Documentation

&

Code-Link

1. Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

collab-link

```python
"""Implementation of Depth First Search (DFS) Algorithm on Water Jug Problem:

Functions:
1. dfs_water_jug(x, y, z, path, visited):
    - This recursive function explores all possible states of the water jugs until
the target amount z is achieved.
    - It uses backtracking to explore all possible combinations of jug operations:
filling, emptying, and pouring water between the jugs.
    - Parameters:
        - x: Capacity of the first water jug.
        - y: Capacity of the second water jug.
        - z: Target amount of water to measure.
        - path: List to store the sequence of jug operations.
        - visited: Set to track visited states to avoid revisiting the same state.

2. water_jug_dfs(x_capacity, y_capacity, z):
    - This function serves as the entry point for the DFS algorithm.
    - It initializes the search with both jugs empty and calls the dfs_water_jug
function recursively to explore all possible states.
    - Parameters:
        - x_capacity: Capacity of the first water jug.
        - y_capacity: Capacity of the second water jug.
        - z: Target amount of water to measure.

Usage:
- Define the capacities of the water jugs (x_capacity and y_capacity) and the
target amount of water (z).
- Call the water_jug_dfs function with the specified capacities and target amount.
- The function explores all possible states of the water jugs and prints the
sequence of jug operations required to measure the target amount of water.
"""

def dfs_water_jug(x, y, z, x_capacity, y_capacity, path, visited):
    if z == 0:
        print("Solution found:", path)
        return True

    visited.add((x, y))

    if x < x_capacity and (x_capacity, y) not in visited:
        print("Filling Jug X")
        if dfs_water_jug(x_capacity, y, z, x_capacity, y_capacity, path + ["Fill
Jug X"], visited):
            return True

    if y < y_capacity and (x, y_capacity) not in visited:
        print("Filling Jug Y")
        if dfs_water_jug(x, y_capacity, z, x_capacity, y_capacity, path + ["Fill
Jug Y"], visited):
            return True

    if x > 0 and (0, y) not in visited:
        print("Emptying Jug X")
        if dfs_water_jug(0, y, z, x_capacity, y_capacity, path + ["Empty Jug X"],
visited):
            return True

    if y > 0 and (x, 0) not in visited:
        print("Emptying Jug Y")
        if dfs_water_jug(x, 0, z, x_capacity, y_capacity, path + ["Empty Jug Y"],
visited):
            return True
    if x > 0 and y < y_capacity:
        amt = min(x, y_capacity - y)
        print(f"Pouring {amt} from Jug X to Jug Y")
        if (x - amt, y + amt) not in visited:
            if dfs_water_jug(x - amt, y + amt, z, x_capacity, y_capacity, path +
[f"Pour {amt} from Jug X to Jug Y"], visited):
                return True

    if y > 0 and x < x_capacity:
        amt = min(y, x_capacity - x)
        print(f"Pouring {amt} from Jug Y to Jug X")
        if (x + amt, y - amt) not in visited:
            if dfs_water_jug(x + amt, y - amt, z, x_capacity, y_capacity, path +
[f"Pour {amt} from Jug Y to Jug X"], visited):
                return True

    return False

def water_jug_dfs(x_capacity, y_capacity, z):
    path = []
    visited = set()
    dfs_water_jug(0, 0, z, x_capacity, y_capacity, path, visited)

x_capacity = 4
y_capacity = 3
z = 2
water_jug_dfs(x_capacity, y_capacity, z)
```

2. Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python collab-link

```
"""
    Problem Description: The Missionaries and Cannibals problem involves
transporting three missionaries and three cannibals from one side of a river to
the other using a boat.
                        The boat can carry at most two people, and there must
never be more cannibals than missionaries on either side of the river, or the
cannibals will eat the missionaries.
                        The goal is to find a sequence of boat crossings that
achieves the transfer of all missionaries and cannibals from one side of the river
to the other, adhering to the constraints.

    State class: - Represents the state of the problem with the number of
missionaries, cannibals, and the boat's position.
                - is_valid(): Checks if the current state is valid according to
the problem constraints.
                - is_goal(): Checks if the current state is the goal state where
all missionaries and cannibals are on the other side of the river.
                - __eq__(): Compares two states for equality.
                - __hash__(): Generates a hash for a state object.


    successors function: - Generates all possible valid moves (successor states)
from the current state.
                        - Handles moving missionaries and cannibals between sides
of the river while ensuring validity.

    best_first_search function: - Performs the Best First Search algorithm to find
the solution path.
                                - Starts with the initial state and explores
successor states based on a heuristic (sum of missionaries and cannibals).
                                - Returns the solution path if found, otherwise
prints "No solution found".

    Algorithm: Input: None (parameters are predefined within the function)
                Output: Solution path (if found)

                1. Initialization:
                  - Set the initial state with 3 missionaries, 3 cannibals, and
the boat on the initial side.
                  - Initialize an empty set to store visited states.
                  - Initialize a queue with the initial state as the starting
path.

                2. Exploration:
                  - While the queue is not empty:
                    - Pop a path from the front of the queue.
                    - Get the current state from the end of the path.
                    - If the current state is the goal state (0 missionaries, 0
cannibals, boat on the initial side), return the path as the solution.
                    - Add the current state to the visited set.
                    - Generate successor states for the current state.

                3. Search:
                  - For each successor state:
                    - If the state has not been visited:
                      - Create a new path by appending the successor state to the
current path.
                      - Add the new path to the queue.

                4. Repeat steps 2-3 until a solution is found or the queue is
empty.

                5. If the queue becomes empty and no solution is found, return "No
solution found".
```

```
"""
class State:
    def __init__(self, missionaries, cannibals, boat):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boat = boat

    def is_valid(self):
        if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or
self.cannibals > 3:
            return False
        if self.missionaries < self.cannibals and self.missionaries > 0:
            return False
        if 3 - self.missionaries < 3 - self.cannibals and 3 - self.missionaries >
0:
            return False
        return True

    def is_goal(self):
        return self.missionaries == 0 and self.cannibals == 0 and self.boat == 0

    def __eq__(self, other):
        return self.missionaries == other.missionaries and self.cannibals ==
other.cannibals and self.boat == other.boat

    def __hash__(self):
        return hash((self.missionaries, self.cannibals, self.boat))


def successors(current_state):
    children = []
    if current_state.boat == 1:
        new_state = State(current_state.missionaries, current_state.cannibals - 2,
0)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries - 2, current_state.cannibals,
0)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries - 1, current_state.cannibals
- 1, 0)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries, current_state.cannibals - 1,
0)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries - 1, current_state.cannibals,
0)
        if new_state.is_valid():
            children.append(new_state)
    else:
        new_state = State(current_state.missionaries, current_state.cannibals + 2,
1)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries + 2, current_state.cannibals,
1)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries + 1, current_state.cannibals
+ 1, 1)
        if new_state.is_valid():
            children.append(new_state)
        new_state = State(current_state.missionaries, current_state.cannibals + 1,
1)
        if new_state.is_valid():
                children.append(new_state)
            new_state = State(current_state.missionaries + 1, current_state.cannibals,
1)
            if new_state.is_valid():
                children.append(new_state)
        return children


def best_first_search():
    m = int(input("Enter the number of missionaries: "))
    c = int(input("Enter the number of cannibals: "))
    initial_state = State(m, c, 1)
    visited = set()
    queue = [[initial_state]]
    while queue:
        path = queue.pop(0)
        current_state = path[-1]
        if current_state.is_goal():
            return path
        visited.add(current_state)
        for child in successors(current_state):
            if child not in visited:
                new_path = list(path)
                new_path.append(child)
                queue.append(new_path)
                queue.sort(key=lambda x: (x[-1].missionaries + x[-1].cannibals),
reverse=True)


if __name__ == "__main__":
    solution = best_first_search()
    if solution:
        for i, state in enumerate(solution):
            print(f"Step {i}:
{state.missionaries},{state.cannibals},{state.boat}")
    else:
        print("No solution found.")
```

