A* Search algorithm

```
import heapq

class Node: def init(self, state, parent=None, action=None, cost=0,
heuristic=0): self.state = state # Current state in the search space
self.parent = parent # Parent node self.action = action # Action that led to
this node from the parent node self.cost = cost # Cost to reach this node from
the start node self.heuristic = heuristic # Heuristic estimate of the cost to
reach the goal

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


def parse_graph_input(): graph = {} num_edges = int(input("Enter the number of
edges: ")) for _ in range(num_edges): u, v, cost = input("Enter an edge
(format: u v cost): ").split() cost = int(cost) if u not in graph: graph[u] =
[] if v not in graph: graph[v] = [] graph[u].append((v, cost))
graph[v].append((u, cost)) return graph

def astar_search(start_state, goal_test, successors, heuristic): # Priority
queue to store nodes ordered by f = g + h frontier = []
heapq.heappush(frontier, Node(start_state, None, None, 0,
heuristic(start_state))) explored = set()

while frontier:
    current_node = heapq.heappop(frontier)
    current_state = current_node.state

    if goal_test(current_state):
        # Reconstruct the path from the goal node to the start node
        path = []
        while current_node.parent is not None:
            path.append((current_node.action, current_node.state))
            current_node = current_node.parent
        path.reverse()
        return path

    explored.add(current_state)

    # Generate successors for the current state using the `successors` function
    for action, successor_state, step_cost in successors(current_state):
        if successor_state not in explored:
            new_cost = current_node.cost + step_cost
            new_node = Node(successor_state, current_node, action, new_cost,
heuristic(successor_state))
            heapq.heappush(frontier, new_node)
```

```
return None  # No path found
```

if **name** == "**main**": # Get user input to define the graph print("Define the graph:") graph = parse_graph_input()

```
start_state = input("Enter the start state: ")
goal_state = input("Enter the goal state: ")

def goal_test(state):
    return state == goal_state

def successors(state):
    # Generate successor states from the current state based on the graph
    successors_list = []
    for neighbor, cost in graph.get(state, []):
        action = f"Move to {neighbor}"  # Default action (e.g., "Move to B")
        successor_state = neighbor
        step_cost = cost
        successors_list.append((action, successor_state, step_cost))
    return successors_list

def heuristic(state):
    # Define a simple heuristic function (e.g., straight-line distance)
    heuristic_values = {key: abs(ord(key) - ord(goal_state)) for key in
graph.keys()}
    return heuristic_values.get(state, float('inf'))  # Default to infinity if
state not found

# Perform A* search using custom successors function
path = astar_search(start_state, goal_test, successors, heuristic)

# Print the resulting path found by A* search
if path:
    print("Path found:")
    for action, state in path:
        print(f"Action: {action}, State: {state}")
else:
    print("No path found.")
```

AO* Search algorithm

```
import heapq
```

```
class Node: def init(self, state, parent=None, action=None, cost=0,
heuristic=0): self.state = state # Current state in the search space
self.parent = parent # Parent node self.action = action # Action that led to
this node from the parent node self.cost = cost # Cost to reach this node from
the start node self.heuristic = heuristic # Heuristic estimate of the cost to
reach the goal

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


def parse_graph_input(): graph = {} num_edges = int(input("Enter the number of
edges: ")) for _ in range(num_edges): u, v, cost = input("Enter an edge
(format: u v cost): ").split() cost = float(cost) if u not in graph: graph[u] =
[] if v not in graph: graph[v] = [] graph[u].append((v, cost)) return graph

def ao_star_search(start_state, goal_state, graph): frontier = []
heapq.heappush(frontier, Node(start_state, None, None, 0,
heuristic(start_state, goal_state))) explored = {}

while frontier:
    current_node = heapq.heappop(frontier)
    current_state = current_node.state

    if current_state == goal_state:
        # Reconstruct the path from the goal node to the start node
        path = []
        while current_node.parent is not None:
            path.append((current_node.action, current_node.state))
            current_node = current_node.parent
        path.reverse()
        return path

    if current_state not in explored or current_node.cost <
explored[current_state]:
        explored[current_state] = current_node.cost

        for neighbor, step_cost in graph.get(current_state, []):
            new_cost = current_node.cost + step_cost
            new_node = Node(neighbor, current_node, f"Move to {neighbor}",
new_cost, heuristic(neighbor, goal_state))
            heapq.heappush(frontier, new_node)

return None  # No path found


def heuristic(state, goal_state): # Simple heuristic function (e.g., straight-
line distance) heuristic_values = {'A': 5, 'B': 3, 'C': 2, 'D': 1, 'E': 2, 'G':
0} # Custom heuristic values based on problem domain return
heuristic_values.get(state, float('inf')) # Default to infinity if state not
found
```

if **name** == "**main**": # Get user input to define the graph print("Define the graph:") graph = parse_graph_input()

```
start_state = input("Enter the start state: ")
goal_state = input("Enter the goal state: ")

# Perform AO* search using the defined graph, start state, and goal state
path = ao_star_search(start_state, goal_state, graph)

# Print the resulting path found by AO* search
if path:
    print("Path found:")
    for action, state in path:
        print(f"Action: {action}, State: {state}")
else:
    print("No path found.")
```

8-Queens Problem

```
def is_safe(board, row, col):
  """ Check if it's safe to place a queen at board[row][col] """
  # Check column
  for i in range(row):
    if board[i][col] == 1:
      return False

  # Check upper diagonal on left side
  i, j = row, col
  while i >= 0 and j >= 0:
    if board[i][j] == 1:
      return False
    i -= 1
    j -= 1

  # Check upper diagonal on right side
  i, j = row, col
  while i >= 0 and j < len(board):
    if board[i][j] == 1:
      return False
    i -= 1
    j += 1
```

```
    return True

def solve_queens(board, row):
    """ Recursively solve the 8-Queens Problem using backtracking """
    n = len(board)

    # Base case: If all queens are placed, return True
    if row >= n:
        return True

    for col in range(n):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place the queen

            # Recur to place the rest of the queens
            if solve_queens(board, row + 1):
                return True

            # If placing queen at board[row][col] doesn't lead to a solution, backtrack
            board[row][col] = 0  # Backtrack

    return False

def print_board(board):
    """ Print the board configuration """
    n = len(board)
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=" ")
        print()

def solve_8queens():
    """ Solve the 8-Queens Problem and print the solution """
    n = 8  # Size of the chessboard (8x8)
    board = [[0] * n for _ in range(n)]  # Initialize empty board
```

```
    if solve_queens(board, 0):
        print("Solution found:")
        print_board(board)
    else:
        print("No solution exists.")

# Call the function to solve the 8-Queens Problem
solve_8queens()
```

TSP using heuristic approach

```
import sys

def nearest_neighbor_tsp(distances): num_cities = len(distances)

# Start from the first city (arbitrary choice)
tour = [0]  # Store the tour as a list of city indices
visited = set([0])  # Track visited cities

current_city = 0
total_distance = 0

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

    # Find the nearest unvisited city
    for next_city in range(num_cities):
        if next_city not in visited and distances[current_city][next_city] <
min_distance:
            nearest_city = next_city
            min_distance = distances[current_city][next_city]

    # Move to the nearest city
    tour.append(nearest_city)
    visited.add(nearest_city)
    total_distance += min_distance
    current_city = nearest_city

# Complete the tour by returning to the starting city
tour.append(0)
total_distance += distances[current_city][0]

return tour, total_distance
```

Example usage:

if **name** == "**main**": # Example distance matrix (symmetric, square matrix)

```
distances = [[ 0,  4,  8,  9, 12], [ 4,  0,  6,  8,  9], [ 8,  6,  0, 10, 11], [ 9,
8, 10,  0,  7], [12,  9, 11,  7,  0]]
# Run nearest neighbor TSP algorithm
tour, total_distance = nearest_neighbor_tsp(distances)

# Print the tour and total distance
print("Nearest Neighbor TSP Tour:", tour)
print("Total Distance:", total_distance)
```

Forward Chaining and Backward Chaining.

def forward_chaining(rules, facts, goal): inferred_facts = set(facts) new_facts = True

```
while new_facts:
    new_facts = False

    for rule in rules:
        condition, result = rule

        if all(cond in inferred_facts for cond in condition) and result not in
inferred_facts:
            inferred_facts.add(result)
            new_facts = True

            if result == goal:
                return True

return False
```

def backward_chaining(rules, facts, goal): def ask(query): if query in facts: return True

```
   for rule in rules:
        condition, result = rule
        if result == query and all(ask(cond) for cond in condition):
            return True

    return False

return ask(goal)
```

rules = [(['hair', 'live young'], 'mammal'), (['feathers', 'fly'], 'bird')]

facts = ['hair', 'live young'] goal = 'mammal'

is_mammal = forward_chaining(rules, facts, goal)

if is_mammal: print("The cat is classified as a mammal.") else: print("The cat is not classified as a mammal.")

facts = ['feathers', 'fly'] goal = 'bird'

is_bird = backward_chaining(rules, facts, goal)

if is_bird: print("The pigeon is classified as a bird.")

else: print("The pigeon is not classified as a bird.")

Resolution principle on First-Order Predicate Logic

```
def negate_literal(literal): """ Negate a literal by adding or removing the
negation symbol '' """ if literal.startswith(''): return literal[1:] # Remove
negation else: return '~' + literal # Add negation

def resolve(clause1, clause2): """ Resolve two clauses to derive a new clause
""" new_clause = [] resolved = False

# Copy literals from both clauses
for literal in clause1:
    if negate_literal(literal) in clause2:
        resolved = True
    else:
        new_clause.append(literal)

for literal in clause2:
    if negate_literal(literal) not in clause1:
        new_clause.append(literal)

if resolved:
    return new_clause
else:
    return None  # No resolution possible


def resolution(propositional_kb, query): """ Use resolution to prove or
disprove a query using propositional logic """ kb = propositional_kb[:]
kb.append(negate_literal(query)) # Add negated query to knowledge base

while True:
    new_clauses = []
    n = len(kb)
    resolved_pairs = set()  # Track resolved pairs to avoid redundant
```

```
resolutions

    for i in range(n):
        for j in range(i + 1, n):
            clause1 = kb[i]
            clause2 = kb[j]

            if (clause1, clause2) not in resolved_pairs:
                resolved_pairs.add((clause1, clause2))
                resolvent = resolve(clause1, clause2)

                if resolvent is None:
                    continue  # No resolution possible for these clauses

                if len(resolvent) == 0:
                    return True  # Empty clause (contradiction), query is
proved

                if resolvent not in new_clauses:
                    new_clauses.append(resolvent)

    if all(clause in kb for clause in new_clauses):
        return False  # No new clauses added, query cannot be proven

    kb.extend(new_clauses)  # Add new clauses to the knowledge base
```

Example usage:

if **name** == "**main**": # Example propositional knowledge base (list of clauses)
propositional_kb = [['~~P', 'Q'], ['P', '~~Q', 'R'], ['~R', 'S']]

```
# Example query to prove/disprove using resolution
query = 'S'

# Use resolution to prove or disprove the query
result = resolution(propositional_kb, query)

if result:
    print(f"The query '{query}' is PROVED.")
else:
    print(f"The query '{query}' is DISPROVED.")
```

Tic-Tac-Toe game

def print_board(board): """ Print the current state of the Tic-Tac-Toe board """ for row in board: print(" | ".join(row)) print("-" * 9)

def check_winner(board, player): """ Check if the specified player has won the game """ for row in board: if all(cell == player for cell in row): return True for col in range(3): if all(board[row][col] == player for row in range(3)): return True if all(board[i][i] == player for i in range(3)): return True if all(board[i][2-i] == player for i in range(3)): return True return False

def is_full(board): """ Check if the board is completely filled """ return all(cell != ' ' for row in board for cell in row)

def tic_tac_toe(): """ Main function to run the Tic-Tac-Toe game """ board = [[' ' for _ in range(3)] for _ in range(3)] current_player = 'X'

```
while True:
    print_board(board)
    print(f"Player {current_player}'s turn.")
    row = int(input("Enter row (1-3): "))
    col = int(input("Enter column (1-3): "))
    row -= 1
    col -= 1

    if board[row][col] == ' ':
        board[row][col] = current_player
    else:
        print("Invalid move! Try again.")
        continue

    # Check if the current player has won
    if check_winner(board, current_player):
        print_board(board)
        print(f"Player {current_player} wins!")
        break

    # Check if the board is full (tie game)
    if is_full(board):
        print_board(board)
        print("It's a tie!")
        break

    # Switch to the other player
    current_player = 'O' if current_player == 'X' else 'X'
```

if **name** == "**main**": tic_tac_toe()