1: Program to implement Breadth First Search Algorithm

```python
from collections import deque


class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, start):
        visited = set()
        queue = deque([start])
        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                queue.extend(
                    neighbor
                    for neighbor in self.graph[vertex]
                    if neighbor not in visited
                )
        return visited


g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.add_edge("B", "E")
g.add_edge("C", "F")
g.add_edge("E", "F")
print(g.bfs("A"))
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 1.py
{'C', 'B', 'E', 'F', 'D', 'A'}
```

2: Program to implement Depth First Search Algorithm

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, start, visited=None):
        if visited is None:
            visited = set()
        visited.add(start)
        for neighbor in self.graph[start]:
            if neighbor not in visited:
                self.dfs(neighbor, visited)
        return visited


g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.add_edge("B", "E")
g.add_edge("C", "F")
g.add_edge("E", "F")
print(g.dfs("A"))
```

Output:

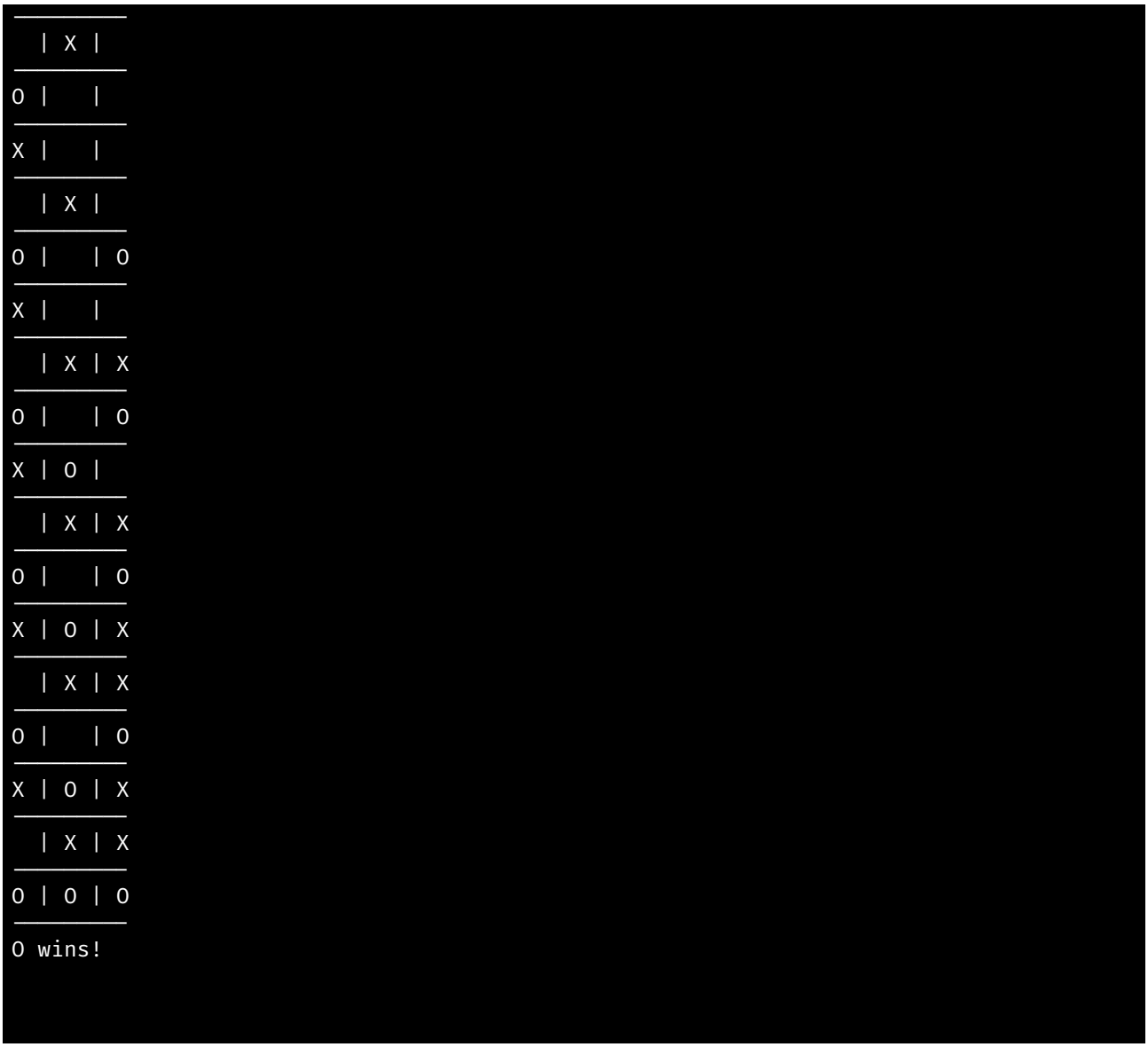
```
rudy@rudy: ~/Documents/college/ai/lab$ python3 2.py
{'D', 'A', 'F', 'E', 'C', 'B'}
```

3: Program to implement Tic-Tac-Toe game using Python.