3. Implement A* Search algorithm

colab-link

```
"""
Classes:
- Node: Represents a node in the search graph.
- PriorityQueue: Implements a priority queue for efficient retrieval of nodes with
the lowest priority.

Functions:
- astar_search: Performs A* search to find the shortest path from a start node to
a goal node.
- euclidean_distance: Calculates the Euclidean distance between two points, used
as the heuristic function for A* search.
- get_point: Prompts the user to input coordinates for a point.

Algorithm:
Input: Start node, goal node, successors function, and heuristic function
Output: Solution path (if found)

1. Initialization:
  - Create an empty priority queue to store nodes to be explored.
  - Add the start node to the priority queue with a priority of 0.
  - Create an empty dictionary to store the cost from the start node to each node.
  - Set the cost of the start node to 0.
  - Create an empty dictionary to store the parent node for each node.

2. Exploration:
  - While the priority queue is not empty:
    - Get the node with the lowest priority from the priority queue.
    - If the node is the goal node:
      - Reconstruct and return the path from the start node to the goal node using
the parent pointers.
    - Otherwise:
      - Generate successor nodes for the current node using the successors
function.
      - For each successor node:
        - Calculate the cost from the start node to the successor node.
        - If the cost is lower than the previously recorded cost for the successor
node or the successor node is not yet visited:
          - Update the cost of the successor node.
          - Calculate the priority of the successor node (cost + heuristic).
          - Add the successor node to the priority queue with the calculated
priority.
          - Set the parent of the successor node to the current node.

3. Termination:
  - If the priority queue becomes empty and the goal node has not been reached,
return "No solution found".
"""
import heapq

def astar_search(start, goal, successors, heuristic):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    cost_so_far = {start: 0}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            while current is not None:
                path.append(current)
                current = came_from.get(current)
            return path[::-1]
        for next_state, cost in successors(current):
            new_cost = cost_so_far[current] + cost
            if next_state not in cost_so_far or new_cost <
cost_so_far[next_state]:
                cost_so_far[next_state] = new_cost
                priority = new_cost + heuristic(next_state, goal)
                heapq.heappush(open_set, (priority, next_state))
                came_from[next_state] = current

    return None

def euclidean_distance(p1, p2):
    return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

def get_neighbors(state):
    x, y = state
    neighbors = []

    # Define possible movements (up, down, left, right)
    movements = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    for dx, dy in movements:
        new_x, new_y = x + dx, y + dy
        # Check if the new position is within bounds
        if 0 <= new_x <= 5 and 0 <= new_y <= 5:
            neighbors.append(((new_x, new_y), 1))  # Cost of moving to a neighbor
is 1

    return neighbors


start_state = (0, 0)
goal_state = (5, 5)

path = astar_search(start_state, goal_state, get_neighbors, euclidean_distance)
print("Path:", path)
```

