Program:

```
from collections import deque
class Graph:
    def __init__(self):  # Corrected constructor
        self.graph = {}

    def add_edge(self, u, v):
        """Adds an edge to the graph (Undirected Graph)"""
        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)  # Remove this for a directed graph

    def bfs(self, start):
        """Performs Breadth-First Search"""
        visited = set()
        queue = deque([start])

        while queue:
            node = queue.popleft()
            if node not in visited:
                print(node, end=" ")
                visited.add(node)
                for neighbor in self.graph.get(node, []):
                    if neighbor not in visited:
                        queue.append(neighbor)

    def dfs(self, start):
        """Performs Depth-First Search"""
        visited = set()
        stack = [start]

        while stack:
            node = stack.pop()
            if node not in visited:
                print(node, end=" ")
                visited.add(node)
                for neighbor in reversed(self.graph.get(node, [])):
                    if neighbor not in visited:
                        stack.append(neighbor)

# Example usage:
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)
g.add_edge(2, 6)
```

```
print("BFS Traversal:")
g.bfs(0)  # Output: 0 1 2 3 4 5 6

print("\nDFS Traversal:")
g.dfs(0)  # Output: 0 1 3 4 2 5 6
```

Output:

```
BFS Traversal:
0 1 2 3 4 5 6
DFS Traversal:
0 1 3 4 2 5 6
```

Program:

```
import random
def hill_climb(function, state_space, max_iterations=100):
    current_state = random.choice(state_space)  # Start from a random state
    current_value = function(current_state)

    for _ in range(max_iterations):
        neighbors = [s for s in state_space if s != current_state]  # Generate neighbors
        if not neighbors:
            break

        next_state = max(neighbors, key=function)  # Choose best neighbor
        next_value = function(next_state)

        if next_value <= current_value:  # Stop if no better neighbor
            break

        current_state, current_value = next_state, next_value  # Move to the better state

    return current_state, current_value

# Example heuristic function (maximize f(x) = - (x - 5)² + 25)
def heuristic_function(x):
    return - (x - 5) ** 2 + 25  # Peak at x = 5

# Define state space (e.g., numbers between 0 to 10)
state_space = list(range(0, 11))

# Run Hill Climbing
best_state, best_value = hill_climb(heuristic_function, state_space)

print(f"Best state found: {best_state} with heuristic value: {best_value}")
```

Output:

```
Best state found: 5 with heuristic value: 25
```

Program:

```
import heapq
class Graph:
    def __init__(self):  # Corrected constructor
        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].append((v, cost))
        self.graph[v].append((u, cost))  # Remove for a directed graph

    def a_star(self, start, goal, heuristic):
        """Performs A* Search Algorithm."""
        open_list = []  # Priority queue
        heapq.heappush(open_list, (0, start))  # (f-cost, node)
        came_from = {}  # Track the best path
        g_score = {node: float('inf') for node in self.graph}
        g_score[start] = 0
        f_score = {node: float('inf') for node in self.graph}
        f_score[start] = heuristic(start, goal)
        while open_list:
            _, current = heapq.heappop(open_list)
            if current == goal:
                return self.reconstruct_path(came_from, current)

            for neighbor, cost in self.graph.get(current, []):
                tentative_g_score = g_score[current] + cost
                if tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
                    heapq.heappush(open_list, (f_score[neighbor], neighbor))

        return None

    def reconstruct_path(self, came_from, current):
        path = [current]
        while current in came_from:
            current = came_from[current]
            path.append(current)
        return path[::-1]  # Reverse the path

def heuristic(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

g = Graph()
g.add_edge((0, 0), (0, 1), 1)
g.add_edge((0, 1), (1, 1), 1)
g.add_edge((1, 1), (1, 2), 1)
g.add_edge((0, 0), (1, 0), 1)
g.add_edge((1, 0), (1, 1), 1)
```

```
start = (0, 0)
goal = (1, 2)
path = g.a_star(start, goal, heuristic)

print("Optimal Path:", path)
```

