Exp. 3:

Implementation of BFS

Code:

graph = {

    'A': ['B', 'C', 'D'],

    'B': ['A'],

    'C': ['A', 'D'],

    'D': ['A', 'C', 'E'],

    'E': ['D']

}

*def* BFS(*node*):

    visited = [False]\*len(graph)

    queue = []

    visited.append(*node*)

    queue.append(*node*)

    print("The traversal path is: ")

    while queue:

        v = queue.pop(0)

        print(v, *end*=" ")

        for neigh in graph[v]:

            # print("\n Neighbour : " + neigh)

            if neigh not in visited:

                # print("append : ", neigh)

                visited.append(neigh)

                queue.append(neigh)

BFS("A")

Exp. 4: Implementation of DFS/ Depth Limited Search

Code:

DFS

graph = {

    'A': ['B', 'C', 'F'],

    'B': ['A'],

    'C': ['A', 'D'],

    'D': ['A', 'C', 'E'],

    'E': ['D'],

    'F': []

}

visited = set()

*def* dfs(*visited*, *graph*, *node*):

    if *node* not in *visited*:

        print(*node*)

*visited*.add(*node*)

        for neighbour in *graph*[*node*]:

            dfs(*visited*, *graph*, neighbour)

print("Following is the Depth-First Search")

dfs(visited, graph, 'A')

DLS

graph = {

    'A': ['B', 'C', 'D'],

    'B': ['E', 'F'],

    'C': ['G'],

    'D': ['H', 'I'],

    'E': ['J', 'K'],

    'F': ['L'],

    'G': [],

    'H': [],

    'I': [],

    'J': [],

    'K': [],

    'L': []

}

*def* dls(*graph*, *node*, *goal*, *depth\_limit*, *current\_depth*=0):

    if *current\_depth* > *depth\_limit*:

        return False

    print(*node*, *end*="    ")

    if *node* == *goal*:

        return True

    for neighbour in *graph*[*node*]:

        if dls(*graph*, neighbour, *goal*, *depth\_limit*, *current\_depth* + 1):

            return True

    return False

limit = 2

goal\_node = 'I'

print("DLS With depth limit {}:".format(limit))

found = dls(graph, 'A', goal\_node, limit)

if found:

    print("\nGoal node '{}' found within depth limit.".format(goal\_node))

else:

    print("\nGoal node '{}' not found within depth limit.".format(goal\_node))

Exp. 5: Implement A\* Search Algorithm

Code:

import heapq

# Define the graph as an adjacency list with weighted edges

graph = {

    'A': [('B', 4), ('C', 3), ('D', 5)],

    'B': [('E', 12)],

    'C': [('D', 10)],

    'D': [('G', 16)],

    'E': [('F', 7)],

    'F': [('G', 2)],

    'G': []

}

# Define heuristic values for each node

heuristic = {

    'A': 14,

    'B': 12,

    'C': 11,

    'D': 11,

    'E': 6,

    'F': 4,

    'G': 2

}

*def* astar(*graph*, *start*, *goal*, *heuristic*):

    open\_set = [(0, *start*)]  # Priority queue to store nodes to be explored

    came\_from = {}  # Dictionary to store the parent node for each node

    g\_score = {node: float('inf') for node in *graph*}

    g\_score[*start*] = 0

    while open\_set:

        current\_cost, current\_node = heapq.heappop(open\_set)

        if current\_node == *goal*:

            path = []

            while current\_node in came\_from:

                path.insert(0, current\_node)

                current\_node = came\_from[current\_node]

            path.insert(0, *start*)

            return path

        for neighbor, weight in *graph*[current\_node]:

            tentative\_g\_score = g\_score[current\_node] + weight

            if tentative\_g\_score < g\_score[neighbor]:

                came\_from[neighbor] = current\_node

                g\_score[neighbor] = tentative\_g\_score

                f\_score = tentative\_g\_score + *heuristic*[neighbor]

                heapq.heappush(open\_set, (f\_score, neighbor))

    return None  # No path found

# Define the start and goal nodes

start\_node = 'A'

goal\_node = 'G'

# Find the path using A\* algorithm

path = astar(graph, start\_node, goal\_node, heuristic)

if path:

    print("Shortest path from {} to {}:".format(start\_node, goal\_node))

    for node in path:

        print(node)

else:

    print("No path found from {} to {}.".format(start\_node, goal\_node))

Exp. 7: Implement knowledgebase in prolog:

Code:

% Facts: Define family relationships

parent(john, mary).

parent(john, lisa).

parent(ellen, mary).

parent(ellen, lisa).

parent(mary, ann).

parent(mary, dave).

% Rules: Define additional relationships

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

% Queries can be used to ask questions about the knowledge base

% For example, "Is John a parent of Mary?"

% Query: parent(john, mary).

% This will return "true."

% To find all children of a parent, you can use a query like

% Query: parent(john, X).

% This will return all children of John, which are Mary and Lisa in this case.

Exp. 8: Implementation of unification algorithm in prolog:

Code:

% Define a few predicates for demonstration

is\_human(socrates).

is\_mortal(X) :- is\_human(X).

is\_philosopher(plato).

% Query 1: Unification of a variable with a constant

%?- is\_mortal(socrates). % This succeeds because socrates is a mortal according to the knowledge base.

% Query 2: Unification of a variable with a rule

%?- is\_mortal(X). % This returns X = socrates, showing that X is unified with socrates.

% Query 3: Unification of two constants

%?- is\_human(socrates) = is\_human(plato). % This fails because the constants socrates and plato do not unify.

% Query 4: Unification with variables

%?- is\_mortal(X) = is\_human(socrates). % This succeeds and unifies X with socrates.

% Query 5: Unification with variables and rules

%?- is\_mortal(X) = is\_human(plato). % This fails because plato is not explicitly defined as a mortal in the knowledge base.

Exp. 9: Implementation of Bayesian Belief Network

Code:

# Define the conditional probability tables (CPTs)

#     'R\_t\_C\_t': {

#         'True': 0.96,

#         'False': 0.04

#     },

#     'R\_t\_C\_f': {

#         'True': 0.85,

#         'False': 0.15

#     },

#     'R\_f\_C\_t': {

#         'True': 0.7,

#         'False0': 0.3

#     },

#     'R\_f\_C\_f': {

#         'True': 0.08,

#         'False': 0.92

#     }

# }

cpt\_cold\_given\_runny\_nose = {'True': 0.85, 'False': 0.15}

cpt\_cold\_given\_cough = {'True': 0.7, 'False': 0.3}

cpt\_runny\_nose = {'True': 0.2, 'False': 0.8}

cpt\_cough = {'True': 0.3, 'False': 0.7}

# symtoms input:

a = input("Do u have runny nose(True/False):")

b = input("Do u have caugh(True/False):")

# Define the evidence (observed symptoms)

evidence = {

    'RunnyNose': a,

    'Cough': b,

    # 'Cold': 'True'

}

# Calculate joint probability

joint\_probability = (

    cpt\_cold\_given\_runny\_nose[evidence['RunnyNose']] \*

    cpt\_cold\_given\_cough[evidence['Cough']] \*

    cpt\_runny\_nose[evidence['RunnyNose']] \*

    cpt\_cough[evidence['Cough']]

)

print("cpt\_cold\_given\_runny\_nose \* cpt\_cold\_given\_cough \* cpt\_runny\_nose \* cpt\_cough")

print(*f*"{cpt\_cold\_given\_runny\_nose[evidence['RunnyNose']]}\*", *end*="  ")

print(*f*"{cpt\_cold\_given\_cough[evidence['Cough']]}\*", *end*="  ")

print(*f*"{cpt\_runny\_nose[evidence['RunnyNose']]}\*", *end*="  ")

print(*f*"{cpt\_cough[evidence['Cough']]}\*")

# Query the network to find the probability of having a cold given the symptoms

probability\_cold = joint\_probability

print(*f*"Probability of having a cold: {probability\_cold*:.2f*}")