4. Implement AO* Search algorithm

collab-link

```python
"""
Algorithm:
Input: Start node, goal node, successors function, heuristic function, and
optimistic heuristic function
Output: Solution path (if found)

1. Initialization:
  - Create an empty priority queue to store nodes to be explored.
  - Add the start node to the priority queue with a priority of 0.
  - Create an empty dictionary to store the cost from the start node to each node.
  - Set the cost of the start node to 0.
  - Create an empty dictionary to store the parent node for each node.

2. Exploration:
  - While the priority queue is not empty:
    - Get the node with the lowest priority from the priority queue.
    - If the node is the goal node:
      - Reconstruct and return the path from the start node to the goal node using
the parent pointers.
    - Otherwise:
      - Generate successor nodes for the current node using the successors
function.
      - For each successor node:
        - Calculate the cost from the start node to the successor node.
        - If the cost is lower than the previously recorded cost for the successor
node or the successor node is not yet visited:
          - Update the cost of the successor node.
          - Calculate the priority of the successor node (cost + heuristic).
          - Add the successor node to the priority queue with the calculated
priority.
          - Set the parent of the successor node to the current node.

3. Termination:
  - If the priority queue becomes empty and the goal node has not been reached,
return "No solution found".
"""
import heapq

def aostar_search(start, goal, successors, heuristic, optimistic_heuristic):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    cost_so_far = {start: 0}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            while current is not None:
                path.append(current)
                current = came_from.get(current)
            return path[::-1]

        for next_state, cost in successors(current):
            new_cost = cost_so_far[current] + cost
            if next_state not in cost_so_far or new_cost <
cost_so_far[next_state]:
                cost_so_far[next_state] = new_cost
                optimistic_estimate = optimistic_heuristic(next_state, goal)
                priority = new_cost + heuristic(next_state, goal) +
optimistic_estimate
                heapq.heappush(open_set, (priority, next_state))
                came_from[next_state] = current
    return None

def euclidean_distance(p1, p2):
    return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

def manhattan_distance(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def get_neighbors(state):
    x, y = state
    movements = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    neighbors = []
    for dx, dy in movements:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x <= 5 and 0 <= new_y <= 5:
            neighbors.append(((new_x, new_y), 1))
    return neighbors

start_state = (0, 0)
goal_state = (5, 5)

path = aostar_search(start_state, goal_state, get_neighbors, euclidean_distance,
manhattan_distance)
print("Path:", path)
```

5. Solve 8-Queens Problem with suitable assumptions collab-link

```
"""
Problem Description:
The 8-Queens problem involves placing 8 queens on an 8x8 chessboard such that no
two queens attack each other. Queens can attack each other if they share the same
row, column, or diagonal. The goal is to find a configuration of queens that
satisfies these conditions.

Functions:
- aostar_search: Performs AO* search to find a solution path from a start state to
a goal state.
- heuristic: Evaluates the number of conflicts or attacks between queens on the
chessboard.
- optimistic_heuristic: Provides an optimistic estimate of the remaining cost to
reach the goal state (always returns 0 for this problem).
- get_successors: Generates successor states by placing a queen in each column of
the next row where no queens already exist.

Algorithm:
Input: Start state, goal state, successors function, heuristic function, and
optimistic heuristic function
Output: Solution path (if found)

1. Initialization:
  - Create an empty priority queue to store states to be explored.
  - Add the start state to the priority queue with a priority of 0.
  - Create an empty dictionary to store the cost from the start state to each
state.
  - Set the cost of the start state to 0.
  - Create an empty dictionary to store the parent state for each state.

2. Exploration:
  - While the priority queue is not empty:
    - Get the state with the lowest priority from the priority queue.
    - If the state is the goal state:
      - Reconstruct and return the path from the start state to the goal state
using the parent pointers.
    - Otherwise:
      - Generate successor states for the current state using the successors
function.
      - For each successor state:
        - Calculate the cost from the start state to the successor state.
        - If the cost is lower than the previously recorded cost for the successor
state or the successor state is not yet visited:
          - Update the cost of the successor state.
          - Calculate the priority of the successor state (cost + heuristic +
optimistic heuristic).
          - Add the successor state to the priority queue with the calculated
priority.
          - Set the parent of the successor state to the current state.

3. Termination:
  - If the priority queue becomes empty and the goal state has not been reached,
return "No solution found".
"""
def is_safe(board, row, col):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i] == col:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i] == j:
            return False
    # Check upper diagonal on right side
    for i, j in zip(range(row, -1, -1), range(col, 8)):
        if board[i] == j:
            return False

    return True

def solve_queens_util(board, row):
    if row >= 8:
        return True

    for col in range(8):
        if is_safe(board, row, col):
            board[row] = col
            if solve_queens_util(board, row + 1):
                return True
            board[row] = -1

    return False

def solve_queens():
    board = [-1] * 8  # Initialize the board with all positions as unoccupied
    if solve_queens_util(board, 0):
        return board
    else:
        return None

def print_board(board):
    for row in board:
        line = ['.'] * 8
        line[row] = 'Q'
        print(' '.join(line))

# Solve and print the solution
solution = solve_queens()
if solution:
    print_board(solution)
else:
    print("No solution found.")
```

