# Practical File

On

# Artificial Intelligence Lab (KCA-351)

## Session 2021-22

****

**Department of Computer Applications**

## KIET Group of Institutions

### Submitted To:

Prof S.D.Mishra

Department of Computer Applications KIET Group of Institutions

**Submitted By**:

TARUNIMA SHARMA

2000290140126

**Index**

#### INSTRUCTIONS -:

1. Program must be written inPython
2. Algorithm and program both arecompulsory.
3. One pilot project is mandatory for individual student example Mini Alexa/Gammingproject

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No.** | **Program List** | **Due Date** | **Completion Date** | **Faculty Signature** |
| **Ex-1** | Installation of Python/Anaconda and introduction of Google Colab | First Week |  |  |
| **Ex-2** | Write a program to implement water jug problem' | First Week |  |  |
| **Ex-3** | Write a program to implement missionary and cannibal algorithm | Second Week |  |  |
| **Ex-4** | Write a program to implement breadth first search | Second Week |  |  |
| **Ex-5** | Write a program to implement depth first search | Third Week |  |  |
| **Ex-6** | Write a program to implement 8 Queen Problems | Third Week |  |  |
| **Ex-7** | Write a program to implement first order logic | Fourth Week |  |  |
| **Ex-8** | Write a program to demonstrate the working of Bayesian network. | Fourth Week |  |  |
| **Ex-9** | Write a program to Implement pattern recognition problems of handwritten character/ digit recognition | Fifth Week |  |  |
| **Ex-10** | Write a program to Implement pattern recognition problems of speech recognition | Sixth Week |  |  |
| **Ex-11** | Write a program to implement naive bayes classification problem | Seventh Week |  |  |
| **Ex-12** | Write a program to implement k-mean clustering problem | Eight Week |  |  |
| **Ex-13** | Write a program to convert text to speech using NLP tool | Ninth Week |  |  |
| **Ex-14** | Pilot Project in AI using Python | Tenth Week |  |  |

**Experiment 2**:Write a program to implement water jug problem'

#### Problem Formulation: There are two jugs of volume A litre and B litre.Neither has any measuring mark on it.There is a pump that can be used to fill the jugs with water.How can you get exactly x litre of water into the A litre jug. Assuming that we have unlimited supply of water.

**Algorithm**:

**Python Program**:

x = 0

y = 0

p = 5

q = 6

print("initial state = (0,0)")

print("capacities = (5,6)")

print("goal state = (2,y)")

while x != 2:

r = int(input("Enter Rule:"))

if(r == 1):

x = p

elif(r == 2):

y = q

elif(r == 3):

x = 0

elif(r == 4):

y = 0

elif(r == 5):

t = q - y

y = q

x -= t

elif(r == 6):

t = p - x

x = p

y -= t

elif(r == 7):

y += x

x = 0

elif(r == 8):

x +=y

y = 0

print (x,y)

**Program Output**:

**Experiment 3**:Write a program to implement missionary and cannibal algorithm

#### Problem Formulation:In this problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, that the missionaries present on the bank cannot be outnumbered by cannibals. The boat cannot cross the river by itself with no people on board.

**Algorithm**:

First let us consider that both the missionaries (M) and cannibals(C) are on the same side of the river.  
 Left Right  
 Initially the positions are : 0M , 0C and 3M , 3C (B)

Now let’s send 2 Cannibals to left of bank : 0M , 2C (B) and 3M , 1C

Send one cannibal from left to right : 0M , 1C and 3M , 2C (B)

Now send the 2 remaining Cannibals to left : 0M , 3C (B) and 3M , 0C  
 Send 1 cannibal to the right : 0M , 2C and 3M , 1C (B)

Now send 2 missionaries to the left : 2M , 2C (B) and 1M . 1C

Send 1 missionary and 1 cannibal to right : 1M , 1C and 2M , 2C (B)

Send 2 missionaries to left : 3M , 1C (B) and 0M , 2C

Send 1 cannibal to right : 3M , 0C and 0M , 3C (B)

Send 2 cannibals to left : 3M , 2C (B) and 0M , 1C

Send 1 cannibal to right : 3M , 1C and 0M , 2C (B)’

Send 2 cannibals to left : 3M , 3C (B) and 0M , 0C

• Here (B) shows the position of the boat after the action is performed.  
 Therefore all the missionaries and cannibals have crossed the river safely.

