**A**

**LAB REPORT**

**ON**

**Artificial Intelligence**

**By**

**Sayog Rai**

**Exam Roll No:13081/20**



**Submitted to:**

**Mr. Kul Prasad Paudel**

**Department of Management**

**Kantipur College of Management and Information Technology**

In partial fulfillment of the requirements for the Course

Artificial Intelligence

Mid Baneshwor, Kathmandu

October, 2024

Table of Contents

[**1** **Write a program to implement BFS** 2](#_Toc180596398)

[**1.1** **Source Code** 2](#_Toc180596399)

[**1.2** **Output** 4](#_Toc180596400)

[**2** **Write a program to implement DFS** 5](#_Toc180596401)

[**2.1** **Source Code** 5](#_Toc180596402)

[**2.2** **Output** 8](#_Toc180596403)

[**3** **Write a program to implement Tower of Hanoi Problem** 8](#_Toc180596404)

[**3.1** **Source Code** 8](#_Toc180596405)

[**3.2** **Output** 8](#_Toc180596406)

[**4** **Write a program to demonstrate Bidirectional Search** 9](#_Toc180596407)

[**4.1** **Source Code** 9](#_Toc180596408)

[**4.2** **Output** 11](#_Toc180596409)

[**5** **Write a program to demonstrate Greedy Best Search** 12](#_Toc180596410)

[**5.1** **Source code** 12](#_Toc180596411)

[**5.2** **Output** 13](#_Toc180596412)

[**6** **Write a program to demonstrate A\* search algorithm** 14](#_Toc180596413)

[**6.1** **Source code** 14](#_Toc180596414)

[**6.2** **Output** 16](#_Toc180596415)

[**7** **Write a program to demonstrate Hill Climbing Algorithm** 16](#_Toc180596416)

[**7.1** **Source code** 16](#_Toc180596417)

[**7.2** **Output** 17](#_Toc180596418)

[**8** **Write a program to demonstrate min max algorithm** 18](#_Toc180596419)

[**8.1** **Source code** 18](#_Toc180596420)

[**8.2** **Output** 21](#_Toc180596421)

[**9** **Write a program to demonstrate alphabeta** 22](#_Toc180596422)

[**9.1** **Source code** 22](#_Toc180596423)

[**9.2** **Output** 25](#_Toc180596424)

[**10** **Write a program to demonstrate bidirectional search** 26](#_Toc180596425)

[**10.1** **Source code** 26](#_Toc180596426)

[**10.2** **Output** 29](#_Toc180596427)

# **Write a program to implement BFS**

## **Source Code**

from collections import deque

class Node:

def \_\_init\_\_(self, data=0):

self.data = data

self.left = None

self.right = None

class Tree:

def \_\_init\_\_(self):

self.head = None

def is\_empty(self):

return self.head is None

def insert(self, n):

new\_node = Node(n)

if self.is\_empty():

self.head = new\_node

else:

ptr = self.head

prev = None

while ptr:

prev = ptr

if new\_node.data > ptr.data:

ptr = ptr.right

else:

ptr = ptr.left

if new\_node.data > prev.data:

prev.right = new\_node

else:

prev.left = new\_node

def bfs(self):

if self.head is None:

return

queue = deque([self.head])

while queue:

current = queue.popleft()

self.visit(current)

if current.left:

queue.append(current.left)

if current.right:

queue.append(current.right)

@staticmethod

def visit(node):

print(node.data, end=", ")

def search(self, el):

ptr = self.head

lvl = 0

while ptr:

if el == ptr.data:

print(f"{el} Found at level {lvl}!")

return ptr

elif el > ptr.data:

ptr = ptr.right

else:

ptr = ptr.left

lvl += 1

print(f"{el} Not Found!")

return None

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

ob = Tree()

while True:

user\_input = input("Enter a number for BFS (or type 'done' to stop): ")

if user\_input.lower() == 'done':

break

try:

ob.insert(int(user\_input))

except ValueError:

print("Please enter a valid integer or 'done' to stop.")