6. Implementation of TSP using heuristic approach [collab-link]

```python
"""
Implementation of TSP using the Nearest Neighbor heuristic algorithm:

Function:
- nearest_neighbor: Finds a near-optimal solution to the TSP using the Nearest
Neighbor heuristic algorithm.

Algorithm:
1. Start with a graph representing distances between cities and specify the
starting city for the TSP.
2. The nearest_neighbor function iteratively selects the nearest unvisited city to
the current city until all cities are visited.
3. At each step, it finds the nearest unvisited city to the current city and adds
it to the tour.
4. The algorithm continues until all cities are visited, resulting in a complete
tour.
5. Finally, the function returns the shortest tour (list of cities) and its total
length.

Example:
- Define a graph representing distances between cities and specify the starting
city.
- Call the nearest_neighbor function with the graph and starting city.
- The function returns the shortest tour (list of cities) and its total length.

Note:
- The Nearest Neighbor algorithm provides a quick heuristic to find a near-optimal
solution to the TSP.
- It does not guarantee an optimal solution but often produces good results,
especially for small to medium-sized instances of the problem.
"""
"""
import sys

def nearest_neighbor(graph, start):
    visited = [start]
    total_distance = 0

    while len(visited) < len(graph):
        nearest_city = None
        min_distance = sys.maxsize

        for city in graph:
            if city not in visited and graph[start][city] < min_distance:
                nearest_city = city
                min_distance = graph[start][city]

        visited.append(nearest_city)
        total_distance += min_distance
        start = nearest_city

    total_distance += graph[visited[-1]][visited[0]]  # Return to the starting
city to complete the tour

    return visited, total_distance

# Example graph representing distances between cities
graph = {
    'A': {'A': 0, 'B': 2, 'C': 5, 'D': 9},
    'B': {'A': 2, 'B': 0, 'C': 2, 'D': 4},
    'C': {'A': 5, 'B': 2, 'C': 0, 'D': 3},
    'D': {'A': 9, 'B': 4, 'C': 3, 'D': 0}
}

start_city = 'A'  # Starting city for the TSP
```

```python
# Find the shortest tour and its length using the Nearest Neighbor algorithm
shortest_tour, tour_length = nearest_neighbor(graph, start_city)

# Print the result
print("Shortest tour:", shortest_tour)
print("Tour length:", tour_length)
```

7. **Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining** collab-link

```python
"""
Function:
- backward_chaining: Attempts to infer a goal from a given knowledge base using
the Backward Chaining strategy.

Algorithm:
1. Initialize an agenda stack to keep track of the goals to be achieved and a set
of inferred facts.
2. Start with the given goal and push it onto the agenda stack.
3. While the agenda stack is not empty:
   - Pop a goal from the agenda stack.
   - If the goal is already inferred, skip it.
   - Check if the goal is present in the knowledge base.
   - If the goal is present and all its supporting facts are already inferred:
     - Mark the goal as inferred and continue to process its supporting facts.
   - If the goal cannot be inferred:
     - Move to the next goal in the agenda stack.
4. Return True if the original goal is inferred, indicating that it can be derived
from the knowledge base; otherwise, return False.

Example:
- Define a knowledge base containing facts and rules.
- Specify the goal to be achieved.
- Call the backward_chaining function with the knowledge base and goal.
- The function attempts to infer the goal using Backward Chaining strategy and
returns True if successful; otherwise, returns False.

Note:
- Backward Chaining is a problem-solving strategy that starts with the goal to be
achieved and works backward, trying to find facts or rules that lead to the goal.
- It is commonly used in rule-based systems and expert systems to derive
conclusions from known facts and rules.
"""
def backward_chaining(knowledge_base, goal):
    agenda = [goal]
    inferred = set()

    while agenda:
        current_goal = agenda.pop()

        if current_goal in inferred:
            continue

        if current_goal in knowledge_base:
            supporting_facts = knowledge_base[current_goal]
            if all(fact in inferred for fact in supporting_facts):
                inferred.add(current_goal)
                if current_goal == goal:
                    return True
                agenda.extend(supporting_facts)

    return False

knowledge_base = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': ['F'],
    'F': []
}

goal = 'A'
result = backward_chaining(knowledge_base, goal)
```

```
if result:
    print(f"The goal '{goal}' can be inferred from the knowledge base using
Backward Chaining.")
else:
    print(f"The goal '{goal}' cannot be inferred from the knowledge base using
Backward Chaining.")
```