**Python Program**:

Missionary = int(input("Enter the number of the missionary: "))

Cannibal = int(input("Enter the number of the Cannibal: "))

# condition of win is if the Missionary are to the next side without eaten

Place\_2 = [0,0]

moving\_Missionary = Missionary

moving\_Cannibal = Cannibal

rules\_used = {}

count = 0

while Place\_2[0] != Missionary or Place\_2[1] != Cannibal:

if moving\_Cannibal < 0 or moving\_Missionary < 0 or moving\_Cannibal > 3 or moving\_Missionary > 3:

break

game\_rule = int(input("Enter the rule to use: "))

#moving from place 1 to place 2

if game\_rule == 1: #1 M 1 C

moving\_Missionary -= 1

moving\_Cannibal -= 1

Place\_2[0] += 1

Place\_2[1] += 1

elif game\_rule == 2: # 2 M

moving\_Missionary -= 2

Place\_2[0] += 2

elif game\_rule == 3: # 2 C

moving\_Cannibal -= 2

Place\_2[1] += 2

#moving from place 2 to place 1

elif game\_rule == 4: # 1 M

moving\_Missionary -= 1

elif game\_rule == 5: # 1 C

moving\_Cannibal += 1

Place\_2[1] -= 1

elif game\_rule == 6: #1 M 1C

moving\_Missionary +=1

moving\_Cannibal += 1

Place\_2[0] -= 1

Place\_2[1] -= 1

elif game\_rule == 7: # 2 M

moving\_Missionary += 2

Place\_2[0] -= 2

elif game\_rule == 8: # 2 C

moving\_Cannibal -= 2

print(f"Missionary = {moving\_Missionary} and Cannibal = {moving\_Cannibal} in place 1")

print(f"Missionary = {Place\_2[0]} and Cannibal = {Place\_2[1]} in place 2")

if game\_rule in rules\_used.keys():

rules\_used[game\_rule] += 1

else:

rules\_used[game\_rule] = 1

count += 1

print(f"Steps Needed: {count}")

for rule in rules\_used.keys():

print(f"Rule {rule} is used {rules\_used[rule]} times")

**Program Output**:

**Experiment 4**:Write a program to implement breadth first search

#### Problem Formulation:[Breadth First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/level-order-tree-traversal/)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

**Algorithm**:

Rule 1− Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2− If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3− Repeat Rule 1 and Rule 2 until the queue is empty.

**Python Program**:

graph = {

'1': ['2', '3'],

'2': ['4', '5', '1'],

'3': ['6', '7', '1'],

'4': ['2'],

'5': ['6', '2'],

'6': ['3'],

'7': ['3']

}

visited = []

queue = []

def Bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

s = queue.pop(0)

print(s, end=" ")

for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

print("The Traversal is:")

Bfs(visited, graph, '2')

**Program Output**:

**Experiment 5**:Write a program to implement depth first search

#### Problem Formulation:Depth First Search for a graph is similar to [Depth First Traversal of a tree.](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

**Algorithm**:

1.Create a recursive function that takes the index of the node and a visited array.

2.Mark the current node as visited and print the node.

3.Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

**Python Program**:

graph = {

'1': ['2', '3'],

'2': ['4', '5', '1'],

'3': ['6', '7', '1'],

'4': ['2'],

'5': ['6', '2'],

'6': ['3'],

'7': ['3']

}

visited = []

stack = []

def Dfs(visited, graph, node):

visited.append(node)

stack.append(node)

while queue:

s = stack.pop()

print(s, end=" ")

for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

stack.append(neighbour)

print("The Traversal is:")

Dfs(visited, graph, '1')

**Program Output**:

**Experiment 6**:Write a program to implement 4 Queen Problems

#### Problem Formulation:The4-Queens Problem consists in placing four queens on a 4 x 4 chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column or the same diagonal.

**Algorithm**:

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing the queen in [row, column] leads to

a solution then return true.

c) If placing queen doesn't lead to a solution then

unmark this [row, column] (Backtrack) and go to

step (a) to try other rows.

3) If all rows have been tried and nothing worked,

return false to trigger backtracking.

**Python Program**:

graph={

'A':['B','D','E','C'],

'B':['F','G'],

'C':['H'],

'D':['I'],

'E':['J','K'],

'F':['L'],

'G':['M'],

'H':['N',],

'I':['O',],

'J':['P',],

'K':['Q',],

'L':[],

'M':['R'],

'N':['S'],

'O':['T'],

'P':['U'],

'Q':[],

'R':[],

'S':[],

'T':[],

'U':[]

}

Traversal=[]

visited=[]

queue=[]

def Bfs(graph,start,target,path):

queue.append(start)

while queue:

s=queue.pop(0)

path.append(s)

visited.append(s)

if s==target:

return path

for neighbour in graph[s]:

if neighbour not in visited:

queue.append(neighbour)

print("The four queen path is:")

Traversal=Bfs(graph,'A','S',Traversal)

print(Traversal)

**Program Output**: