**Experiment 1**

**Source code:**

def water\_jug\_dfs(capacity1, capacity2, target):

    visited = set()

    path = []

    def dfs(jug1, jug2):

        if (jug1, jug2) in visited:

            return False

        visited.add((jug1, jug2))

        path.append((jug1, jug2))

        if jug1 == target or jug2 == target:

            return True

        if dfs(3, jug2):

            return True

        if dfs(jug1, 5):

            return True

        if dfs(0, jug2):

            return True

        if dfs(jug1, 0):

            return True

        if dfs(max(0, jug1 - (5 - jug2)), min(5, jug1 + jug2)):

            return True

        if dfs(min(3, jug1 + jug2), max(0, jug2 - (3 - jug1))):

            return True

        path.pop()

        return False

    dfs(0, 0)

    return path

capacity1 = 3

capacity2 = 4

target = 2

solution = water\_jug\_dfs(capacity1, capacity2, target)

if solution:

    print("Solution steps in DFS:")

    for step in solution:

        print(step)

else:

    print("No solution found.")

**Experiment – 2.2**

**Aim:** Write a program to solve the Water Jug Problem (Using BFS).

**Algorithm:**

* Initialise a queue to implement **BFS.**
* Since, initially, both the jugs are empty, insert the state {0, 0} into the queue.
* Perform the following state, till the queue becomes empty:
  + Pop out the first element of the queue.
  + If the value of popped element is equal to **Z**, return True.
  + Let **X\_left**and **Y\_left** be the amount of water left in the jugs respectively.
  + Now perform the **fill** operation:
    - If the value of **X\_left < X,**insert ({**X\_left, Y**}) into the hashmap, since this state hasn’t been visited and some water can still be poured in the jug.
    - If the value of **Y\_left < Y,**insert ({**Y\_left, X**}) into the hashmap, since this state hasn’t been visited and some water can still be poured in the jug.
  + Perform the **empty** operation:
    - If the state **({0, Y\_left})** isn’t visited, insert it into the hashmap, since we can empty any of the jugs.
    - Similarly, if the state **({X\_left, 0)** isn’t visited, insert it into the hashmap, since we can empty any of the jugs.
  + Perform the **transfer of water** operation:
    - **min({X-X\_left, Y})**can be poured from second jug to first jug. Therefore, in case – **{X +** **min({X-X\_left, Y}) , Y – min({X-X\_left, Y})** isn’t visited, put it into hashmap.
    - **min({X\_left, Y-Y\_left})**can be poured from first jug to second jug. Therefore, in case – **{X\_left – min({X\_left, Y – X\_left}) , Y + min({X\_left, Y – Y\_left})** isn’t visited, put it into hashmap.
* Return False, since, it is not possible to measure **Z** litres.

**Source code:**

from collections import deque

def bfs(a, b, target):

    visited = set()

    queue = deque()

    initial\_state = (0, 0)

    queue.append((initial\_state, []))

    visited.add(initial\_state)

    while queue:

        current\_state, path = queue.popleft()

        x, y = current\_state

        if x == target or y == target:

            print("Solution step in BFS:")

            for state in path:

                print(state)

            return True

        if (a, y) not in visited:

            new\_state = (a, y)

            new\_path = path + [new\_state]

            queue.append((new\_state, new\_path))

            visited.add(new\_state)

        if (x, b) not in visited:

            new\_state = (x, b)

            new\_path = path + [new\_state]

            queue.append((new\_state, new\_path))

            visited.add(new\_state)

        if (0, y) not in visited:

            new\_state = (0, y)

            new\_path = path + [new\_state]

            queue.append((new\_state, new\_path))

            visited.add(new\_state)

        if (x, 0) not in visited:

            new\_state = (x, 0)

            new\_path = path + [new\_state]

            queue.append((new\_state, new\_path))

            visited.add(new\_state)

        pour = min(x, b - y)

        if (x - pour, y + pour) not in visited:

            new\_state = (x - pour, y + pour)

            new\_path = path + [new\_state]

            queue.append((new\_state, new\_path))

            visited.add(new\_state)

        pour = min(y, a - x)

        if (x + pour, y - pour) not in visited:

            new\_state = (x + pour, y - pour)

            new\_path = path + [new\_state]

            queue.append((new\_state, new\_path))

            visited.add(new\_state)

    return False

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

    a = 3

    b = 4

    target = 2

    if not bfs(a, b, target):

        print("No solution exists!")

**Experiment – 3**

**Aim:** Write a program to solve a robot traversal problem using the Means-End Analysis technique.

**Algorithm:** Robot Traversal Using Means-End Analysis

**Input:**

* A grid representing the environment (0: empty space, 1: obstacle)
* Starting position (x, y)
* Goal position (x, y)

**Output:**

* Path from start to goal position if it exists
* Otherwise, indicate no solution exists

**Steps:**

1. **Define the difference function** that calculates the Manhattan distance between the current state and the goal state.
2. **Initialize the current state** as the starting position.
3. **While the current state is not the goal state**, do the following:
   * Calculate the difference between the current state and the goal state.
   * Generate all possible moves (operators) from the current state.
   * For each operator, calculate how much it reduces the difference.
   * Select the operator that maximally reduces the difference.
   * If no operator reduces the difference, backtrack or try alternative approaches.
   * Apply the selected operator to move to a new state.
   * Add the new state to the solution path.
4. **If the goal state is reached**, return the solution path.
5. **If all possibilities are exhausted and the goal is not reached**, report no solution.

**Source code:**

def means\_end\_analysis(start, goal):

path = []

current = start

print(f"Starting MEA from {start} to {goal}\n")

while current != goal:

path.append(current)

x1, y1 = current

x2, y2 = goal

if x1 < x2:

current = (x1 + 1, y1)

action = "Move Down"

elif x1 > x2:

current = (x1 - 1, y1)

action = "Move Up"

elif y1 < y2:

current = (x1, y1 + 1)

action = "Move Right"

elif y1 > y2:

current = (x1, y1 - 1)

action = "Move Left"

print(f"Action: {action} -> New Position: {current}")

path.append(goal)

print("\nReached the goal!")

return path

start\_pos = (0, 0)

goal\_pos = (3, 4)

final\_path = means\_end\_analysis(start\_pos, goal\_pos)

print("\nPath followed:")

for step in final\_path:

print(step)

**Experiment – 4**

**Aim:** Write a program to convert Predicate Logic formulas to Propositional Logic.

**Predicate Logic vs Propositional Logic:** Predicate Logic extends Propositional Logic by allowing variables, quantifiers, and predicates. While Propositional Logic deals with simple statements that are either true or false, Predicate Logic can represent more complex statements involving variables, properties, and relationships. Converting from Predicate Logic to Propositional Logic involves eliminating quantifiers and variables through techniques like instantiation.

**Algorithm: Converting Predicate Logic to Propositional Logic**

**Input:**

* A predicate logic formula with quantifiers, variables, and predicates

**Output:**

* Equivalent propositional logic formula or formulas

**Steps:**

1. **Parse the predicate logic formula** to identify quantifiers, variables, predicates, and logical operators.
2. **Eliminate universal quantifiers (∀)** through instantiation over the domain of discourse.
3. **Eliminate existential quantifiers (∃)** by replacing them with disjunctions over the domain.
4. **Replace predicates with propositional variables** for each instantiated object.
5. **Simplify the resulting propositional formula** using logical equivalences if necessary.

**Source code:**

def convert\_predicates\_to\_propositions(predicates, operator='∧'):

    mapping = {}

    proposition = ""

    variable = ord('P')

    print("Mapping of predicates to propositional variables:\n")

    for predicate in predicates:

        if predicate not in mapping:

            key = chr(variable)

            mapping[predicate] = key

            print(f"{key} = {predicate}")

            variable += 1

    print("\nPropositional logic form:\n")

    # Join mapped variables using the chosen logical operator

    proposition = f" {operator} ".join(mapping[pred] for pred in predicates)

    print(proposition)

    return proposition, mapping

# Example predicates

predicates = [

    "Student(John)",

    "Studies(John, Math)",

    "Studies(John, Math) → Passes(John, Math)"

]

convert\_predicates\_to\_propositions(predicates)

**Experiment – 5**

**Aim:** Write a program to implement a solution to the N-Queens problem.

**N-Queens Problem:** The N-Queens problem involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. In chess, a queen can attack horizontally, vertically, and diagonally. The challenge is to find a configuration where no two queens share the same row, column, or diagonal.

**Approach: Backtracking with Constraint Satisfaction**

Backtracking is particularly well-suited for the N-Queens problem as it systematically explores the solution space while pruning branches that cannot lead to valid solutions. We place queens one by one in different columns, starting from the leftmost column. When we place a queen in a position, we check if it conflicts with already placed queens. If there's no conflict, we mark this position as part of the solution and recursively check if placing queens in remaining columns leads to a solution.

**Algorithm: N-Queens Using Backtracking**

**Input:**

* N: The size of the board and the number of queens to place

**Output:**

* A valid arrangement of N queens on an N×N board, or indication that no solution exists

**Steps:**

1. Start in the leftmost column.
2. If all queens are placed, return the solution.
3. Try all rows in the current column. For each row, do the following:
   * If the queen can be placed safely in this row, mark this cell and move to the next column.
   * If placing the queen in the current row leads to a solution, return true.
   * If placing the queen in the current row does not lead to a solution, unmark this cell (backtrack) and try the next row.
4. If all rows have been tried and none worked, return false to trigger backtracking.

**Source code:**

def print\_board(board):

    for row in board:

        print(" ".join(row))

    print()

def is\_safe(board, row, col, n):

    # Check vertical

    for i in range(row):

        if board[i][col] == 'Q':

            return False

    # Check upper-left diagonal

    i, j = row, col

    while i >= 0 and j >= 0:

        if board[i][j] == 'Q':

            return False

        i -= 1

        j -= 1

    # Check upper-right diagonal

    i, j = row, col

    while i >= 0 and j < n:

        if board[i][j] == 'Q':

            return False

        i -= 1

        j += 1

    return True

def solve\_n\_queens(board, row, n, solutions):

    if row == n:

        # All queens are placed

        solution = ["".join(r) for r in board]

        solutions.append(solution)

        return

    for col in range(n):

        if is\_safe(board, row, col, n):

            board[row][col] = 'Q'  # Place queen

            solve\_n\_queens(board, row + 1, n, solutions)

            board[row][col] = '.'  # Backtrack

def n\_queens(n):

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

    solutions = []

    solve\_n\_queens(board, 0, n, solutions)

    print(f"Total solutions for {n}-Queens: {len(solutions)}\n")

    for sol in solutions:

        for row in sol:

            print(row)

        print()

# Example usage

n = 4

n\_queens(n)

**Experiment – 6**

**Aim:** Write a program to implement the Travelling Salesman Problem in Artificial Intelligence.

**Travelling Salesman Problem:** In AI, the Traveling Salesman Problem (TSP) is a classic combinatorial optimization challenge where the goal is to find the shortest possible route that visits a set of cities exactly once and then returns to the starting city. It's a fundamental problem with wide-ranging applications in logistics, delivery, and other areas where efficient routing is crucial.

TSP has real-world applications in logistics, route optimization, circuit design, robotics, and supply chain management. The primary constraints of solving TSP include:

* Ensuring each city is visited once.
* Finding the optimal path while minimizing cost and time.
* Handling large-scale problems where brute force solutions become impractical.

Traditional methods like brute force and dynamic programming work well for small-scale TSP instances but become inefficient for larger datasets. AI and machine learning algorithms provide heuristic and metaheuristic solutions that offer near-optimal results efficiently. Techniques like genetic algorithms, ant colony optimization, and neural networks allow AI-driven systems to solve TSP faster and more effectively.

**Approaches to Solving TSP**

The Traveling Salesman Problem (TSP) is computationally complex, and finding an optimal solution requires efficient algorithms. Various approaches, ranging from exact algorithms to AI-driven heuristic solutions, help tackle TSP efficiently.

**Exact Algorithms**

* **Brute Force Approach:** The brute force method checks all possible permutations of routes and selects the shortest one. While it guarantees the optimal solution, its  complexity makes it impractical for large datasets.
* **Dynamic Programming (Held-Karp Algorithm):** The Held-Karp algorithm (also known as the Bellman-Held-Karp algorithm) improves efficiency using memoization and state transition equations. It reduces complexity to ² but still becomes impractical for large-scale problems.

**Approximation Algorithms**

* **Greedy Algorithm:** A simple approach where the salesman always selects the shortest available path. While fast, it does not always yield optimal solutions.
* **Nearest Neighbor Algorithm:** This heuristic selects the nearest unvisited city at each step. It provides a quick approximation but may lead to suboptimal paths due to local optima traps.

**AI-Based Solutions**

* **Genetic Algorithms:** This evolutionary technique uses mutation, crossover, and selection to refine routes iteratively, making it effective for large datasets.
* **Ant Colony Optimization:** Inspired by ant behavior, this algorithm simulates pheromone trails to find the most efficient paths. It works well for dynamic and complex routing problems.
* **Neural Networks:** Deep learning models, such as Hopfield networks and reinforcement learning, can learn to predict optimized routes based on historical data, improving scalability for real-world logistics.

**Algorithm**

Function TSP\_Brute\_Force(Graph, Start):

Input:

Graph - A 2D matrix representing the cost between cities

Start - The starting city index

Output:

MinPath - Minimum travel cost

BestPath - The optimal path taken

Initialize:

N ← Number of cities (length of Graph)

Vertices ← list of all cities excluding Start

MinPath ← ∞

BestPath ← Empty list

For each Permutation in AllPermutations(Vertices):

CurrentWeight ← 0

K ← Start

Path ← [Start]

For each City in Permutation:

CurrentWeight ← CurrentWeight + Graph[K][City]

K ← City

Append City to Path

CurrentWeight ← CurrentWeight + Graph[K][Start] // return to start

Append Start to Path

If CurrentWeight < MinPath:

MinPath ← CurrentWeight

BestPath ← Path

Return MinPath, BestPath

**Explanation:**

* AllPermutations(Vertices) generates all possible orderings of the cities excluding the starting city.
* Graph[i][j] gives the travel cost from city i to city j.
* The algorithm checks every possible route that starts and ends at the Start city.
* It keeps track of the path with the minimum cost and returns it.

**Source Code –**

from itertools import permutations

def tsp\_brute\_force(graph, start):

    n = len(graph)

    vertices = list(range(n))

    vertices.remove(start)

    min\_path = float('inf')

    best\_path = []

    for perm in permutations(vertices):

        current\_pathweight = 0

        k = start

        path = [start]

        for j in perm:

            current\_pathweight += graph[k][j]

            k = j

            path.append(j)

        current\_pathweight += graph[k][start]

        path.append(start)

        if current\_pathweight < min\_path:

            min\_path = current\_pathweight

            best\_path = path

    return min\_path, best\_path

# Example usage

graph = [

    [0, 10, 15, 20],

    [10, 0, 35, 25],

    [15, 35, 0, 30],

    [20, 25, 30, 0]

]

cost, path = tsp\_brute\_force(graph, 0)

print("Minimum cost:", cost)

print("Best path:", path)

**Experiment – 7**

**Aim:** Write a program to implement the Grammar Checker using language \_tool\_python.

**Concept:**

This project demonstrates the use of Natural Language Processing (NLP) in Python to perform automated grammar checking. The tool is built using the language\_tool\_python library, which is a Python wrapper for [LanguageTool](https://languagetool.org), a popular open-source grammar and spell-checking engine.

The core functionality of the project is:

* Grammar Analysis: It analyzes a given English sentence to detect grammatical, punctuation, and stylistic issues.
* Contextual Error Detection: Unlike basic spell checkers, this tool can detect contextual errors such as subject-verb agreement, article misuse, or verb tense issues.
* Suggestions and Corrections: For each detected error, the tool provides a message explaining the issue, a list of suggested corrections, and the position of the error in the text.
* Auto-Correction: The tool also auto-generates a corrected version of the sentence by applying the most suitable suggestion for each error found.

This program is a practical example of how AI and NLP can be integrated into applications to enhance writing accuracy, readability, and user communication. It can be extended into more advanced applications such as grammar-aware text editors, educational writing aids, or chatbots that provide real-time feedback.

**Algorithm:**

Step 1: Import the language\_tool\_python library.  
Step 2: Initialize the grammar checker with the English language (en-US).  
Step 3: Prompt the user to input a sentence.  
Step 4: Use the check() method to find grammar issues in the input text.  
Step 5: Display the original sentence.  
Step 6: If any matches (errors) are found:  
     Display the error message, suggested replacements, and the position of the error in the sentence.  
Step 7: If no errors are found, notify the user accordingly.  
Step 8: Use the correct() method to apply the suggested corrections and print the corrected sentence.

**Source code –**

import language\_tool\_python

# Initialize the grammar checker

tool = language\_tool\_python.LanguageTool('en-US')

# Take user input

text = input("Enter a sentence: ")

# Check for grammar issues

matches = tool.check(text)

# Show original sentence

print("\nOriginal Sentence:", text)

# Print suggestions

if matches:

    print("\nSuggestions:")

    for match in matches:

        print(f"- {match.message}")

        print(f"  ➤ Suggested Correction(s): {match.replacements}")

        print(f"  ➤ Error at position {match.offset}-{match.offset + match.errorLength}\n")

else:

    print("\n No grammar issues found.")

# Auto-correct and display corrected sentence

corrected\_text = language\_tool\_python.utils.correct(text, matches)

print("Corrected Sentence:", corrected\_text)