**Experiment-2**

**Aim -** 8 Puzzle Single Player Game (Breadth First Search)

**Code-**

# Import the necessary libraries

from time import time

from queue import Queue

# Creating a class Puzzle

class Puzzle:

# Setting the goal state of 8-puzzle

goal\_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]

num\_of\_instances = 0

# constructor to initialize the class members

def \_\_init\_\_(self, state, parent, action):

self.parent = parent

self.state = state

self.action = action

# Incrementing the number of instances by 1

Puzzle.num\_of\_instances += 1

# function used to display a state of 8-puzzle

def \_\_str\_\_(self):

return str(self.state[0:3]) + '\n' + str(self.state[3:6]) + '\n' + str(self.state[6:9])

# method to compare the current state with the goal state

def goal\_test(self):

# Comparing the current state with the goal state

if self.state == Puzzle.goal\_state:

return True

return False

# static method to find the legal action based on the current board position

@staticmethod

def find\_legal\_actions(i, j):

legal\_action = ['U', 'D', 'L', 'R']

if i == 0:

# if row is 0 in board, then 'U' (Up) is disabled

legal\_action.remove('U')

elif i == 2:

# if row is 2 in board, then 'D' (Down) is disabled

legal\_action.remove('D')

if j == 0:

# if column is 0, then 'L' (Left) is disabled

legal\_action.remove('L')

elif j == 2:

# if column is 2, then 'R' (Right) is disabled

legal\_action.remove('R')

return legal\_action

# method to generate the child of the current state of the board

def generate\_child(self):

# Create an empty list for children

children = []

x = self.state.index(0)

i = int(x / 3)

j = int(x % 3)

# Find the legal actions based on i and j values

legal\_actions = self.find\_legal\_actions(i, j)

# Iterate over all legal actions

for action in legal\_actions:

new\_state = self.state.copy()

# If the legal action is UP

if action == 'U':

# Swapping between current index of 0 with its up element on the board

new\_state[x], new\_state[x - 3] = new\_state[x - 3], new\_state[x]

elif action == 'D':

# Swapping between current index of 0 with its down element on the board

new\_state[x], new\_state[x + 3] = new\_state[x + 3], new\_state[x]

elif action == 'L':

# Swapping between the current index of 0 with its left element on the board

new\_state[x], new\_state[x - 1] = new\_state[x - 1], new\_state[x]

elif action == 'R':

# Swapping between the current index of 0 with its right element on the board

new\_state[x], new\_state[x + 1] = new\_state[x + 1], new\_state[x]

children.append(Puzzle(new\_state, self, action))

# Return the children

return children

# method to find the solution

def find\_solution(self):

solution = []

solution.append(self.action)

path = self

while path.parent != None:

path = path.parent

solution.append(path.action)

solution = solution[:-1]

solution.reverse()

return solution

# method for breadth first search

def breadth\_first\_search(initial\_state):

start\_node = Puzzle(initial\_state, None, None)

print("Initial state:")

print(start\_node)

if start\_node.goal\_test():

return start\_node.find\_solution()

q = Queue()

q.put(start\_node)

explored = []

# Iterate the queue until empty

while not q.empty():

node = q.get()

# Append the state of node in the explored list

explored.append(node.state)

# Generate the child nodes of the current node

children = node.generate\_child()

# Iterate over each child node in children

for child in children:

if child.state not in explored:

if child.goal\_test():

return child.find\_solution()

q.put(child)

return None

# Start executing the 8-puzzle with setting up the initial state

state = [

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

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

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

]

# Iterate over number of initial states

for i in range(0, 3):

# Initialize the num\_of\_instances to zero

Puzzle.num\_of\_instances = 0

# Set t0 to current time

t0 = time()

# Call breadth\_first\_search

bfs = breadth\_first\_search(state[i])

# Get the time t1 after executing the breadth\_first\_search method

t1 = time() - t0

# Output the result of BFS, the space used (number of instances created), and the time taken

print('BFS Solution:', bfs)

print('Space (number of instances):', Puzzle.num\_of\_instances)

print('Time taken:', t1)

print()

print('------------------------------------------')

**Output –**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Experiment-3**

**Aim -**8 Puzzle Single Player Game (A\* Algorithm)

**Code-**

from time import time

from queue import PriorityQueue

import math

class Puzzle:

goal\_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]

heuristic = None

evaluation\_function = None

needs\_heuristic = False

num\_of\_instances = 0

def \_\_init\_\_(self, state, parent, action, path\_cost, needs\_heuristic=False):

self.parent = parent

self.state = state

self.action = action

if parent:

self.path\_cost = parent.path\_cost + path\_cost

else:

self.path\_cost = path\_cost

if needs\_heuristic:

self.needs\_heuristic = True

self.generate\_heuristic()

self.evaluation\_function = self.path\_cost + self.heuristic

else:

self.evaluation\_function = self.path\_cost

Puzzle.num\_of\_instances += 1

def \_\_str\_\_(self):

return str(self.state[0:3]) + '\n' + str(self.state[3:6]) + '\n' + str(self.state[6:9])

def generate\_heuristic(self):

self.heuristic = 0

for num in range(1, 9):

distance = self.state.index(num)

goal\_index = Puzzle.goal\_state.index(num)

i = int(distance / 3)

j = int(distance % 3)

goal\_i = int(goal\_index / 3)

goal\_j = int(goal\_index % 3)

self.heuristic += abs(i - goal\_i) + abs(j - goal\_j)

def goal\_test(self):

if self.state == Puzzle.goal\_state:

return True

return False

@staticmethod

def find\_legal\_actions(i, j):

legal\_action = ['U', 'D', 'L', 'R']

if i == 0:

legal\_action.remove('U')

elif i == 2:

legal\_action.remove('D')

if j == 0:

legal\_action.remove('L')

elif j == 2:

legal\_action.remove('R')

return legal\_action

def generate\_child(self):

children = []

x = self.state.index(0)

i = x // 3

j = x % 3

legal\_actions = Puzzle.find\_legal\_actions(i, j)

for action in legal\_actions:

new\_state = self.state.copy()

if action == 'U':

new\_state[x], new\_state[x - 3] = new\_state[x - 3], new\_state[x]

elif action == 'D':

new\_state[x], new\_state[x + 3] = new\_state[x + 3], new\_state[x]

elif action == 'L':

new\_state[x], new\_state[x - 1] = new\_state[x - 1], new\_state[x]

elif action == 'R':

new\_state[x], new\_state[x + 1] = new\_state[x + 1], new\_state[x]

children.append(Puzzle(new\_state, self, action, 1, True))

return children

def find\_solution(self):

solution = []

solution.append(self.action)

path = self

while path.parent is not None:

path = path.parent

solution.append(path.action)

solution = solution[:-1]

solution.reverse()

return solution

def Astar\_search(initial\_state):

count = 0

explored = []

start\_node = Puzzle(initial\_state, None, None, 0, True)

q = PriorityQueue()

q.put((start\_node.evaluation\_function, count, start\_node))

while not q.empty():

\_, \_, node = q.get()

explored.append(node.state)

if node.goal\_test():

return node.find\_solution()

children = node.generate\_child()

for child in children:

if child.state not in explored:

count += 1

q.put((child.evaluation\_function, count, child))

return None

state = [

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

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

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

]

for i in range(0, 3):

Puzzle.num\_of\_instances = 0

t0 = time()

astar = Astar\_search(state[i])

t1 = time() - t0

print('A\* Solution:', astar)

print('Space (number of instances):', Puzzle.num\_of\_instances)

print('Time taken:', t1)

print()

print('------------------------------------------')

**Output –**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Experiment-4**

**Aim-** WATER JUG PROBLEM USING BFS & DFS

**Code -**

import collections

def get\_index(node):

return pow(7, node[0]) \* pow(5, node[1])

def get\_search\_type():

s = input("Enter 'b' for BFS, 'd' for DFS: ")

s = s.lower()

while s != 'b' and s != 'd':

s = input("The input is not valid! Enter 'b' for BFS, 'd' for DFS: ").lower()

return True if s == 'b' else False

def get\_jugs():

print("Receiving the volume of the jugs...")

