**Uniformed search**

from collections import deque

def bfs(adjList, startNode, visited):

q = deque()

visited[startNode] = True

q.append(startNode)

while q:

currentNode = q.popleft()

print(currentNode, end=" ")

for neighbor in adjList[currentNode]:

if not visited[neighbor]:

visited[neighbor] = True

q.append(neighbor)

def addEdge(adjList, u, v):

adjList[u].append(v)

def main():

vertices = 5

adjList = [[] for \_ in range(vertices)]

addEdge(adjList, 0, 1)

addEdge(adjList, 0, 2)

addEdge(adjList, 1, 3)

addEdge(adjList, 1, 4)

addEdge(adjList, 2, 4)

visited = [False] \* vertices

print("Breadth First Traversal starting from vertex 0:", end=" ")

bfs(adjList, 0, visited)

print('\n')

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

main()

**Informed search**

import heapq

def heuristic(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def greedy\_best\_first\_search(maze, start, end):

open\_list = []

came\_from = {}

visited = set()

heapq.heappush(open\_list, (heuristic(start, end), start))

came\_from[start] = None

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

while open\_list:

current\_heuristic, current = heapq.heappop(open\_list)

if current == end:

path = []

while current:

path.append(current)

current = came\_from[current]

return path[::-1]

visited.add(current)

for direction in directions:

neighbor = (current[0] + direction[0], current[1] + direction[1])

if (0 <= neighbor[0] < len(maze)) and (0 <= neighbor[1] < len(maze[0])):

if maze[neighbor[0]][neighbor[1]] == 0 and neighbor not in visited:

visited.add(neighbor)

came\_from[neighbor] = current

heapq.heappush(open\_list, (heuristic(neighbor, end), neighbor))

return None

maze = [

[0, 0, 0, 0, 1],

[0, 1, 1, 0, 1],

[0, 0, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 0, 0]

]

start = (0, 0)

end = (4, 4)

path = greedy\_best\_first\_search(maze, start, end)

print("Path from start to end:", path)

**Local Search**

def findLocalMaxima(n, arr):

mx = []

if arr[0] > arr[1]:

mx.append(0)

for i in range(1, n - 1):

if arr[i - 1] < arr[i] > arr[i + 1]:

mx.append(i)

if arr[-1] > arr[-2]:

mx.append(n - 1)

if len(mx) > 0:

print("Points of Local maxima are: ", end="")

print(\*mx)

else:

print("There are no points of Local maxima.")

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

n = 9

arr = [10, 10, 15, 14, 13, 25, 5, 4, 3]

findLocalMaxima(n, arr)

**Prolog Monkey**

% Clauses:

in\_room(bananas).

in\_room(chair).

in\_room(monkey).

clever(monkey).

can\_climb(monkey, chair).

tall(chair).

can\_move(monkey, chair, bananas).

can\_reach(X, Y):-clever(X).

get\_on(X,Y):-

can\_climb(X,Y).

under(Y,Z):-

in\_room(X),in\_room(Y),

in\_room(Z),can\_climb(X,Y,Z).

clever(X,Z):-

get\_on(X,Y), under(Y,Z);

tall(Y).

*Questions:*

*can\_reach(A, B).*

*can\_reach(monkey, banana).*

*can\_reach(banana, monkey).*

**Bayesian Network**

import numpy as np

import pandas as pd

from pgmpy.models import BayesianModel

from pgmpy.inference import VariableElimination

from pgmpy.inference import BeliefPropagation

from pgmpy.factors.discrete import TabularCPD

model = BayesianModel([('Weather', 'Umbrella')])

cpd\_weather = TabularCPD(variable='Weather',

variable\_card=2,

values=[[0.7], [0.3]])

cpd\_umbrella = TabularCPD(variable='Umbrella',

variable\_card=2,

values=[[0.9, 0.2],

[0.1, 0.8]],

evidence=['Weather'],

evidence\_card=[2])

model.add\_cpds(cpd\_weather, cpd\_umbrella)

assert model.check\_model()

inference = VariableElimination(model)

query\_result = inference.query(variables=['Umbrella'], evidence={'Weather': 1})

print("P(Umbrella | Weather=Rainy):")

print(query\_result)

query\_result = inference.query(variables=['Weather'], evidence={'Umbrella': 1})

print("\nP(Weather=Sunny | Umbrella=Yes):")

print(query\_result)

query\_result = inference.query(variables=['Umbrella'], evidence={'Weather': 0})

print("\nP(Umbrella | Weather=Sunny):")

print(query\_result)

**Prolog 8 puzzle**

% Simple Prolog Planner for the 8 Puzzle Problem

/\* This predicate initialises the problem states. The first argument of solve is the initial state, the 2nd the goal state, and the third the plan that will be produced.\*/

test(Plan):-

write('Initial state:'),nl,

Init= [at(tile4,1), at(tile3,2), at(tile8,3), at(empty,4), at(tile2,5), at(tile6,6), at(tile5,7), at(tile1,8), at(tile7,9)],

write\_sol(Init),

Goal= [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7), at(tile7,8), at(tile8,9)],

nl,write('Goal state:'),nl,

write(Goal),nl,nl,

solve(Init,Goal,Plan).

solve(State, Goal, Plan):-

solve(State, Goal, [], Plan).

% Determines whether Current and Destination tiles are a valid move.

is\_movable(X1,Y1) :- (1 is X1 - Y1) ; (-1 is X1 - Y1) ; (3 is X1 - Y1) ; (-3 is X1 - Y1).

/\* This predicate produces the plan. Once the Goal list is a subset of the current State the plan is complete and it is written to the screen using write\_sol \*/

solve(State, Goal, Plan, Plan):-

is\_subset(Goal, State), nl,

write\_sol(Plan).

solve(State, Goal, Sofar, Plan):-

act(Action, Preconditions, Delete, Add),

is\_subset(Preconditions, State),

\+ member(Action, Sofar),

delete\_list(Delete, State, Remainder),

append(Add, Remainder, NewState),

solve(NewState, Goal, [Action|Sofar], Plan).

/\* The problem has three operators.

1st arg = name

2nd arg = preconditions

3rd arg = delete list

4th arg = add list. \*/

% Tile can move to new position only if the destination tile is empty & Manhattan distance = 1

act(move(X,Y,Z),

[at(X,Y), at(empty,Z), is\_movable(Y,Z)],

[at(X,Y), at(empty,Z)],

[at(X,Z), at(empty,Y)]).

% Utility predicates.

% Check is first list is a subset of the second

is\_subset([H|T], Set):-

member(H, Set),

is\_subset(T, Set).

is\_subset([], \_).

% Remove all elements of 1st list from second to create third.

delete\_list([H|T], Curstate, Newstate):-

remove(H, Curstate, Remainder),

delete\_list(T, Remainder, Newstate).

delete\_list([], Curstate, Curstate).

remove(X, [X|T], T).

remove(X, [H|T], [H|R]):-

remove(X, T, R).

write\_sol([]).

write\_sol([H|T]):-

write\_sol(T),

write(H), nl.

append([H|T], L1, [H|L2]):-

append(T, L1, L2).

append([], L, L).

member(X, [X|\_]).

member(X, [\_|T]):-

member(X, T).

*Questions:*

*test(Plan).*