```python
import random


def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board):
    for row in board:
        if row.count(row[0]) == 3 and row[0] != " ":
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] !=
" ":
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
        return board[0][2]
    return None


def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    for turn in range(9):
        player = players[turn % 2]
        row, col = random.randint(0, 2), random.randint(0, 2)
        while board[row][col] != " ":
            row, col = random.randint(0, 2), random.randint(0, 2)
        board[row][col] = player
        print_board(board)
        winner = check_winner(board)
        if winner:
            print(f"{winner} wins!")
            return
    print("It's a draw!")


tic_tac_toe()
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 3.py
  |   |
---------
  | X |
---------
  |   |
---------
  |   |
---------
  | X |
---------
O |   |
---------
X |   |
```


  | X |
O | |
X | |
  | X |
O | | O
X | |
  | X | X
O | | O
X | O |
  | X | X
O | | O
X | O | X
  | X | X
O | | O
X | O | X
  | X | X
O | O | O
O wins!

4: Program to implement Water-Jug problem using Python.

```python
def water_jug(a, b, target):
    def gcd(x, y):
        while y:
            x, y = y, x % y
        return x

    if target > max(a, b) or target % gcd(a, b) != 0:
        print("Not possible")
        return

    x, y = 0, 0
    steps = []

    while x != target and y != target:
        if x == 0:
            x = a
            steps.append(f"Fill Jug A: ({x}, {y})")
        elif y == b:
            y = 0
            steps.append(f"Empty Jug B: ({x}, {y})")
        else:
            transfer = min(x, b - y)
            x -= transfer
            y += transfer
            steps.append(f"Transfer from A to B: ({x}, {y})")

    for step in steps:
        print(step)


water_jug(3, 5, 4)
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 4.py
Fill Jug A: (3, 0)
Transfer from A to B: (0, 3)
Fill Jug A: (3, 3)
Transfer from A to B: (1, 5)
Empty Jug B: (1, 0)
Transfer from A to B: (0, 1)
Fill Jug A: (3, 1)
Transfer from A to B: (0, 4)
```

5:

Program to implement Travelling Salesman Problem using Python.

```
import itertools


class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, cost):
        if u not in self.graph:
            self.graph[u] = {}
        if v not in self.graph:
            self.graph[v] = {}
        self.graph[u][v] = cost
        self.graph[v][u] = cost

    def tsp(self):
        if not self.graph:
            return [], 0

        vertices = list(self.graph.keys())
        min_path = float("inf")
        min_route = []

        for perm in itertools.permutations(vertices[1:]):
            current_path = [vertices[0]] + list(perm) + [vertices[0]]
            current_cost = sum(
                self.graph[current_path[i]].get(current_path[i + 1],
float("inf"))
                for i in range(len(current_path) - 1)
            )
            if current_cost < min_path:
                min_path = current_cost
                min_route = current_path

        return min_route, min_path


g = Graph()
g.add_edge("A", "B", 10)
g.add_edge("A", "C", 15)
g.add_edge("B", "C", 35)
g.add_edge("B", "D", 25)
g.add_edge("C", "D", 30)
print(g.tsp())
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 5.py
(['A', 'B', 'D', 'C', 'A'], 80)
```

6: Program to implement Tower of Hanoi using Python.

```python
class TowerOfHanoi:
    def __init__(self, n):
        self.n = n

    def solve(self, n=None, source="A", target="C", auxiliary="B"):
        if n is None:
            n = self.n
        if n == 1:
            print(f"Move disk 1 from {source} to {target}")
            return
        self.solve(n - 1, source, auxiliary, target)
        print(f"Move disk {n} from {source} to {target}")
        self.solve(n - 1, auxiliary, target, source)


hanoi = TowerOfHanoi(3)
hanoi.solve()
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 6.py
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

7: Program to implement Monkey Banana problem using Python.

```
import unittest
import logging

logging.basicConfig(level=logging.INFO, format="%(levelname)s - %(message)s")


class MonkeyBananaProblem:
    def __init__(self):
        self.monkey_position = "floor"
        self.banana_position = "high"
        self.stool_position = "floor"
        self.actions = []

    def perform_action(self, action):
        if action == "move_to_stool":
            return self.move_to_stool()
        elif action == "climb":
            return self.climb()
        elif action == "grab_banana":
            return self.grab_banana()
        elif action == "descend":
            return self.descend()
        elif action == "move_to_floor":
            return self.move_to_floor()
        else:
            return False

    def move_to_stool(self):
        if self.monkey_position == "floor" and self.stool_position == "floor":
            self.monkey_position = "stool"
            self.actions.append("move_to_stool")
            return True
        return False

    def climb(self):
        if self.monkey_position == "stool" and self.banana_position == "high":
            self.monkey_position = "high"
            self.actions.append("climb")
            return True
        return False

    def grab_banana(self):
        if self.monkey_position == "high" and self.banana_position == "high":
            self.banana_position = "held"
            self.actions.append("grab_banana")
            return True
        return False

    def descend(self):
        if self.monkey_position == "high":
            self.monkey_position = "stool"
            self.actions.append("descend")
            return True
        return False

    def move_to_floor(self):
        if self.monkey_position == "stool":
            self.monkey_position = "floor"
            self.actions.append("move_to_floor")
            return True
        return False
```

```
    def solve(self):
        actions = ["move_to_stool", "climb", "grab_banana", "descend",
"move_to_floor"]
        for action in actions:
            if not self.perform_action(action):
                return "Failed to perform action: " + action
        return "Banana acquired!"


class TestMonkeyBananaProblem(unittest.TestCase):
    def setUp(self):
        self.problem = MonkeyBananaProblem()

    def test_initial_state(self):
        self.assertEqual(self.problem.monkey_position, "floor")
        self.assertEqual(self.problem.banana_position, "high")
        self.assertEqual(self.problem.stool_position, "floor")
        self.assertEqual(self.problem.actions, [])
        logging.info("Initial state test passed.")

    def test_actions(self):
        self.assertTrue(self.problem.perform_action("move_to_stool"))
        self.assertEqual(self.problem.monkey_position, "stool")
        self.assertTrue(self.problem.perform_action("climb"))
        self.assertEqual(self.problem.monkey_position, "high")
        self.assertTrue(self.problem.perform_action("grab_banana"))
        self.assertEqual(self.problem.banana_position, "held")
        self.assertTrue(self.problem.perform_action("descend"))
        self.assertEqual(self.problem.monkey_position, "stool")
        self.assertTrue(self.problem.perform_action("move_to_floor"))
        self.assertEqual(self.problem.monkey_position, "floor")
        logging.info("Actions test passed.")

    def test_solve(self):
        result = self.problem.solve()
        self.assertEqual(result, "Banana acquired!")
        self.assertEqual(
            self.problem.actions,
            ["move_to_stool", "climb", "grab_banana", "descend",
"move_to_floor"],
        )
        logging.info("Solve test passed.")

    def test_invalid_action(self):
        result = self.problem.perform_action("invalid_action")
        self.assertFalse(result)
        logging.info("Invalid action test passed.")

    def test_failed_action(self):
        self.problem.monkey_position = "floor"
        result = self.problem.perform_action("climb")
        self.assertFalse(result)
        logging.info("Failed action test result: %s", result)

    def tearDown(self):
        logging.info("Actions taken: %s", self.problem.actions)


if __name__ == "__main__":
    unittest.main()
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 7.py
INFO - Actions test passed.
INFO - Actions taken: ['move_to_stool', 'climb', 'grab_banana', 'descend',
'move_to_floor']
.INFO - Failed action test result: False
INFO - Actions taken: []
.INFO - Initial state test passed.
INFO - Actions taken: []
.INFO - Invalid action test passed.
INFO - Actions taken: []
.INFO - Solve test passed.
INFO - Actions taken: ['move_to_stool', 'climb', 'grab_banana', 'descend',
'move_to_floor']
.
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

8: Program to implement N-Queens Problem using Python.

```python
class NQueens:
    def __init__(self, n):
        self.N = n
        self.board = [[0 for _ in range(n)] for _ in range(n)]

    def print_solution(self):
        for row in self.board:
            print(" ".join(str(cell) for cell in row))
        print()

    def is_safe(self, row, col):
        for i in range(col):
            if self.board[row][i] == 1:
                return False

        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if self.board[i][j] == 1:
                return False

        for i, j in zip(range(row, self.N, 1), range(col, -1, -1)):
            if self.board[i][j] == 1:
                return False

        return True

    def solve_util(self, col):
        if col >= self.N:
            return True

        for i in range(self.N):
            if self.is_safe(i, col):
                self.board[i][col] = 1

                if self.solve_util(col + 1):
                    return True

                self.board[i][col] = 0

        return False

    def solve(self):
        if not self.solve_util(0):
            print("Solution does not exist")
            return False

        self.print_solution()
        return True


n_queens = NQueens(4)
n_queens.solve()
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 8.py
0 0 1 0
1 0 0 0
0 0 0 1
```

```
0 1 0 0
```

9: Program to implement Naïve Bayes Algorithm using Python.

```python
class NaiveBayes:
    def __init__(self):
        self.data = []
        self.class_probs = {}

    def train(self, data):
        self.data = data
        total_count = len(data)
        for item in data:
            label = item[-1]
            if label not in self.class_probs:
                self.class_probs[label] = 0
            self.class_probs[label] += 1
        for label in self.class_probs:
            self.class_probs[label] /= total_count

    def predict(self, item):
        label = max(self.class_probs, key=self.class_probs.get)
        print(f"Predicted class for {item}: {label}")
        return label


nb = NaiveBayes()
data = [
    ["sunny", "hot", "high", "false", "no"],
    ["sunny", "hot", "high", "true", "no"],
    ["overcast", "hot", "high", "false", "yes"],
    ["rainy", "mild", "high", "false", "yes"],
]
nb.train(data)
nb.predict(["sunny", "cool", "high", "false"])
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 9.py
Predicted class for ['sunny', 'cool', 'high', 'false']: no
```

10:

Program to implement Backpropagation Algorithm using Python.

```
import numpy as np


class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.uniform(-1, 1, (input_size,
hidden_size))
        self.weights_hidden_output = np.random.uniform(
            -1, 1, (hidden_size, output_size)
        )
        self.learning_rate = 0.1

    def feedforward(self, inputs):
        self.hidden = self.sigmoid(np.dot(inputs, self.weights_input_hidden))
        self.output = self.sigmoid(np.dot(self.hidden,
self.weights_hidden_output))
        return self.output

    def backward(self, inputs, target):
        output_error = target - self.output
        output_delta = output_error * self.sigmoid_derivative(self.output)

        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden)

        # Update weights
        self.weights_hidden_output += (
            self.hidden.T.dot(output_delta) * self.learning_rate
        )
        self.weights_input_hidden += inputs.T.dot(hidden_delta) *
self.learning_rate

    def train(self, inputs, target):
        self.feedforward(inputs)
        self.backward(inputs, target)

    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def sigmoid_derivative(x):
        return x * (1 - x)


if __name__ == "__main__":
    nn = NeuralNetwork(3, 3, 1)
    inputs = np.array([[1, 0, 1], [0, 1, 0], [1, 1, 1]])
    targets = np.array([[1], [0], [1]])

    for epoch in range(1000):
        for i in range(len(inputs)):
            nn.train(inputs[i].reshape(1, -1), targets[i])  # Reshape inputs

    print("Final output after training:")
    for input in inputs:
        print(
            f"Input: {input}, Predicted Output: {nn.feedforward(input.reshape(1,
```

```
  -1))}"
        )
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 10.py
Final output after training:
Input: [1 0 1], Predicted Output: [[0.92320461]]
Input: [0 1 0], Predicted Output: [[0.22138659]]
Input: [1 1 1], Predicted Output: [[0.8575018]]
```

11: Program to implement Genetics Algorithm using Python.

```
import random


class GeneticAlgorithm:
    def __init__(self, population_size, gene_length):
        self.population_size = population_size
        self.gene_length = gene_length
        self.population = self.initialize_population()

    def initialize_population(self):
        return [
            [random.randint(0, 1) for _ in range(self.gene_length)]
            for _ in range(self.population_size)
        ]

    def fitness(self, individual):
        return sum(individual)

    def selection(self):
        sorted_population = sorted(self.population, key=self.fitness,
reverse=True)
        return sorted_population[: self.population_size // 2]

    def crossover(self, parent1, parent2):
        point = random.randint(1, self.gene_length - 1)
        return parent1[:point] + parent2[point:]

    def mutate(self, individual):
        for i in range(len(individual)):
            if random.random() < 0.01:
                individual[i] = 1 - individual[i]

    def evolve(self):
        selected = self.selection()
        next_generation = []
        while len(next_generation) < self.population_size:
            parent1, parent2 = random.sample(selected, 2)
            child = self.crossover(parent1, parent2)
            self.mutate(child)
            next_generation.append(child)
        self.population = next_generation


ga = GeneticAlgorithm(10, 5)
for _ in range(10):
    ga.evolve()
print(ga.population)
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 11.py
[[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1],
[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

12: Program to implement A* Search Algorithm.

```
import heapq


class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, u, v, cost):
        if u not in self.edges:
            self.edges[u] = {}
        if v not in self.edges:
            self.edges[v] = {}
        self.edges[u][v] = cost
        self.edges[v][u] = cost


class AStarSearch:
    def __init__(self, graph):
        self.graph = graph

    def heuristic(self, a, b):
        return 1

    def search(self, start, goal):
        queue = []
        heapq.heappush(queue, (0, start))
        came_from = {}
        cost_so_far = {start: 0}

        while queue:
            current = heapq.heappop(queue)[1]
            if current == goal:
                break
            for neighbor, cost in self.graph.edges[current].items():
                new_cost = cost_so_far[current] + cost
                if neighbor not in cost_so_far or new_cost <
cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + self.heuristic(goal, neighbor)
                    heapq.heappush(queue, (priority, neighbor))
                    came_from[neighbor] = current
        return came_from


graph = Graph()
graph.add_edge("A", "B", 1)
graph.add_edge("A", "C", 4)
graph.add_edge("B", "C", 2)
graph.add_edge("B", "D", 5)
graph.add_edge("C", "D", 1)

astar = AStarSearch(graph)
came_from = astar.search("A", "D")
print(came_from)
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 12.py
{'B': 'A', 'C': 'B', 'D': 'C'}
```

[redacted]

13: Program to implement Greedy Search Algorithm.

```
class GreedySearch:
    def __init__(self, graph):
        self.graph = graph

    def search(self, start, goal):
        visited = set()
        queue = [(start, 0)]
        while queue:
            queue.sort(key=lambda x: x[1])
            current, cost = queue.pop(0)
            if current in visited:
                continue
            visited.add(current)
            if current == goal:
                return cost
            for neighbor, edge_cost in self.graph[current].items():
                if neighbor not in visited:
                    queue.append((neighbor, cost + edge_cost))
        return None


graph = {
    "A": {"B": 1, "C": 2},
    "B": {"A": 1, "D": 3},
    "C": {"A": 2, "D": 1},
    "D": {"B": 3, "C": 1},
}
greedy = GreedySearch(graph)
result = greedy.search("A", "D")
print(result)
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 13.py
3
```

14: Program to implement the Uniform Cost Search Algorithm.

```
import heapq


class Node:
    def __init__(self, position, cost=0, parent=None):
        self.position = position
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost


def uniform_cost_search(start, goal, graph):
    open_list = []
    closed_list = set()

    start_node = Node(start, 0)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        current_position = current_node.position

        if current_position in closed_list:
            continue

        closed_list.add(current_position)

        if current_position == goal:
            path = []
            total_cost = current_node.cost
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1], total_cost

        for neighbor, cost in graph.get(current_position, {}).items():
            if neighbor in closed_list:
                continue

            neighbor_node = Node(neighbor, current_node.cost + cost,
current_node)
            heapq.heappush(open_list, neighbor_node)

    return None, float("inf")


graph = {
    "A": {"B": 1, "C": 4},
    "B": {"A": 1, "C": 2, "D": 5},
    "C": {"A": 4, "B": 2, "D": 1},
    "D": {"B": 5, "C": 1},
}

start = "A"
goal = "D"

path, cost = uniform_cost_search(start, goal, graph)
```

```
  print("Path found:", path)
  print("Total cost:", cost)
```

Output:

```
rudy@rudy: ~/Documents/college/ai/lab$ python3 14.py
Path found: ['A', 'B', 'C', 'D']
Total cost: 4
```