jugs = []

temp = int(input("Enter first jug volume (>1): "))

while temp < 1:

temp = int(input("Enter a valid amount (>1): "))

jugs.append(temp)

temp = int(input("Enter second jug volume (>1): "))

while temp < 1:

temp = int(input("Enter a valid amount (>1): "))

jugs.append(temp)

return jugs

def get\_goal(jugs):

print("Receiving the desired amount of the water...")

max\_amount = max(jugs)

s = "Enter the desired amount of water (1 - {0}): ".format(max\_amount)

goal\_amount = int(input(s))

while goal\_amount < 1 or goal\_amount > max\_amount:

goal\_amount = int(input("Enter a valid amount (1 - {0}): ".format(max\_amount)))

return goal\_amount

def is\_goal(path, goal\_amount):

print("Checking if the goal is achieved...")

return path[-1][0] == goal\_amount or path[-1][1] == goal\_amount

def been\_there(node, check\_dict):

print("Checking if {0} is visited before...".format(node))

return check\_dict.get(tuple(node), False)

def next\_transitions(jugs, path, check\_dict):

print("Finding next transitions and checking for the loops...")

result = []

next\_nodes = []

node = []

a\_max = jugs[0]

b\_max = jugs[1]

a = path[-1][0]

b = path[-1][1]

node.append(a\_max)

node.append(b)

if not been\_there(node, check\_dict):

next\_nodes.append(node)

node = []

node.append(a)

node.append(b\_max)

if not been\_there(node, check\_dict):

next\_nodes.append(node)

node = []

node.append(min(a\_max, a + b))

node.append(b - (node[0] - a))

if not been\_there(node, check\_dict):

next\_nodes.append(node)

node = []

node.append(a - (min(a + b, b\_max) - b))

node.insert(0, min(a + b, a\_max))

if not been\_there(node, check\_dict):

next\_nodes.append(node)

node = []

node.append(0)

node.append(b)

if not been\_there(node, check\_dict):

next\_nodes.append(node)

node = []

node.append(a)

node.append(0)

if not been\_there(node, check\_dict):

next\_nodes.append(node)

for i in range(0, len(next\_nodes)):

temp = list(path)

temp.append(next\_nodes[i])

result.append(temp)

if len(next\_nodes) == 0:

print("No more unvisited nodes...\nBacktracking...")

else:

print("Possible transitions: ")

for nnode in next\_nodes:

print(nnode)

return result

def transition(old, new, jugs):

a = old[0]

b = old[1]

a\_prime = new[0]

b\_prime = new[1]

a\_max = jugs[0]

b\_max = jugs[1]

if a > a\_prime:

if b == b\_prime:

return "Clear {0}-liter jug:\t\t\t".format(a\_max)

else:

return "Pour {0}-liter jug into {1}-liter jug:\t".format(a\_max, b\_max)

else:

if b > b\_prime:

if a == a\_prime:

return "Clear {0}-liter jug:\t\t\t".format(b\_max)

else:

return "Pour {0}-liter jug into {1}-liter jug:\t".format(b\_max, a\_max)

else:

if a == a\_prime:

return "Fill {0}-liter jug:\t\t\t".format(b\_max)

else:

return "Fill {0}-liter jug:\t\t\t".format(a\_max)

def print\_path(path, jugs):

print("Starting from:\t\t\t\t", path[0])

for i in range(0, len(path) - 1):

print(i + 1, ":", transition(path[i], path[i + 1], jugs), path[i + 1])

def search(starting\_node, jugs, goal\_amount, check\_dict, is\_breadth):

if is\_breadth:

print("Implementing BFS...")

else:

print("Implementing DFS...")

goal = []

accomplished = False

q = collections.deque()

q.appendleft(starting\_node)

while len(q) != 0:

path = q.popleft()

check\_dict[get\_index(path[-1])] = True

if len(path) >= 2:

print(transition(path[-2], path[-1], jugs), path[-1])

if is\_goal(path, goal\_amount):

accomplished = True

goal = path

break

next\_moves = next\_transitions(jugs, path, check\_dict)

for i in next\_moves:

if is\_breadth:

q.append(i)

else:

q.appendleft(i)

if accomplished:

print("The goal is achieved\nPrinting the sequence of the moves...\n")

print\_path(goal, jugs)

else:

print("Problem cannot be solved.")

if \_\_name\_\_ == '\_\_main\_\_':

starting\_node = [[0, 0]]

jugs = get\_jugs()

goal\_amount = get\_goal(jugs)

check\_dict = {}

is\_breadth = get\_search\_type()

search(starting\_node, jugs, goal\_amount, check\_dict, is\_breadth)

**Experiment-5**

**Aim-**Tic-Tac-Toe Game using Min-Max Algorithm

**Code-**

import numpy as np

from math import inf as infinity

# Set the Empty Board

game\_state = [[' ',' ',' '],

[' ',' ',' '],

[' ',' ',' ']]

# Create the Two Players as 'X'/'O'

players = ['X', 'O']

# Method for checking the correct move on Tic-Tac-Toe

def play\_move(state, player, block\_num):

if state[int((block\_num-1)/3)][(block\_num-1)%3] == ' ':

state[int((block\_num-1)/3)][(block\_num-1)%3] = player

else:

block\_num = int(input("Block is not empty, ya blockhead! Choose again: "))

play\_move(state, player, block\_num)

# Method to copy the current game state to new\_state of Tic-Tac-Toe

def copy\_game\_state(state):

new\_state = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]

for i in range(3):

for j in range(3):

new\_state[i][j] = state[i][j]

return new\_state

# Method to check the current state of the Tic-Tac-Toe

def check\_current\_state(game\_state):

draw\_flag = 1

for i in range(3):

for j in range(3):

if game\_state[i][j] == ' ':

draw\_flag = 0

if draw\_flag == 1:

return None, "Draw"

# Check horizontals

for i in range(3):

if game\_state[i][0] == game\_state[i][1] == game\_state[i][2] and game\_state[i][0] != ' ':

return game\_state[i][0], "Done"

# Check verticals

for j in range(3):

if game\_state[0][j] == game\_state[1][j] == game\_state[2][j] and game\_state[0][j] != ' ':

return game\_state[0][j], "Done"

# Check diagonals

if game\_state[0][0] == game\_state[1][1] == game\_state[2][2] and game\_state[0][0] != ' ':

return game\_state[0][0], "Done"

if game\_state[0][2] == game\_state[1][1] == game\_state[2][0] and game\_state[0][2] != ' ':

return game\_state[0][2], "Done"

return None, "Not Done"

# Method to print the Tic-Tac-Toe Board

def print\_board(game\_state):

print('----------------')

for row in game\_state:

print('| ' + ' || '.join(row) + ' |')

print('----------------')

# Method for implementing the Minimax Algorithm

def getBestMove(state, player):

winner\_loser, done = check\_current\_state(state)

if done == "Done" and winner\_loser == 'O':

return 1

elif done == "Done" and winner\_loser == 'X':

return -1

elif done == "Draw":

return 0

moves = []

empty\_cells = []

for i in range(3):

for j in range(3):

if state[i][j] == ' ':

empty\_cells.append(i\*3 + (j+1))

for empty\_cell in empty\_cells:

move = {}

move['index'] = empty\_cell

# Copy the game state

new\_state = copy\_game\_state(state)

# Simulate the move

play\_move(new\_state, player, empty\_cell)

if player == 'O':

result = getBestMove(new\_state, 'X')

move['score'] = result

else:

result = getBestMove(new\_state, 'O')

move['score'] = resul

moves.append(move)

# Find best move

best\_move = None

if player == "O": # Computer's turn

best = -infinity

for move in moves:

if move['score'] > best:

best = move['score']

best\_move = move['index']

else: # Human's turn

best = infinity

for move in moves:

if move['score'] < best:

best = move['score']

best\_move = move['index']

return best\_move

# Now Playing the Tic-Tac-Toe Game

play\_again = 'Y'

while play\_again == 'Y' or play\_again == 'y':

game\_state = [[' ',' ',' '],

[' ',' ',' '],

[' ',' ',' ']]

current\_state = "Not Done"

print("\nNew Game!")

print\_board(game\_state)

player\_choice = input("Choose which player goes first - X (You) or O(Computer): ")

winner = None

if player\_choice == 'X' or player\_choice == 'x':

current\_player\_idx = 0

else:

current\_player\_idx = 1

while current\_state == "Not Done":

if current\_player\_idx == 0: # Human's turn

block\_choice = int(input("Your turn please! Choose where to place (1 to 9): "))

play\_move(game\_state, players[current\_player\_idx], block\_choice)

else: # Computer's turn

block\_choice = getBestMove(game\_state, players[current\_player\_idx])

play\_move(game\_state, players[current\_player\_idx], block\_choice)

print("AI plays move: " + str(block\_choice))

print\_board(game\_state)

winner, current\_state = check\_current\_state(game\_state)

if winner is not None:

print(str(winner) + " won!")

else:

current\_player\_idx = (current\_player\_idx + 1) % 2

if current\_state == "Draw":

print("Draw!")

play\_again = input('Wanna try again?(Y/N) : ')

if play\_again == 'N':

print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')

**Output-**

A white background with black text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Experiment-7**

**Aim -** Constraint Satisfaction Problems

**Code-**

from typing import Generic, TypeVar, Dict, List, Optional

from abc import ABC, abstractmethod

V = TypeVar('V')

D = TypeVar('D')

class Constraint(Generic[V, D], ABC):

def \_\_init\_\_(self, variables: List[V]) -> None:

self.variables = variables

@abstractmethod

def satisfied(self, assignment: Dict[V, D]) -> bool:

pass

class CSP(Generic[V, D]):

def \_\_init\_\_(self, variables: List[V], domains: Dict[V, List[D]]) -> None:

self.variables: List[V] = variables

self.domains: Dict[V, List[D]] = domains

self.constraints: Dict[V, List[Constraint[V, D]]] = {}

for variable in self.variables:

self.constraints[variable] = []

if variable not in self.domains:

raise LookupError("Every variable should have a domain assigned to it.")

def add\_constraint(self, constraint: Constraint[V, D]) -> None:

for variable in constraint.variables:

if variable not in self.variables:

raise LookupError("Variable in constraint not in CSP")

else:

self.constraints[variable].append(constraint)

def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:

for constraint in self.constraints[variable]:

if not constraint.satisfied(assignment):

return False

return True

def backtracking\_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:

if len(assignment) == len(self.variables):

return assignment

unassigned: List[V] = [v for v in self.variables if v not in assignment]

first: V = unassigned[0]

for value in self.domains[first]:

local\_assignment = assignment.copy()

local\_assignment[first] = value

if self.consistent(first, local\_assignment):

result: Optional[Dict[V, D]] = self.backtracking\_search(local\_assignment)

if result is not None:

return result

return None

class MapColoringConstraint(Constraint[str, str]):

def \_\_init\_\_(self, place1: str, place2: str) -> None:

super().\_\_init\_\_([place1, place2])

self.place1: str = place1

self.place2: str = place2

def satisfied(self, assignment: Dict[str, str]) -> bool:

if self.place1 not in assignment or self.place2 not in assignment:

return True

return assignment[self.place1] != assignment[self.place2]

if \_\_name\_\_ == "\_\_main\_\_":

variables: List[str] = ["BOX\_1", "BOX\_2", "BOX\_4", "BOX\_3", "BOX\_5", "BOX\_6", "BOX\_7"]

domains: Dict[str, List[str]] = {}

for variable in variables:

domains[variable] = ["red", "green", "blue"]

csp: CSP[str, str] = CSP(variables, domains)

csp.add\_constraint(MapColoringConstraint("BOX\_1", "BOX\_2"))

csp.add\_constraint(MapColoringConstraint("BOX\_1", "BOX\_4"))

csp.add\_constraint(MapColoringConstraint("BOX\_4", "BOX\_2"))

