Prg 1: Water jug

```
def dfs(start, goal, capacity):
    stack = [start]
    explored = set()
    while stack:
        state = stack.pop()
        if state in explored:
            continue
        explored.add(state)
        print(f"Exploring: {state}")
        if state == goal:
            print(f"Goal reached: {state}")
            return True
        j1, j2 = state
        c1, c2 = capacity
        stack.append((c1, j2))
        stack.append((j1, c2))
        stack.append((0, j2))
        stack.append((j1, 0))
        transfer = min(j1, c2 - j2)
        stack.append((j1 - transfer, j2 +
transfer))
        transfer = min(j2, c1 - j1)
        stack.append((j1 + transfer, j2 -
transfer))
    print("No Output found.")
    return False
start_state = (0, 0)
goal_state = (2, 0)
jug_capacities = (4, 3)
dfs(start_state, goal_state, jug_capacities)
```

Output:

```
Exploring: (0, 0)
Exploring: (0, 3)
Exploring: (3, 0)
Exploring: (3, 3)
Exploring: (4, 2)
Exploring: (4, 0)
Exploring: (1, 3)
Exploring: (1, 0)
Exploring: (0, 1)
Exploring: (4, 1)
Exploring: (2, 3)
Exploring: (2, 0)
Goal reached: (2, 0)
True
```

Prg 2: Missionaries-Cannibals

```
from heapq import heappop, heappush
def is_valid(state):
    missionaries_left, cannibals_left, boat =
state
    missionaries_right = 3 -
missionaries_left
    cannibals_right = 3 - cannibals_left
    if missionaries_left >= 0 and
cannibals_left >= 0 \
        and missionaries_right >= 0 and
cannibals_right >= 0 \
        and (missionaries_left == 0 or
missionaries_left >= cannibals_left) \
        and (missionaries_right == 0 or
missionaries_right >= cannibals_right):
        return True
    return False
def get_neighbors(state):
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1,
1)]
    missionaries_left, cannibals_left, boat =
state
    neighbors = []
    for m, c in moves:
        if boat == 1:
            new_state = (missionaries_left -
m,
                         cannibals_left - c, 0)
        else:
            new_state = (missionaries_left +
m,
                         cannibals_left + c, 1)
        if is_valid(new_state):
            neighbors.append(new_state)
    return neighbors
def best_first_search(start, goal):
    frontier = []
    heappush(frontier, (0, start))
    explored = set()
    while frontier:
        current_state = heappop(frontier)
        if current_state in explored:
            continue
        if current_state == goal:
            return True
        explored.add(current_state)
        for neighbor in
get_neighbors(current_state):
            if neighbor not in explored:
                heappush(frontier,
                         (neighbor[0] + neighbor[1],
neighbor))
    return False
start_state = (3, 3, 1)
goal_state = (0, 0, 0)
result = best_first_search(start_state,
goal_state)
print("Output found" if result else "No
Output.")
```

Output: Output found

Prg 3: A* Search

```
import heapq
class Node:
    def __init__(self, name, g=0, h=0):
        self.name = name
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = None
    def __lt__(self, other):
        return self.f < other.f
def a_star_search(start, goal, graph,
heuristic):
```

```
    open_list = []
    closed_list = set()
    start_node = Node(start, g=0,
h=heuristic[start])
    goal_node = Node(goal, g=0, h=0)
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node =
heapq.heappop(open_list)
        if current_node.name in closed_list:
            continue
        if current_node.name ==
goal_node.name:
            path = []
            while current_node:

path.append(current_node.name)
                current_node =
current_node.parent
            return path[::-1]
        closed_list.add(current_node.name)
        for neighbor, cost in
graph[current_node.name]:
            if neighbor in closed_list:
                continue
            g = current_node.g + cost
            h = heuristic[neighbor]
            neighbor_node = Node(neighbor,
g, h)
            neighbor_node.parent =
current_node
            heapq.heappush(open_list,
neighbor_node)
    return None
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('D', 2), ('E', 5)],
    'C': [('A', 4), ('F', 3)],
    'D': [('B', 2), ('G', 1)],
    'E': [('B', 5), ('G', 2)],
    'F': [('C', 3), ('G', 2)],
    'G': [('D', 1), ('E', 2), ('F', 2)]
}
heuristic = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 1,
    'E': 4,
    'F': 3,
    'G': 0
}
start = 'A'
goal = 'G'
path = a_star_search(start, goal, graph,
heuristic)
print("Path found:", path)
```

Output: Path found: ['A', 'B', 'D', 'G']

Prg 4: AO* Search

```
class Node:
    def __init__(self, name,
is_and_node=False):
        self.name = name
        self.is_and_node = is_and_node
        self.children = []
        self.heuristic = float('inf')
        self.best_child = None
        self.solved = False
    def add_children(self,
children_with_costs):

self.children.append(children_with_costs)
def ao_star_search(node):
    if node.solved:
        return node.heuristic
    if not node.children:
        node.heuristic = 0
        node.solved = True
        return node.heuristic
    min_cost = float('inf')
    best_child_set = None
    for child_set in node.children:
        cost = 0
        for child, child_cost in child_set:
            cost += ao_star_search(child) +
child_cost
        if cost < min_cost:
            min_cost = cost
            best_child_set = child_set
    node.heuristic = min_cost
    node.best_child = best_child_set
    node.solved = all(child.solved for child,
_ in best_child_set)
    return node.heuristic
def print_Output(node, indent=0):
    if node.best_child:
        if node.is_and_node:
            print(" " * indent + f"{node.name}
(AND node)")
        else:
            print(" " * indent + f"{node.name}
(OR node)")
        for child, _ in node.best_child:
            print_Output(child, indent + 1)
    else:
        print(" " * indent + node.name)
A = Node('A', is_and_node=False)
B = Node('B', is_and_node=True)
C = Node('C', is_and_node=False)
D = Node('D', is_and_node=False)
E = Node('E', is_and_node=False)
F = Node('F', is_and_node=False)
A.add_children([(B, 1), (C, 2)])
B.add_children([(D, 1), (E, 2)])
C.add_children([(F, 3)])
D.add_children([])
E.add_children([])
F.add_children([])
ao_star_search(A)
print("Output:")
print_Output(A)
```

Output: A (OR node)
B (AND node)
D
E
C (OR node)
F

Prg 5: 8 Queen Problem

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i][col] == 1:
            return False
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, -1, -1),
                    range(col, len(board))):
        if board[i][j] == 1:
            return False
    return True
def solve_n_queens_util(board, row):
    if row >= len(board):
        return True
    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row][col] = 1
            if solve_n_queens_util(board, row
+ 1):
                return True
            board[row][col] = 0
    return False
def solve_n_queens(n):
    board = [[0] * n for _ in range(n)]
    if not solve_n_queens_util(board, 0):
        return None
    return board
def print_board(board):
    if board is None:
        print("No Output exists.")
        return
    for row in board:
        print(" ".join("Q" if col == 1 else "." for
col in row))
    print()
n = 8
Output = solve_n_queens(n)
print_board(Output)
```

Output:

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

Prg 6:TSP using heuristic

```
import numpy as np
def nearest_neighbor_tsp(graph, start):
    n = len(graph)
    visited = [False] * n
    path = [start]
    visited[start] = True
    current = start
    total_cost = 0
    for _ in range(n - 1):
        next_city = None
        min_cost = float('inf')
        for city in range(n):
            if not visited[city] and
graph[current][city] < min_cost:
                next_city = city
                min_cost = graph[current][city]
        path.append(next_city)
        visited[next_city] = True
        total_cost += min_cost
        current = next_city
    total_cost += graph[current][start]
    path.append(start)
    return path, total_cost
graph = [
    [0, 29, 20, 21],
    [29, 0, 15, 17],
    [20, 15, 0, 28],
    [21, 17, 28, 0]
]
start_city = 0
path, cost = nearest_neighbor_tsp(graph,
start_city)
print("Path:", path)
print("Total cost:", cost)
```

Output: Path: [0, 2, 1, 3, 0]
Total cost: 73

Prg 7: FWD & BWD Chaining

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
    def add_fact(self, fact):
        self.facts.add(fact)
    def add_rule(self, rule):
        self.rules.append(rule)
    def is_fact(self, fact):
        return fact in self.facts
    def backward_chain(self, goal):
        if self.is_fact(goal):
            return True
        for rule in self.rules:
            if rule.conclusion == goal:
                if all(self.backward_chain(fact)
                       for fact in rule.condition):
                    self.facts.add(goal)
                    return True
        return False
class Rule:
    def __init__(self, condition, conclusion):
        self.condition = condition
        self.conclusion = conclusion
kb = KnowledgeBase()
kb.add_fact('fever')
kb.add_fact('cough')
```

```
rule1 = Rule({'fever', 'cough'}, 'flu')
rule2 = Rule({'headache', 'flu'},
'consult_doctor')
kb.add_rule(rule1)
kb.add_rule(rule2)
goal = 'consult_doctor'
result = kb.backward_chain(goal)
print("Goal:", goal)
print("Achieved:", result)
print("Known facts after inference:",
kb.facts)
```

Output: Goal: consult_doctor
Achieved: False
Known facts after inference: {'fever',
'cough'}

Prg 8: reOutput principle FOPL

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
    def add_fact(self, fact):
        self.facts.add(fact)
        self.infer()
    def add_rule(self, rule):
        self.rules.append(rule)
        self.infer()
    def infer(self):
        new_facts = True
        while new_facts:
            new_facts = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    new_facts = True
class Rule:
    def __init__(self, condition, conclusion):
        self.condition = condition
        self.conclusion = conclusion
    def apply(self, facts):
        if all(fact in facts for fact in
self.condition) \
           and self.conclusion not in facts:
            facts.add(self.conclusion)
            return True
        return False
kb = KnowledgeBase()
kb.add_fact('fever')
kb.add_fact('cough')
rule1 = Rule({'fever', 'cough'}, 'flu')
rule2 = Rule({'flu', 'headache'},
'consult_doctor')
kb.add_rule(rule1)
kb.add_rule(rule2)
print("Initial facts:", kb.facts)
kb.add_fact('headache')
print("Facts after adding 'headache':",
kb.facts)
```

Output: Initial facts: {'fever', 'flu', 'cough'}
Facts after adding 'headache': {'flu',
'cough', 'consult_doctor', 'fever',
'headache'}

Prg 9: Tic Tac Toe game

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)
def check_win(board, player):
    win_conditions = [
        [board[0][0], board[0][1], board[0][2]],
        [board[1][0], board[1][1], board[1][2]],
        [board[2][0], board[2][1], board[2][2]],
        [board[0][0], board[1][0], board[2][0]],
        [board[0][1], board[1][1], board[2][1]],
        [board[0][2], board[1][2], board[2][2]],
        [board[0][0], board[1][1], board[2][2]],
        [board[2][0], board[1][1], board[0][2]]
    ]
    return [player, player, player] in
win_conditions
def check_draw(board):
    return all(cell != ' ' for row in board for
cell in row)
def get_move():
    while True:
        try:
            row, col = map(
                int,
                input("Enter your move (row
and column): ").split()
            )
            if row in range(1, 4) and col in
range(1, 4):
                return row - 1, col - 1
            else:
                print("Invalid move. Please
enter row and column between 1 and 3.")
        except ValueError:
            print("Invalid input. Please enter
two numbers separated by space.")
def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in
range(3)]
    current_player = 'X'
    while True:
        print_board(board)
        print(f"Player {current_player}'s turn")
        row, col = get_move()
        if board[row][col] == ' ':
            board[row][col] = current_player
            if check_win(board,
current_player):
                print_board(board)
                print(f"Player {current_player}
wins!")
                break
            if check_draw(board):
                print_board(board)
                print("It's a draw!")
                break
```

```
            current_player = 'O' if
current_player == 'X' else 'X'
        else:
            print("Cell already taken. Choose
another.")
if __name__ == "__main__":
    tic_tac_toe()
```

Output:

```
-----
| |
-----
| |
-----
Player X's turn
Enter your move (row and column): 1 1
X | |
-----
| |
-----
| |
-----
Player O's turn
Enter your move (row and column): 2 2…
```

Prg 10:Info text in search box

```
!pip install wikipedia-api
import wikipediaapi
class WikiBot:
    def __init__(self):
        user_agent = "WikiBot/1.0
(https://example.com/wiki-bot;
contact@example.com)"
        self.wiki = wikipediaapi.Wikipedia(
            'en',
            headers={'User-Agent':
user_agent}
        )
    def get_summary(self, query):
        page = self.wiki.page(query)
        if page.exists():
            return page.summary
        else:
            return "Sorry, no information found
for your query."
def main():
    bot = WikiBot()
    while True:
        query = input(
            "Enter your search query (or type
'exit' to quit): "
        )
        if query.lower() == 'exit':
            break
        summary = bot.get_summary(query)
        print(f"\nSummary for
'{query}':\n{summary}\n")
if __name__ == "__main__":
    main()
```

Output: Enter your search query (or type 'exit' to quit): AI
Summary for 'AI':
Artificial intelligence (AI)….and other fields.
Enter your search query (or type 'exit' to quit): exit

Prg 11: Number Guessing Game

```
import random
def guess_the_number():
    print("Welcome to the Number
Guessing Game!")
    print("I'm thinking of a number between
1 and 100.")
    number_to_guess = random.randint(1,
100)
    attempts = 0
    while True:
        try:
            guess = int(input("Make a guess:
"))
            attempts += 1
            if guess < number_to_guess:
                print("Too low. Try again.")
            elif guess > number_to_guess:
                print("Too high. Try again.")
            else:
                print(f"Congratulations! You've
guessed the number {number_to_guess}
in {attempts} attempts.")
                break
        except ValueError:
            print("Invalid input. Please enter a
number between 1 and 100.")
    print("Thanks for playing the Number
Guessing Game!")
guess_the_number()
```

Output: Welcome to the Number Guessing Game!
I'm thinking of a number between 1 and 100.
Make a guess: 9
Too low. Try again.
Make a guess: 15
Too low. Try again.
Make a guess: 200
Too high. Try again.
Make a guess: 100
Too high. Try again.
Make a guess: 50
Congratulations! You've guessed the number 50