8. Implement resolution principle on FOPL related problems collab-link

```python
"""
Implementation of Resolution Principle for First-Order Predicate Logic (FOPL):

Classes:
- Predicate: Represents a predicate with a name and arguments.
- Clause: Represents a disjunction of predicates.

Functions:
- negate(predicate): Negates a given predicate.
- resolve(clause1, clause2): Resolves two clauses using the resolution principle.
- resolution(knowledge_base, query): Performs resolution on a knowledge base and a
query.

Algorithm:
1. Define a Predicate class to represent predicates with a name and arguments.
2. Define a Clause class to represent a disjunction of predicates.
3. Implement a negate function to negate a given predicate.
4. Implement a resolve function to resolve two clauses using the resolution
principle.
5. Implement a resolution function to perform resolution on a knowledge base and a
query:
   - Negate the query and add it to the knowledge base.
   - Repeat until no new clauses can be inferred or an empty clause is derived:
     - Generate new clauses by resolving every pair of clauses from the knowledge
base.
     - Add new clauses to the knowledge base.
   - If an empty clause is derived, return True (contradiction detected);
otherwise, return False.

Example Usage:
- Define predicates, clauses, knowledge base, and a query.
- Call the resolution function with the knowledge base and query.
- The function attempts to infer whether the query can be derived from the
knowledge base using the resolution principle.

Note:
- The resolution principle is a fundamental inference rule in logic programming,
used to derive new clauses from existing ones.
- It is a sound and complete method for refutation in first-order logic.
- The provided implementation demonstrates the application of resolution in
determining logical entailment in FOPL.
"""
class Predicate:
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __eq__(self, other):
        return self.name == other.name and self.args == other.args

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

    def __str__(self):
        return f"{self.name}({', '.join(self.args)})"

class Clause:
    def __init__(self, predicates):
        self.predicates = predicates

    def __eq__(self, other):
        return set(self.predicates) == set(other.predicates)

    def __hash__(self):
        return hash(frozenset(self.predicates))
    def __str__(self):
        return f"{' | '.join(str(pred) for pred in self.predicates)}"

def negate(predicate):
    return Predicate(f"~{predicate.name}", predicate.args)

def resolve(clause1, clause2):
    resolved = set()
    for pred1 in clause1.predicates:
        for pred2 in clause2.predicates:
            if pred1 == negate(pred2):
                resolved |= (set(clause1.predicates) - {pred1}) |
(set(clause2.predicates) - {pred2})
    return Clause(resolved)

def resolution(knowledge_base, query):
    kb = set(knowledge_base)
    negated_query = Clause([negate(predicate) for predicate in query.predicates])
    new_clause = Clause(kb | {negated_query})
    while True:
        new_clauses = set()
        for clause1 in kb:
            for clause2 in kb:
                if clause1 != clause2:
                    resolvent = resolve(clause1, clause2)
                    if not resolvent.predicates:
                        return True
                    new_clauses.add(resolvent)
        if new_clauses.issubset(kb):
            return False
        kb |= new_clauses

predicate1 = Predicate("P", ["x"])
predicate2 = Predicate("Q", ["x"])
predicate3 = Predicate("R", ["y"])
predicate4 = Predicate("P", ["y"])
clause1 = Clause([predicate1, predicate2])
clause2 = Clause([predicate3])
clause3 = Clause([predicate4])

knowledge_base = [clause1, clause2]
query = Clause([Predicate("Q", ["y"])])

result = resolution(knowledge_base, query)
print("Result:", result)
```

9. Implement Tic-Tac-Toe game using Python collab-link

```python
"""
Description:
This Python program allows two players to play the classic game of Tic-Tac-Toe. The
game is played on a 3x3 grid where players take turns marking their symbol ('X' or
'O') in an empty cell. The first player to align three of their symbols
horizontally, vertically, or diagonally wins the game. If all cells are filled and
no player has won, the game ends in a draw.

Functions:
1. print_board(board): Displays the current state of the game board.
2. check_winner(board, player): Checks if a player has won the game.
3. tic_tac_toe(): Main game loop where players take turns making moves until the
game ends.

Algorithm:
1. Initialize an empty 3x3 grid as the game board.
2. Alternately prompt each player to make a move by selecting a row and column.
3. After each move, check if the current player has won the game.
4. If a player wins or the game ends in a draw, display the final state of the
board and the result.
5. Allow players to play again if desired.

Usage:
- Run the program, and follow the prompts to enter row and column numbers to make
moves.
- The game will continue until one player wins or the game ends in a draw.
"""

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board, player):

    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or all([board[j][i] ==
player for j in range(3)]):
            return True

    if all([board[i][i] == player for i in range(3)]) or all([board[i][2-i] ==
player for i in range(3)]):
        return True
    return False

def tic_tac_toe():
    board = [[" "]*3 for _ in range(3)]
    current_player = "X"

    for _ in range(9):
        print_board(board)
        print(f"Player {current_player}'s turn:")
        while True:
            row = int(input("Enter row (0, 1, 2): "))
            col = int(input("Enter column (0, 1, 2): "))
            if board[row][col] == " ":
                break
            else:
                print("That position is already taken. Try again.")

        board[row][col] = current_player
        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            return
        current_player = "O" if current_player == "X" else "X"
    print_board(board)
    print("It's a draw!")

tic_tac_toe()
```

10. Build a bot which provides all the information related to text in search box

collab-link

```
import streamlit as st
import wikipediaapi

def search_wikipedia(query):
    wiki_en = wikipediaapi.Wikipedia(
        language="en",
        extract_format=wikipediaapi.ExtractFormat.WIKI,
        user_agent="MyCoolTool/1.0"
    )
    page = wiki_en.page(query)
    if page.exists():
        return page.summary
    else:
        return "No information found for the given query."

st.title("Wikipedia Bot")
query = st.text_input("Enter your search query:")

if st.button("Search"):
    result = search_wikipedia(query)
    st.info(result)


-- Using tinker

import tkinter as tk
from tkinter import messagebox
import wikipediaapi

def search_wikipedia(query):
    wiki_en = wikipediaapi.Wikipedia(
        language="en",
        extract_format=wikipediaapi.ExtractFormat.WIKI,
        user_agent="MyCoolTool/1.0"
    )
    page = wiki_en.page(query)
    if page.exists():
        return page.summary
    else:
        return "No information found for the given query."

def search():
    query = entry.get()
    result = search_wikipedia(query)
    messagebox.showinfo("Wikipedia Bot", result)

window = tk.Tk()
window.title("Wikipedia Bot")


entry = tk.Entry(window, width=50)
entry.pack(pady=10)
search_button = tk.Button(window, text="Search", command=search)
search_button.pack()

window.mainloop()
```