print("BFS: ", end="")

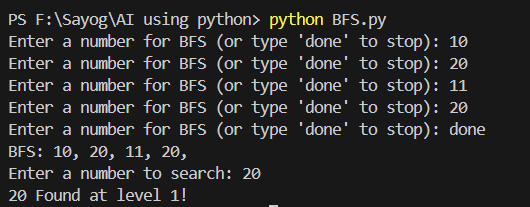
ob.bfs()

print()

search\_number = int(input("Enter a number to search: "))

ob.search(search\_number)

## **Output**



# **Write a program to implement DFS**

## **Source Code**

class Node:

def \_\_init\_\_(self, data=0):

self.data = data

self.left = None

self.right = None

class Tree:

def \_\_init\_\_(self):

self.head = None

def is\_empty(self):

return self.head is None

def insert(self, n):

new\_node = Node(n)

if self.is\_empty():

self.head = new\_node

else:

ptr = self.head

prev = None

while ptr:

prev = ptr

if new\_node.data > ptr.data:

ptr = ptr.right

else:

ptr = ptr.left

if new\_node.data > prev.data:

prev.right = new\_node

else:

prev.left = new\_node

def preorder(self, ptr):

if ptr is not None:

self.visit(ptr)

self.preorder(ptr.left)

self.preorder(ptr.right)

def inorder(self, ptr):

if ptr is not None:

self.inorder(ptr.left)

self.visit(ptr)

self.inorder(ptr.right)

def postorder(self, ptr):

if ptr is not None:

self.preorder(ptr.left)

self.preorder(ptr.right)

self.visit(ptr)

@staticmethod

def visit(node):

print(node.data, end=", ")

def search(self, el):

ptr = self.head

lvl = 0

while ptr:

if el == ptr.data:

print(f"{el} found in {lvl}th level!")

return ptr

elif el > ptr.data:

ptr = ptr.right

lvl += 1

else:

ptr = ptr.left

lvl += 1

print(f"{el} not found!")

return None

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

ob = Tree()

while True:

user\_input = input("Enter a number for DFS (or 'done' to stop):")

if user\_input.lower() == "done":

break

try:

number = int(user\_input)

ob.insert(number)

except ValueError:

print("Please enter a valid number.")

print("preorder: ", end="")

ob.preorder(ob.head)

print()

print("inorder: ", end="")

ob.inorder(ob.head)

print()

print("postorder: ", end="")

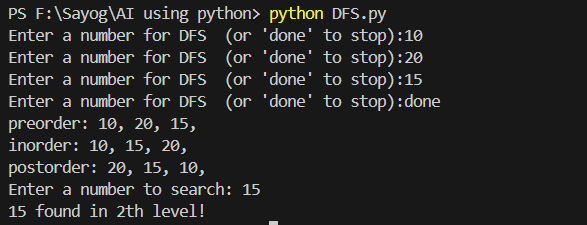
ob.postorder(ob.head)

print()

search\_value = int(input("Enter a number to search: "))

ob.search(search\_value)

## **Output**



# **Write a program to implement Tower of Hanoi Problem**

## **Source Code**

def move(n, from\_rod, to\_rod, aux\_rod):

if n == 1:

print(f"Move top disk from {from\_rod} to {to\_rod}")

return

move(n - 1, from\_rod, aux\_rod, to\_rod)

move(1, from\_rod, to\_rod, aux\_rod)

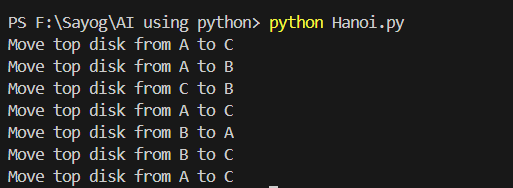
move(n - 1, aux\_rod, to\_rod, from\_rod)