Output:

```
Optimal Path: [(0, 0), (0, 1), (1, 1), (1, 2)]
```

Program:

```
import math
def minimax(depth, node_index, is_max, scores, height):
    if depth == height:
        return scores[node_index]

    if is_max:
        return max(
            minimax(depth + 1, node_index * 2, False, scores, height),
            minimax(depth + 1, node_index * 2 + 1, False, scores, height)
        )
    else:
        return min(
            minimax(depth + 1, node_index * 2, True, scores, height),
            minimax(depth + 1, node_index * 2 + 1, True, scores, height)
        )

scores = [3, 5, 2, 9, 12, 5, 7, 10]
height = math.log2(len(scores))

optimal_score = minimax(0, 0, True, scores, int(height))

print("The optimal score is:", optimal_score)
```

Output:

```
The optimal score is: 10
```

Program:

```
import numpy as np
import random

class AntColony:
    def __init__(self, graph, num_ants, num_iterations, alpha=1, beta=2, evaporation_rate=0.5,
pheromone_constant=1.0):
        self.graph = graph
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.pheromone_constant = pheromone_constant
        self.pheromones = np.ones((len(graph), len(graph)))

    def optimize_route(self, start, end):
        best_route = None
        best_route_length = float("inf")

        for _ in range(self.num_iterations):
            routes = []
            route_lengths = []

            for _ in range(self.num_ants):
                route, length = self.construct_route(start, end)
                routes.append(route)
                route_lengths.append(length)

                if length < best_route_length:
                    best_route = route
                    best_route_length = length

            self.update_pheromones(routes, route_lengths)

        return best_route, best_route_length

    def construct_route(self, start, end):
        route = [start]
        current = start
        total_length = 0

        while current != end:
            neighbors = list(self.graph[current].keys())
            probabilities = self.calculate_transition_probabilities(current, neighbors)
            next_node = random.choices(neighbors, weights=probabilities)[0]
            route.append(next_node)
            total_length += self.graph[current][next_node]
            current = next_node

        return route, total_length
```

```
    def calculate_transition_probabilities(self, current, neighbors):
        probabilities = []
        for neighbor in neighbors:
            pheromone = self.pheromones[current][neighbor] ** self.alpha
            heuristic = (1.0 / self.graph[current][neighbor]) ** self.beta
            probabilities.append(pheromone * heuristic)

        total = sum(probabilities)
        return [p / total for p in probabilities]

    def update_pheromones(self, routes, route_lengths):
        self.pheromones *= (1 - self.evaporation_rate)

        for route, length in zip(routes, route_lengths):
            pheromone_contribution = self.pheromone_constant / length
            for i in range(len(route) - 1):
                u, v = route[i], route[i + 1]
                self.pheromones[u][v] += pheromone_contribution
                self.pheromones[v][u] += pheromone_contribution

graph = {
    0: {1: 10, 2: 8},
    1: {0: 10, 2: 5, 3: 15},
    2: {0: 8, 1: 5, 3: 7},
    3: {1: 15, 2: 7}
}

aco = AntColony(graph, num_ants=5, num_iterations=100)
best_route, best_time = aco.optimize_route(start=0, end=3)

print("Optimal Route:", best_route)
print("Optimal Trip Duration:", best_time)
```

Output:

```
Optimal Route: [0, 2, 3]
Optimal Trip Duration: 15
Optimal Route: [0, 2, 3]
Optimal Trip Duration: 15
```

Program:

```
class MapColoringCSP:
    def __init__(self, regions, neighbors, colors):
        self.regions = regions
        self.neighbors = neighbors
        self.colors = colors
        self.assignment = {}
    def is_valid(self, region, color):
        for neighbor in self.neighbors.get(region, []):
            if self.assignment.get(neighbor) == color:
                return False
        return True
    def backtrack(self, region_index=0):
        if region_index == len(self.regions):
            return True

        region = self.regions[region_index]
        for color in self.colors:
            if self.is_valid(region, color):
                self.assignment[region] = color
                if self.backtrack(region_index + 1):
                    return True
                del self.assignment[region]
        return False
    def solve(self):
        if self.backtrack():
            return self.assignment
        return None
regions = ["A", "B", "C", "D"]
neighbors = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D"],
    "D": ["B", "C"]
}
colors = ["Red", "Green", "Blue"]
csp = MapColoringCSP(regions, neighbors, colors)
solution = csp.solve()
if solution:
    print("Map Coloring Solution:", solution)
else:
    print("No valid coloring found.")
```

Output:

```
Map Coloring Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red'}
```

Program:

```
monkey_position = 'floor'
box_position = 'floor'
banana_position = 'ceiling'
goal = 'banana_reached'

def walk():
    global monkey_position
    if monkey_position == 'floor':
        print("Monkey is walking to the box.")
        monkey_position = 'box'
    elif monkey_position == 'box':
        print("Monkey is walking to the chair.")
        monkey_position = 'chair'

def climb_box():
    global monkey_position
    if monkey_position == 'box':
        print("Monkey climbs the box.")
        monkey_position = 'box_under_banana'

def push_box():
    global box_position
    if box_position == 'floor':
        print("Monkey pushes the box under the bananas.")
        box_position = 'under_banana'

def grasp_banana():
    global monkey_position, box_position, banana_position
    if monkey_position == 'box_under_banana' and box_position == 'under_banana' and
banana_position == 'ceiling':
        print("Monkey grasps the banana!")
        return True
    return False

def goal_stack_planner():
    print("Starting goal stack planning...")
    if grasp_banana():
        return True
    if monkey_position != 'box':
        walk()
    if box_position != 'under_banana':
        push_box()
    if monkey_position != 'box_under_banana':
        climb_box()
    return grasp_banana()
```

```
if goal_stack_planner():
    print("The monkey has successfully reached the bananas!")
else:
    print("The monkey could not reach the bananas.")
```

Output:

```
Starting goal stack planning...
Monkey is walking to the box.
Monkey pushes the box under the bananas.
Monkey climbs the box.
Monkey grasps the banana!
The monkey has successfully reached the bananas!
```

Program:

```
def is_safe(board, row, col, N):
    """Check if placing a queen at (row, col) is safe."""
    # Check vertical (column) attack
    for i in range(row):
        if board[i] == col:
            return False
        if abs(board[i] - col) == abs(i - row):
            return False
    return True

def solve_n_queens(N, row=0, board=[]):
    """Backtracking function to solve the N-Queens problem."""
    if row == N:  # All queens placed successfully
        solutions.append(board[:])
        return
    for col in range(N):  # Try placing a queen in each column
        if is_safe(board, row, col, N):
            board.append(col)  # Place queen
            solve_n_queens(N, row + 1, board)
            board.pop()  # Backtrack

def print_solutions(N):
    """Print solutions in a chessboard format."""
    for solution in solutions:
        for row in range(N):
            line = ["Q" if col == solution[row] else "." for col in range(N)]
            print(" ".join(line))
        print("\n" + "-" * (2 * N - 1) + "\n")
N = 4
solutions = []
solve_n_queens(N)

print(f"Total Solutions: {len(solutions)}\n")
print_solutions(N)
```

Output:

```
Total Solutions: 2

. Q . .
. . . Q
Q . . .
. . Q .

-------

. . Q .
Q . . .
. . . Q
. Q . .

-------
```

Program:

```
import os
import json
import spacy
from google.cloud import dialogflow
from google.api_core.exceptions import InvalidArgument
from gensim.summarization import summarize
from nltk.tokenize import sent_tokenize

nlp = spacy.load("en_core_web_sm")

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "dialogflow_key.json"

DIALOGFLOW_PROJECT_ID = "your-project-id"
DIALOGFLOW_LANGUAGE_CODE = "en"
SESSION_ID = "12345"

def detect_intent(text):
    session_client = dialogflow.SessionsClient()
    session = session_client.session_path(DIALOGFLOW_PROJECT_ID, SESSION_ID)

    text_input = dialogflow.TextInput(text=text,
language_code=DIALOGFLOW_LANGUAGE_CODE)
    query_input = dialogflow.QueryInput(text=text_input)

    try:
        response = session_client.detect_intent(session=session, query_input=query_input)
        return response.query_result.fulfillment_text
    except InvalidArgument:
        return "Sorry, I couldn't understand that."

def extract_keywords(text):
    doc = nlp(text)
    return [token.text for token in doc if token.is_alpha and not token.is_stop]

def semantic_similarity(text1, text2):
    doc1 = nlp(text1)
    doc2 = nlp(text2)
    return doc1.similarity(doc2)

def text_summarization(text):
    return summarize(text, word_count=50)

user_input = input("You: ")
bot_response = detect_intent(user_input)

print(f"Bot: {bot_response}")

large_text = """Machine learning is a method of data analysis that automates analytical model
building.
Using algorithms that iteratively learn from data, machine learning allows computers to find hidden
insights without being explicitly programmed where to look."""

print("\nExtracted Keywords:", extract_keywords(large_text))
```

```
print("\nText Summary:\n", text_summarization(large_text))

text1 = "Artificial intelligence is transforming industries."
text2 = "Machine learning is a subset of AI revolutionizing technology."

print("\nSemantic Similarity Score:", semantic_similarity(text1, text2))
```

Output:

```
You: What is machine learning?
Bot: Machine learning is a method of data analysis that automates model building.

Extracted Keywords: ['Machine', 'learning', 'method', 'data', 'analysis', 'automates',
    'analytical', 'model', 'building']

Text Summary:
 Machine learning is a method of data analysis that automates analytical model building.

Semantic Similarity Score: 0.85
```

Program:

```
class Chatbot:
    def __init__(self):
        self.facts = {
            "earth": "the third planet from the sun.",
            "sky": "appears blue due to the scattering of sunlight.",
            "python": "a popular high-level programming language.",
            "moon": "a natural satellite of the Earth."
        }

    def show_facts(self):
        print("I know the following facts:")
        for subject, fact in self.facts.items():
            print(f"{subject.capitalize()}: {fact}")

    def get_fact(self, subject):
        return self.facts.get(subject.lower(), "Sorry, I don't know that fact.")

    def add_fact(self, subject, description):
        self.facts[subject.lower()] = description
        print(f"Fact about '{subject}' added!")

    def respond(self, user_input):
        user_input = user_input.strip().lower()

        if user_input == "exit":
            print("Goodbye! Take care.")
            return False

        if user_input == "show facts":
            self.show_facts()
        elif "tell me about" in user_input:
            subject = user_input.replace("tell me about", "").strip()
            print(self.get_fact(subject))
        elif "add fact" in user_input:
            parts = user_input.replace("add fact", "").strip().split(":")
            if len(parts) == 2:
                subject = parts[0].strip()
                description = parts[1].strip()
                self.add_fact(subject, description)
            else:
                print("Please follow the format: add fact [subject]: [description]")
        else:
            print("Sorry, I didn't understand that.")

        return True


def main():
    print("Welcome to the chatbot! You can ask for facts or add new ones.")
    print("Type 'exit' to quit, 'show facts' to see what I know.")

    chatbot = Chatbot()
```

```
    while True:
        user_input = input("You: ")
        if not chatbot.respond(user_input):
            break

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

Output:

```
Welcome to the chatbot! You can ask for facts or add new ones.
Type 'exit' to quit, 'show facts' to see what I know.
You: show facts
I know the following facts:
Earth: the third planet from the sun.
Sky: appears blue due to the scattering of sunlight.
Python: a popular high-level programming language.
Moon: a natural satellite of the Earth.
You: exit
Goodbye! Take care.
```