csp.add\_constraint(MapColoringConstraint("BOX\_3", "BOX\_2"))

csp.add\_constraint(MapColoringConstraint("BOX\_3", "BOX\_4"))

csp.add\_constraint(MapColoringConstraint("BOX\_3", "BOX\_5"))

csp.add\_constraint(MapColoringConstraint("BOX\_5", "BOX\_4"))

csp.add\_constraint(MapColoringConstraint("BOX\_6", "BOX\_4"))

csp.add\_constraint(MapColoringConstraint("BOX\_6", "BOX\_5"))

csp.add\_constraint(MapColoringConstraint("BOX\_6", "BOX\_7"))

solution: Optional[Dict[str, str]] = csp.backtracking\_search()

if solution is None:

print("No solution found!")

else:

print(solution)

class SendMoreMoneyConstraint(Constraint[str, int]):

def \_\_init\_\_(self, letters: List[str]) -> None:

super().\_\_init\_\_(letters)

self.letters: List[str] = letters

def satisfied(self, assignment: Dict[str, int]) -> bool:

if len(set(assignment.values())) < len(assignment):

return False

if len(assignment) == len(self.letters):

s: int = assignment["S"]

e: int = assignment["E"]

n: int = assignment["N"]

d: int = assignment["D"]

m: int = assignment["M"]

o: int = assignment["O"]

r: int = assignment["R"]

y: int = assignment["Y"]

send: int = s \* 1000 + e \* 100 + n \* 10 + d

more: int = m \* 1000 + o \* 100 + r \* 10 + e

money: int = m \* 10000 + o \* 1000 + n \* 100 + e \* 10 + y

return send + more == money

return True

if \_\_name\_\_ == "\_\_main\_\_":

letters: List[str] = ["S", "E", "N", "D", "M", "O", "R", "Y"]

possible\_digits: Dict[str, List[int]] = {}

for letter in letters:

possible\_digits[letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

possible\_digits["M"] = [1]

csp: CSP[str, int] = CSP(letters, possible\_digits)

csp.add\_constraint(SendMoreMoneyConstraint(letters))

solution: Optional[Dict[str, int]] = csp.backtracking\_search()

if solution is None:

print("No solution found!")

else:

print(solution)

**Output-**

****

**Experiment-8**

**Aim-**Implement a Knapsack problem using Brute Force Method and Dynamic Programming

**Code-**

from itertools import product

from collections import namedtuple

try:

from itertools import izip

except ImportError:

izip = zip

Reward = namedtuple('Reward', 'name value weight volume')

bagpack = Reward('bagpack', 0, 25.0, 0.25)

items = [Reward('laptop', 3000, 0.3, 0.025), Reward('printer', 1800, 0.2, 0.015), Reward('headphone', 2500, 2.0, 0.002)]

def tot\_value(items\_count):

global items, bagpack

weight = sum(n \* item.weight for n, item in izip(items\_count, items))

volume = sum(n \* item.volume for n, item in izip(items\_count, items))

if weight <= bagpack.weight and volume <= bagpack.volume:

return sum(n \* item.value for n, item in izip(items\_count, items)), -weight, -volume

else:

return -1, 0, 0

def knapsack():

global items, bagpack

max1 = [min(int(bagpack.weight // item.weight), int(bagpack.volume // item.volume)) for item in items]

return max(product(\*[range(n + 1) for n in max1]), key=tot\_value)

max\_items = knapsack()

maxvalue, max\_weight, max\_volume = tot\_value(max\_items)

max\_weight = -max\_weight

max\_volume = -max\_volume

print("The maximum value achievable (by exhaustive search) is %g." % maxvalue)

item\_names = ", ".join(item.name for item in items)

print(" The number of %s items to achieve this is: %s, respectively." % (item\_names, max\_items))

print(" The weight to carry is %.3g, and the volume used is %.3g." % (max\_weight, max\_volume))

**Output-**

**A black text on a white background

AI-generated content may be incorrect.**

**Experiment-9**

**Aim-**Preprocessing Techniques in NLP Using NLTK package

**Code-**