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

n = 3

move(n,"A","C","B")

## **Output**



# **Write a program to demonstrate Bidirectional Search**

## **Source Code**

from collections import deque

class Node:

def \_\_init\_\_(self, state, parent=None):

self.state = state

self.parent = parent

def get\_path(self):

path = []

current = self

while current:

path.append(current.state)

current = current.parent

return path[::-1]

def bidirectional\_search(start\_state, goal\_state):

if start\_state == goal\_state:

return Node(start\_state)

start\_frontier = deque([Node(start\_state)])

goal\_frontier = deque([Node(goal\_state)])

explored\_from\_start = {start\_state: True}

explored\_from\_goal = {goal\_state: True}

while start\_frontier and goal\_frontier:

if start\_frontier:

start\_node = start\_frontier.popleft()

if start\_node.state in explored\_from\_goal:

return join\_paths(start\_node, start\_node.state, explored\_from\_goal)

for neighbor in expand(start\_node):

if neighbor.state not in explored\_from\_start:

explored\_from\_start[neighbor.state] = True

start\_frontier.append(neighbor)

if goal\_frontier:

goal\_node = goal\_frontier.popleft()

if goal\_node.state in explored\_from\_start:

return join\_paths(goal\_node, goal\_node.state, explored\_from\_start)

for neighbor in expand(goal\_node):

if neighbor.state not in explored\_from\_goal:

explored\_from\_goal[neighbor.state] = True

goal\_frontier.append(neighbor)

return None

def join\_paths(node, meeting\_point, explored):

path\_from\_start = node.get\_path()

path\_from\_goal = []

current = meeting\_point

while current in explored:

path\_from\_goal.append(current)

current = explored[current]

return path\_from\_start + path\_from\_goal[::-1][1:]

def expand(node):

children = []

for i in range(1, 4):

child\_state = f"{node.state}-{i}"

child\_node = Node(child\_state, parent=node)

children.append(child\_node)

return children

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

start = "A"

goal = "A-3"

result\_node = bidirectional\_search(start, goal)

if result\_node:

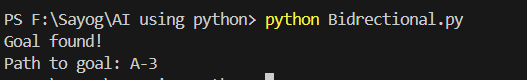
print("Goal found!")

print("Path to goal:", " -> ".join(result\_node))

else:

print("Goal not found.")

## **Output**



# **Write a program to demonstrate Greedy Best Search**

## **Source code**

from queue import PriorityQueue

class Node:

def \_\_init\_\_(self, state, heuristic, parent=None):

self.state = state

self.heuristic = heuristic

self.parent = parent

def \_\_lt\_\_(self, other):

return self.heuristic < other.heuristic

def get\_path(self):

path = []

current = self

while current:

path.append(current.state)

current = current.parent

return path[::-1]

def greedy\_best\_first\_search(start\_state, goal\_state, heuristic):

start\_node = Node(start\_state, heuristic[start\_state])

frontier = PriorityQueue()

frontier.put(start\_node)

explored = set()

while not frontier.empty():

current\_node = frontier.get()

if current\_node.state == goal\_state:

return current\_node.get\_path()

explored.add(current\_node.state)

for neighbor in expand(current\_node.state):

if neighbor not in explored:

neighbor\_node = Node(neighbor, heuristic.get(neighbor, float('inf')), current\_node)

frontier.put(neighbor\_node)

return None

def expand(state):

return [f"{state}-{i}" for i in range(1, 4)]

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

heuristic = {

"A": 6,

"A-1": 5,

"A-2": 4,

"A-3": 3,

"A-1-1": 4,

"A-1-2": 2,

"A-2-1": 2,

"A-2-2": 1,

"A-3-1": 2,

"A-3-2": 0,

"A-3": 0

}

start = "A"

goal = "A-3-2"

result\_path = greedy\_best\_first\_search(start, goal, heuristic)

if result\_path:

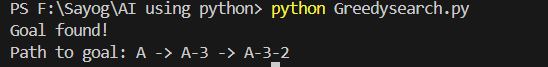
print("Goal found!")

print("Path to goal:", " -> ".join(result\_path))

else:

print("Goal not found.")

