higher_permutation(nums):
  i = len(nums)-2
  while i>=0 and nums[i]>=nums[i+1]:
    i-=1
  if (i == -1):
    return None
  j = len(nums)-1
  while(nums[j] <= nums[i]):
    j-=1
  nums[i], nums[j] = nums[j], nums[i]
  nums[i+1:] = reversed(nums[i+1:])
  return nums
```

Testing the code-

```
print(next_higher_permutation([7,6,3]))
```

Output -

```
None
```

Testing the code with another number-

```
print(next_higher_permutation([1,8,1,5,9,6,4,3]))
```

Output -

```
[1, 8, 1, 6, 3, 4, 5, 9]
```

2. **Develop a program that systematically generates the sequence of moves required to transfer a set of N discs from an initial arrangement on one pole to a target arrangement on another pole. The setup involves multiple poles and discs, each with varying diameters. The primary goal is to move all discs from the initial pole to the target pole while respecting two fundamental constraints: only the topmost disc on a pole can be moved at any given time, and no disc may be placed on a smaller disc. This programming task entails devising an algorithm capable of efficiently producing the sequence of moves, demonstrating heightened complexity due to the nuanced management of disc placements and strict adherence to the specified rules.**

The program to solve the Tower of Hanoi game is given below.

```
def towerOfHanoi(n,source_rod,dest_rod,using_rod):
    if n == 0:
        return
    towerOfHanoi(n-1,source_rod,using_rod,dest_rod)
    print("Moving Disk", n, "from tower", source_rod, "to tower", dest_rod)
    towerOfHanoi(n-1,using_rod,dest_rod,source_rod)
```

Testing the code-

```
towerOfHanoi(6,'A','C','B')
```

Output -

```
Moving Disk 1 from tower A to tower B
Moving Disk 2 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 3 from tower A to tower B
Moving Disk 1 from tower C to tower A
Moving Disk 2 from tower C to tower B
Moving Disk 1 from tower A to tower B
Moving Disk 4 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 2 from tower B to tower A
Moving Disk 1 from tower C to tower A
Moving Disk 3 from tower B to tower C
Moving Disk 1 from tower A to tower B
Moving Disk 2 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 5 from tower A to tower B
Moving Disk 1 from tower C to tower A
Moving Disk 2 from tower C to tower B
Moving Disk 1 from tower A to tower B
Moving Disk 3 from tower C to tower A
Moving Disk 1 from tower B to tower C
Moving Disk 2 from tower B to tower A
Moving Disk 1 from tower C to tower A
Moving Disk 4 from tower C to tower B
Moving Disk 1 from tower A to tower B
Moving Disk 2 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 3 from tower A to tower B
Moving Disk 1 from tower C to tower A
Moving Disk 2 from tower C to tower B
Moving Disk 1 from tower A to tower B
Moving Disk 6 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 2 from tower B to tower A
Moving Disk 1 from tower C to tower A
Moving Disk 3 from tower B to tower C
Moving Disk 1 from tower A to tower B
Moving Disk 2 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 4 from tower B to tower A
Moving Disk 1 from tower C to tower A
Moving Disk 2 from tower C to tower B
Moving Disk 1 from tower A to tower B
Moving Disk 3 from tower C to tower A
Moving Disk 1 from tower B to tower C
Moving Disk 2 from tower B to tower A
Moving Disk 1 from tower C to tower A
Moving Disk 5 from tower B to tower C
Moving Disk 1 from tower A to tower B
Moving Disk 2 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 3 from tower A to tower B
Moving Disk 1 from tower C to tower A
Moving Disk 2 from tower C to tower B
Moving Disk 1 from tower A to tower B
```

```
Moving Disk 4 from tower A to tower C
Moving Disk 1 from tower B to tower C
Moving Disk 2 from tower B to tower A
Moving Disk 1 from tower C to tower A
Moving Disk 3 from tower B to tower C
Moving Disk 1 from tower A to tower B
Moving Disk 2 from tower A to tower C
Moving Disk 1 from tower B to tower C
```

Testing the code with another number-

```
towerOfHanoi(3,'P','R','Q')
```

Output -

```
Moving Disk 1 from tower P to tower R
Moving Disk 2 from tower P to tower Q
Moving Disk 1 from tower R to tower Q
Moving Disk 3 from tower P to tower R
Moving Disk 1 from tower Q to tower P
Moving Disk 2 from tower Q to tower R
Moving Disk 1 from tower P to tower R
```

3. **Create a program designed to address a classic optimization challenge involving the determination of the most efficient route among a set of cities. In this problem, the objective is to find the shortest possible path that visits each city exactly once, returning to the starting city. The program should adeptly explore and assess potential routes, providing an optimal solution. Accomplishing this task necessitates the incorporation of an algorithm, whether heuristic or exact, capable of efficiently navigating the intricacies inherent in identifying the shortest path within a specified collection of cities. Use the hill climbing algorithm to solve the problem.**

The program to find the most efficient route is given below.

```
import itertools

def distance(city1,city2):
    # Function to calculate the Euclidean distance between two cities
    return ((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)**0.5

def total_distance(route,cities):
    # Function to calculate the total distance of a route
    total_dist = 0
    for i in range(len(route)-1):
        total_dist += distance(cities[route[i]],cities[route[i+1]])
    total_dist += distance(cities[route[-1]],cities[route[0]])  # Return to the starting city
    return total_dist

def hill_climbing_tsp(cities):
    # Hill Climbing algorithm for solving TSP
    num_cities = len(cities)
    best_route = list(range(num_cities))
    best_distance = total_distance(best_route,cities)

    while True:
        found_better = False
        for i in range(1, num_cities-1):
            for j in range(i+1, num_cities):
                new_route = best_route[:i] + best_route[i:j][::-1] + best_route[j:]
                new_distance = total_distance(new_route,cities)
```

```
                if new_distance < best_distance:
                    best_route = new_route
                    best_distance = new_distance
                    found_better = True

        if not found_better:
            break

    return best_route, best_distance
```

Testing the code-

```
cities = [(0, 0), (7, 8), (6, 3), (4, 5), (3, 4), (7, 12), (10, 5), (2, 2)]
optimal_route, optimal_distance = hill_climbing_tsp(cities)

print("Optimal Route:", optimal_route)
print("Optimal Distance:", optimal_distance)
```

Output -

```
Optimal Route: [0, 2, 6, 1, 5, 3, 4, 7]
Optimal Distance: 33.517462345101215
```

4. **In the realm of Artificial Intelligence, contemplate a problem involving two containers of indeterminate capacity, referred to as jugs. One jug has a capacity of 3 units, while the other holds up to 4 units. There is no markings or additional measuring instruments, the objective is to develop a strategic approach to precisely fill the 4-unit jug with 2 units of water. The restriction stipulates the use of solely the aforementioned jugs, excluding any supplementary tools. Both jugs initiate the scenario in an empty state. The aim is to attain the desired water quantity in the 4-unit jug by executing a sequence of permissible operations, including filling, emptying, and pouring water between the jugs. The challenge in this scenario involves crafting an algorithm, such as Depth First Search, to systematically explore and determine the optimal sequence of moves for accomplishing the task while adhering to the defined constraints.**

The program to determine sequence of moves to fill the jugs as wanted, is given below.

```
# Defining all valid moves
def emptyjug1(state):
  return (0,state[1],"Empty jug1")
def emptyjug2(state):
  return (state[0],0,"Empty jug2")
def filljug1(state,capacity):
  return (capacity[0],state[1],"Fill jug1")
def filljug2(state,capacity):
  return (state[0],capacity[1],"Fill jug2")
def pour1to2(state,capacity):
  if state[0]+state[1] <= capacity[1]:
    return (0,state[1]+state[0],"Pour from jug1 to jug2")
  else:
    return (state[0]+state[1]-capacity[1],capacity[1],"Pour from jug1 to jug2")
def pour2to1(state,capacity):
  if state[0]+state[1] <= capacity[0]:
    return (state[1]+state[0],0,"Pour from jug2 to jug1")
  else:
    return (capacity[0],state[0]+state[1]-capacity[0],"Pour from jug2 to jug1")

def depthFirstSearch(currstate,capacity,target,visited,sequence):
  if(currstate==target):
    return True
  moves = []
```

```
  visited.add(currstate)
  x,y = currstate[0], currstate[1]
  if(x>0):
    moves.append(emptyjug1(currstate))
  if(y>0):
    moves.append(emptyjug2(currstate))
  if(x<capacity[0]):
    moves.append(filljug1(currstate,capacity))
  if(y<capacity[1]):
    moves.append(filljug2(currstate,capacity))
  if(x>0 and y<capacity[1]):
    moves.append(pour1to2(currstate,capacity))
  if(y>0 and x<capacity[0]):
    moves.append(pour2to1(currstate,capacity))
  for move in moves:
    next_state = move[:-1]          # taking without the description
    next_move = move[-1]            # taking the description
    if next_state not in visited:
      sequence.append(next_move)
      if depthFirstSearch(next_state,capacity,target,visited,sequence):
        return True
      sequence.pop()                # backtrack if the solution is not found
  return False

def twoJugProblem(capacity,target):
  visited = set()
  currstate = (0,0)                 # initial state
  sequence=[]
  result = depthFirstSearch(currstate,capacity,target,visited,sequence)
  if result:
     print("Solution is found")
     for step, move in enumerate(sequence,start=1):
      print("Step",step,":",move)
  else:
     print("Solution not found")
```

Testing the code-

```
capacity = (6,4)
target = (2,4)
twoJugProblem(capacity,target)
```

Output -

```
Solution is found
Step 1 : Fill jug1
Step 2 : Fill jug2
Step 3 : Empty jug1
Step 4 : Pour from jug2 to jug1
Step 5 : Fill jug2
Step 6 : Pour from jug2 to jug1
Step 7 : Empty jug1
Step 8 : Pour from jug2 to jug1
Step 9 : Fill jug2
```

Testing the code with another number-

```
capacity = (5,7)
target = (3,4)
twoJugProblem(capacity,target)
```

Output -

```
Solution not found
```

5. **Develop a program to execute Breadth First Search (BFS) for solving the Tic-Tac-Toe problem. In the game of Tic Tac Toe, also recognized as Noughts and Crosses or Xs and Os, two players alternate turns to place their marks (X or O) in a 3x3 grid. The objective is to achieve three consecutive marks either horizontally, vertically, or diagonally, resulting in a victory for the respective player. For this implementation, Player 1 is represented by the mark 'X' and Player 2 by 'O'. The program should employ a Breadth First Search algorithm to systematically explore the game tree, considering possible moves and board configurations, and ultimately determining an optimal strategy for the players.**

The program to develop a startegy for the Tic-Tac-Toe is given below.

```
from collections import deque

# Constants
EMPTY, PLAYER_X, PLAYER_O = 0, 1, 2
BOARD_SIZE = 3
MOVE_TABLE_SIZE = 39

# Convert board to a string for indexing the move table
def board_to_string(board):
    return ''.join(str(cell) for row in board for cell in row)

# Convert string to a list for updating the game board
def string_to_board(string):
    return [[int(string[i * BOARD_SIZE + j]) for j in range(BOARD_SIZE)] for i in range(BOARD_SIZE)]

# Display the Tic-Tac-Toe board
def display_board(board):
    for row in board:
        print(row)
    print()

# Perform a move based on the move table
def make_move(board, index):
    new_board_str = MOVE_TABLE[index]
    return string_to_board(new_board_str)

# Breadth-First Search to find optimal strategy
def ticTacToe():
    start_board = [[EMPTY] * BOARD_SIZE for _ in range(BOARD_SIZE)]
    visited = set()
    queue = deque([(start_board, PLAYER_X)])

    while queue:
        current_board, current_player = queue.popleft()
        current_board_str = board_to_string(current_board)

        if current_board_str in visited:
            continue
        visited.add(current_board_str)

        display_board(current_board)

        if is_winner(current_board, PLAYER_X):
            print("PLAYER 1 WINS")
            break
        elif is_winner(current_board, PLAYER_O):
```

```
            print("PLAYER 2 WINS")
            break
        elif all(cell != EMPTY for row in current_board for cell in row):
            print("DRAW")
            break

        for i in range(MOVE_TABLE_SIZE):
            next_board = make_move(current_board, i)
            next_player = PLAYER_X if current_player == PLAYER_O else PLAYER_O
            queue.append((next_board, next_player))

# Check if the current player has won
def is_winner(board, player):
    # Check rows, columns, and diagonals
    for i in range(BOARD_SIZE):
        if all(board[i][j] == player for j in range(BOARD_SIZE)) or all(board[j][i] == player for j in range
            return True

    if all(board[i][i] == player for i in range(BOARD_SIZE)) or all(board[i][BOARD_SIZE-1-i] == player for i
        return True

    return False
```

Testing the code-

```
# Move Table
MOVE_TABLE = [
    "000010000", "020000001", "000100002",
    "002000010", "010000100", "000001020",
    "001000200", "000020000", "000000001",
    "000010002", "020000010", "000100020",
    "010000001", "020001000", "000102000",
    "001000002", "000010020", "020000100",
    "000001002", "002000020", "010000010",
    "001000000", "000000020", "000100001",
    "002000100", "000010000", "001020000",
    "002010000", "000001000", "010000020",
    "000001200", "002000010", "000100000",
    "010000002", "020000100", "000010200",
    "002000010", "000100020", "010000001"
]

# Run the BFS Tic-Tac-Toe Solver
ticTacToe()
```

Output -

```
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]

[0, 0, 0]
[0, 1, 0]
[0, 0, 0]

[0, 2, 0]
[0, 0, 0]
[0, 0, 1]
```

```
[0, 0, 0]
[1, 0, 0]
[0, 0, 2]

[0, 0, 2]
[0, 0, 0]
[0, 1, 0]

[0, 1, 0]
[0, 0, 0]
[1, 0, 0]

[0, 0, 0]
[0, 0, 1]
[0, 2, 0]

[0, 0, 1]
[0, 0, 0]
[2, 0, 0]

[0, 0, 0]
[0, 2, 0]
[0, 0, 0]

[0, 0, 0]
[0, 0, 0]
[0, 0, 1]