## **Output**



# **Write a program to demonstrate A\* search algorithm**

## **Source code**

from queue import PriorityQueue

class Node:

def \_\_init\_\_(self, state, g, h, parent=None):

self.state = state

self.g = g

self.h = h

self.f = g + h

self.parent = parent

def \_\_lt\_\_(self, other):

return self.f < other.f

def get\_path(self):

path = []

current = self

while current:

path.append(current.state)

current = current.parent

return path[::-1]

def a\_star\_search(start\_state, goal\_state, heuristic):

start\_node = Node(start\_state, 0, heuristic[start\_state])

frontier = PriorityQueue()

frontier.put(start\_node)

explored = set()

while not frontier.empty():

current\_node = frontier.get()

if current\_node.state == goal\_state:

return current\_node.get\_path()

explored.add(current\_node.state)

for neighbor in expand(current\_node.state):

g = current\_node.g + 1

h = heuristic.get(neighbor, float('inf'))

neighbor\_node = Node(neighbor, g, h, current\_node)

if neighbor not in explored:

frontier.put(neighbor\_node)

return None

def expand(state):

return [f"{state}-{i}" for i in range(1, 4)]

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

heuristic = {

"A": 6,

"A-1": 5,

"A-2": 4,

"A-3": 3,

"A-1-1": 4,

"A-1-2": 2,

"A-2-1": 2,

"A-2-2": 1,

"A-3-1": 2,

"A-3-2": 0,

"A-3": 0

}

start = "A"

goal = "A-3-2"

result\_path = a\_star\_search(start, goal, heuristic)

if result\_path:

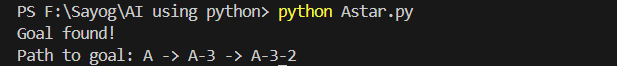
print("Goal found!")

print("Path to goal:", " -> ".join(result\_path))

else:

print("Goal not found.")

## **Output**



# **Write a program to demonstrate Hill Climbing Algorithm**

## **Source code**

import random

class HillClimbing:

def \_\_init\_\_(self, objective\_function):

self.objective\_function = objective\_function

def get\_neighbors(self, current\_state):

return [current\_state + step for step in [-1, 1]]

def search(self, initial\_state):

current\_state = initial\_state

current\_value = self.objective\_function(current\_state)

while True:

neighbors = self.get\_neighbors(current\_state)

next\_state = None

next\_value = current\_value

for neighbor in neighbors:

neighbor\_value = self.objective\_function(neighbor)

if neighbor\_value > next\_value:

next\_value = neighbor\_value

next\_state = neighbor

if next\_state is None:

break

current\_state = next\_state

current\_value = next\_value

return current\_state, current\_value

def objective\_function(x):

return -1 \* (x\*\*2) + 10 \* x # Example function: A downward-facing parabola

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

hc = HillClimbing(objective\_function)

initial\_state = random.randint(0, 10)

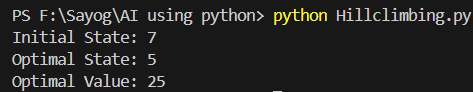
result = hc.search(initial\_state)

print(f"Initial State: {initial\_state}")

print(f"Optimal State: {result[0]}")

print(f"Optimal Value: {result[1]}")

## **Output**



# **Write a program to demonstrate min max algorithm**

## **Source code**

import random

class TicTacToe:

def \_\_init\_\_(self):

self.board = [' ' for \_ in range(9)]

self.current\_player = 'X'

def print\_board(self):

for i in range(3):

print('|'.join(self.board[i\*3:(i+1)\*3]))

if i < 2:

print('-----')

def is\_winner(self, player):

win\_conditions = [

(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)

]

return any(all(self.board[i] == player for i in condition) for condition in win\_conditions)

def is\_draw(self):

return ' ' not in self.board

def minimax(self, depth, is\_maximizing):

if self.is\_winner('X'):

return 1

elif self.is\_winner('O'):

return -1

elif self.is\_draw():

return 0

if is\_maximizing:

best\_score = float('-inf')

for i in range(9):

if self.board[i] == ' ':

self.board[i] = 'X'

score = self.minimax(depth + 1, False)

self.board[i] = ' '

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = float('inf')

for i in range(9):

if self.board[i] == ' ':

self.board[i] = 'O'

score = self.minimax(depth + 1, True)

self.board[i] = ' '

best\_score = min(score, best\_score)

return best\_score

def best\_move(self):

best\_score = float('-inf')

move = -1

for i in range(9):

if self.board[i] == ' ':

self.board[i] = 'X'

score = self.minimax(0, False)

self.board[i] = ' '

if score > best\_score:

best\_score = score

move = i

return move

def player\_move(self):

while True:

try:

move = int(input("Enter your move (1-9): ")) - 1

if self.board[move] == ' ':

self.board[move] = 'O'

break

else:

print("Invalid move! Try again.")

except (ValueError, IndexError):

print("Invalid input! Please enter a number between 1 and 9.")

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

game = TicTacToe()

while True:

game.print\_board()

if game.is\_draw():

print("It's a draw!")

break

if game.is\_winner('O'):

print("O wins!")

break

game.player\_move()

if game.is\_draw():

game.print\_board()

print("It's a draw!")

break

if game.is\_winner('O'):

game.print\_board()

print("O wins!")

break

print("AI's turn (X):")

move = game.best\_move()

if move != -1:

game.board[move] = 'X'

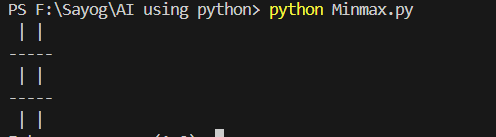
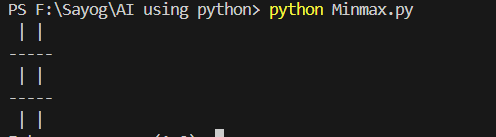
if game.is\_winner('X'):

game.print\_board()

print("X wins!")

break

## **Output**

****

# **Write a program to demonstrate alphabeta**

## **Source code**

class TicTacToe:

def \_\_init\_\_(self):

self.board = [' ' for \_ in range(9)]

self.current\_player = 'X'

def print\_board(self):

for i in range(3):

print('|'.join(self.board[i\*3:(i+1)\*3]))

if i < 2:

print('-----')

def is\_winner(self, player):

win\_conditions = [

(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)

]

return any(all(self.board[i] == player for i in condition) for condition in win\_conditions)

def is\_draw(self):

return ' ' not in self.board

def minimax(self, depth, alpha, beta, is\_maximizing):

if self.is\_winner('X'):

return 1

elif self.is\_winner('O'):

return -1

elif self.is\_draw():

return 0

if is\_maximizing:

best\_score = float('-inf')

for i in range(9):

if self.board[i] == ' ':

self.board[i] = 'X'

score = self.minimax(depth + 1, alpha, beta, False)

self.board[i] = ' '

best\_score = max(score, best\_score)

alpha = max(alpha, best\_score)

if beta <= alpha:

break

return best\_score

else:

best\_score = float('inf')

for i in range(9):

if self.board[i] == ' ':

self.board[i] = 'O'

score = self.minimax(depth + 1, alpha, beta, True)

self.board[i] = ' '

best\_score = min(score, best\_score)

beta = min(beta, best\_score)

if beta <= alpha:

break

return best\_score

def best\_move(self):

best\_score = float('-inf')

move = -1

alpha = float('-inf')

beta = float('inf')

for i in range(9):

if self.board[i] == ' ':

self.board[i] = 'X'

score = self.minimax(0, alpha, beta, False)

self.board[i] = ' '

if score > best\_score:

best\_score = score

move = i

return move

def player\_move(self):

while True:

try:

move = int(input("Enter your move (1-9): ")) - 1

if self.board[move] == ' ':

self.board[move] = 'O'

break

else:

print("Invalid move! Try again.")

except (ValueError, IndexError):

print("Invalid input! Please enter a number between 1 and 9.")

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

game = TicTacToe()

while True:

game.print\_board()

if game.is\_draw():

print("It's a draw!")

break

if game.is\_winner('O'):

print("O wins!")

break

game.player\_move()

if game.is\_draw():

game.print\_board()

print("It's a draw!")

break

if game.is\_winner('O'):

game.print\_board()

print("O wins!")

break

print("AI's turn (X):")

move = game.best\_move()

if move != -1:

game.board[move] = 'X'

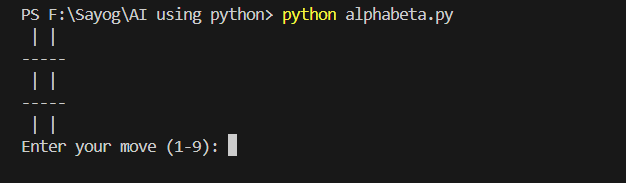
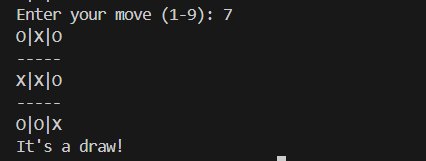
if game.is\_winner('X'):

game.print\_board()

print("X wins!")

break

## **Output**



# **Write a program to demonstrate bidirectional search**

## **Source code**

from collections import deque

class Node:

def \_\_init\_\_(self, state, parent=None):

self.state = state

self.parent = parent

def get\_path(self):

path = []

current = self

while current:

path.append(current.state)

current = current.parent

return path[::-1]

def bidirectional\_search(start\_state, goal\_state):

if start\_state == goal\_state:

return Node(start\_state)

start\_frontier = deque([Node(start\_state)])

goal\_frontier = deque([Node(goal\_state)])

explored\_from\_start = {start\_state: True}

explored\_from\_goal = {goal\_state: True}

while start\_frontier and goal\_frontier:

if start\_frontier:

start\_node = start\_frontier.popleft()

if start\_node.state in explored\_from\_goal:

return join\_paths(start\_node, start\_node.state, explored\_from\_goal)

for neighbor in expand(start\_node):

if neighbor.state not in explored\_from\_start:

explored\_from\_start[neighbor.state] = True

start\_frontier.append(neighbor)

if goal\_frontier:

goal\_node = goal\_frontier.popleft()

if goal\_node.state in explored\_from\_start:

return join\_paths(goal\_node, goal\_node.state, explored\_from\_start)

for neighbor in expand(goal\_node):

if neighbor.state not in explored\_from\_goal:

explored\_from\_goal[neighbor.state] = True

goal\_frontier.append(neighbor)

return None

def join\_paths(node, meeting\_point, explored):

path\_from\_start = node.get\_path()

path\_from\_goal = []

current = meeting\_point

while current in explored:

path\_from\_goal.append(current)

current = explored[current]

return path\_from\_start + path\_from\_goal[::-1][1:]

def expand(node):

children = []

for i in range(1, 4):

child\_state = f"{node.state}-{i}"

child\_node = Node(child\_state, parent=node)

children.append(child\_node)

return children

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

start = "A"

goal = "A-3"

result\_node = bidirectional\_search(start, goal)

if result\_node:

print("Goal found!")

print("Path to goal:", " -> ".join(result\_node))

else:

print("Goal not found.")

## **Output**