[0, 0, 0]
[0, 1, 0]
[0, 0, 2]

[0, 2, 0]
[0, 0, 0]
[0, 1, 0]

[0, 0, 0]
[1, 0, 0]
[0, 2, 0]

[0, 1, 0]
[0, 0, 0]
[0, 0, 1]

[0, 2, 0]
[0, 0, 1]
[0, 0, 0]

[0, 0, 0]
[1, 0, 2]
[0, 0, 0]

[0, 0, 1]
[0, 0, 0]
[0, 0, 2]

[0, 0, 0]
[0, 1, 0]
```

```
[0, 2, 0]

[0, 2, 0]
[0, 0, 0]
[1, 0, 0]

[0, 0, 0]
[0, 0, 1]
[0, 0, 2]

[0, 0, 2]
[0, 0, 0]
[0, 2, 0]

[0, 1, 0]
[0, 0, 0]
[0, 1, 0]

[0, 0, 1]
[0, 0, 0]
[0, 0, 0]

[0, 0, 0]
[0, 0, 0]
[0, 2, 0]

[0, 0, 0]
[1, 0, 0]
[0, 0, 1]

[0, 0, 2]
[0, 0, 0]
[1, 0, 0]

[0, 0, 1]
[0, 2, 0]
[0, 0, 0]

[0, 0, 2]
[0, 1, 0]
[0, 0, 0]

[0, 0, 0]
[0, 0, 1]
[0, 0, 0]

[0, 1, 0]
[0, 0, 0]
[0, 2, 0]

[0, 0, 0]
[0, 0, 1]
[2, 0, 0]

[0, 0, 0]
[1, 0, 0]
[0, 0, 0]
```

```
[0, 1, 0]
[0, 0, 0]
[0, 0, 2]

[0, 0, 0]
[0, 1, 0]
[2, 0, 0]
```

6. **You are the proprietor of Sammy's Sport Shop. You have just received a shipment of three boxes filled with tennis ball. One box contains only yellow tennis balls, one box contains only white tennis balls, and one contains both yellow and white tennis balls. You would like to stock the tennis balls in appropriate places on your shelves. Unfortunately, the boxes have been labeled incorrectly; the manufacturer tells you that you have exactly one box of each, but that each box is labeled wrong. One ball is drawn from each box and observed (assumed to be correct). Given the initial (incorrect) labeling of the boxes, and the three observations, use Propositional Logic to device the correct labeling of the middle box.**

The meaning of the various observations, labels and problem constraints are given below.

Implications of observations:

- OnY $\implies$ CnY $\vee$ CnB $\forall$ n, n = 1,2,3
- OnW $\implies$ CnW $\vee$ CnB $\forall$ n, n = 1,2,3

Implications of labelling:

- LnY $\implies$ $\neg$ CnY $\forall$ n, n = 1,2,3
- LnW $\implies$ $\neg$ CnW $\forall$ n, n = 1,2,3
- LnB $\implies$ $\neg$ CnB $\forall$ n, n = 1,2,3

Implication of the constraint that there is only one box of each color:

- C1Y $\vee$ C1W $\vee$ C1B
- C2Y $\vee$ C2W $\vee$ C2B
- C3Y $\vee$ C3W $\vee$ C3B

Implication of the constraint that there are no two boxes of the same color:

- C1p $\implies$ ($\neg$ C2p) $\wedge$ ($\neg$ C3p) $\forall$ p, p = Y,W,B
- C2p $\implies$ ($\neg$ C3p) $\wedge$ ($\neg$ C1p) $\forall$ p, p = Y,W,B
- C3p $\implies$ ($\neg$ C1p) $\wedge$ ($\neg$ C2p) $\forall$ p, p = Y,W,B

Proof that box 2 must contain white balls:

- from O3Y derive C3Y $\vee$ C3B by *Modus Ponens* ... (I)
- from L3B derive $\neg$ C3B by *Modus Ponens* ... (II)
- from (I) and (II), derive C3Y by *Resolution* ... (III)
- from (III), derive ($\neg$ C1Y) $\wedge$ ($\neg$ C2Y) by *Modus Ponens* ... (IV)
- from O1Y, derive CnY $\vee$ CnB by *Modus Ponens* ... (V)
- from (IV), derive ($\neg$ C1Y) by *Conjunction Elimination* ... (VI)
- from (V) and (VI), derive C1B by *Resolution* ... (VII)
- from O2W, derive C2W $\vee$ C2B by *Modus Ponens* ... (VIII)
- from (VII), derive ($\neg$ C2B) $\wedge$ ($\neg$ C3B) by *Modus Ponens* ... (IX)
- from (IX), derive ($\neg$ C2B) by *Conjunction Elimination* ... (X)
- from (VIII) and (X), derive C2W by *Resolution* ... **[Proved]**