Uniformed code

```
from collections import deque


def bfs(adjList, startNode, visited):
    """
    Performs a breadth-first search on the graph starting from the given node.


    Args:
    adjList (list): The adjacency list representation of the graph.
    startNode (int): The node to start the search from.
    visited (list): A list to keep track of visited nodes.
    """
    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):
    """
    Adds an edge to the graph.


    Args:
    adjList (list): The adjacency list representation of the graph.
```

```
    u (int): The source node of the edge.
    v (int): The destination node of the edge.
    """
    adjList[u].append(v)

def main():
    """
    The main function.
    """
    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)

if _name_ == "_main_":
    main()
```

Informed code

```
mport heapq


def heuristic(a, b):
    """Calculate the Manhattan distance from a to b"""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


def greedy_best_first_search(maze, start, end):
    """Perform Greedy Best-First Search in a grid maze"""
    open_list = []
    came_from = {}  # Tracks path history
    visited = set()  # Keeps track of visited nodes to prevent revisiting

    heapq.heappush(open_list, (heuristic(start, end), start))
    came_from[start] = None  # Start has no parent

    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

    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)

        # Explore neighbors
```

```
        for direction in directions:
            neighbor = (current[0] + direction[0], current[1] + direction[1])

            # Ensure the neighbor is within the bounds of the maze
            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  # If no path is found

# Example maze: 0 - walkable, 1 - blocked
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)
```

Bayesian code

```
from pgmpy.models import BayesianNetwork
from pgmpy.inference import VariableElimination
from pgmpy.factors.discrete import TabularCPD

# Step 1: Define the structure of the BBN
model = BayesianNetwork([('Disease_A', 'Symptom_X'),
                ('Disease_B', 'Symptom_X')])

# Step 2: Define the Conditional Probability Distributions (CPDs)

# P(Disease A)
cpd_disease_a = TabularCPD(variable='Disease_A',
                variable_card=2,  # 0: No Disease A, 1: Disease A
                values=[[0.9], [0.1]])  # P(Disease A)

# P(Disease B)
cpd_disease_b = TabularCPD(variable='Disease_B',
                variable_card=2,  # 0: No Disease B, 1: Disease B
                values=[[0.95], [0.05]])  # P(Disease B)

# P(Symptom X | Disease A, Disease B)
cpd_symptom_x = TabularCPD(variable='Symptom_X',
                variable_card=2,  # 0: No Symptom X, 1: Symptom X
                values=[[0.01, 0.6, 0.6, 0.8],  # P(Symptom X=0 | ...)
                    [0.99, 0.4, 0.4, 0.2]],  # P(Symptom X=1 | ...)
                evidence=['Disease_A', 'Disease_B'],
                evidence_card=[2, 2])  # 2 states for each disease

# Step 3: Add CPDs to the model
```

```python
model.add_cpds(cpd_disease_a, cpd_disease_b, cpd_symptom_x)

# Step 4: Verify the model
assert model.check_model()

# Step 5: Perform inference
inference = VariableElimination(model)

# Query: What is the probability of Disease A given Symptom X?
query_result = inference.query(variables=['Disease_A'],
                    evidence={'Symptom_X': 1})

print(query_result)
```

Prolog monkey banana

```
% Define the initial state
initial_state(state(monkey, box, banana, on_ground)).

% Define actions
action(move(Location1, Location2), state(Monkey, Box, Banana, Location1), state(Monkey, Box, Banana, Location2)).
action(climb(Box), state(Monkey, Box, Banana, on_box), state(Monkey, Box, Banana, on_ceiling)).
action(grab(Monkey), state(Monkey, Box, Banana, on_ceiling), state(Monkey, Box, banana_grabbed, on_ceiling)).

% Define conditions for actions
can_move(Location1, Location2) :- Location1 \= Location2.
can_climb(Box) :- Box \= none.
can_grab(Banana) :- Banana = banana.

% Define the goal
goal(state(Monkey, Box, banana_grabbed, _)).

% Define the plan
plan(State, []) :- goal(State).
plan(State, [Action | Rest]) :-
   action(Action, State, NewState),
   plan(NewState, Rest).

% Run the simulation
run :-
   initial_state(InitialState),
   plan(InitialState, Actions),
   write('Actions to achieve goal: '), nl,
   write(Actions).
```

```prolog
% Helper predicate to display the state
display_state(State) :-
    write('Current state: '), write(State), nl.
```

Prolog 8 puzzle

```
% Define the initial state
initial_state([[2, 8, 1], [4, 3, 6], [7, 5, 0]]).

% Define the goal state
goal_state([[1, 2, 3], [4, 5, 6], [7, 8, 0]]).

% Define the possible moves
move(up, State, NewState) :-
   State = [A, B, C],
   nth0(1, A, 0),
   append([B, [0|T]], C, NewState),
   append([T, 0], A, B).
move(down, State, NewState) :-
   State = [A, B, C],
   nth0(1, C, 0),
   append([A, [0|T]], B, NewState),
   append([T, 0], C, B).
move(left, State, NewState) :-
   State = [A, B, C],
   nth0(0, A, 0),
   append([[0|T], A2], [B, C], NewState),
   append([T, 0], A2, A).
move(right, State, NewState) :-
   State = [A, B, C],
   nth0(2, A, 0),
   append([[T, 0], A2], [B, C], NewState),
   append(A2, [0, T], A).
```

```
% Define the heuristic function (Manhattan distance)
heuristic(State, H) :-
   goal_state(Goal),
   H is manhattan_distance(State, Goal).

manhattan_distance(State, Goal) :-
   findall(D, (nth0(I, State, Row), nth0(J, Row, X), nth0(I, Goal, GRow), nth0(J, GRow, Y), D is abs(I - I) +
abs(J - J) + abs(X - Y)), Ds),
   sumlist(Ds, Sum),
   Sum is Manhattan.

% Define the A* search algorithm
a_star_search(InitialState, GoalState, Path) :-
   a_star_search(InitialState, GoalState, [], Path).

a_star_search(State, GoalState, Visited, [State|Path]) :-
   heuristic(State, H),
   a_star_search(State, GoalState, Visited, Path, H).

a_star_search(State, GoalState, Visited, Path, H) :-
   move(Move, State, NewState),
   \+ member(NewState, Visited),
   heuristic(NewState, NewH),
   NewH < H,
   a_star_search(NewState, GoalState, [State|Visited], Path, NewH).

a_star_search(State, GoalState, Visited, Path, _) :-
   State = GoalState,
   reverse([State|Visited], Path).

% Run the A* search algorithm
```

```
run :-
    initial_state(InitialState),
    goal_state(GoalState),
    a_star_search(InitialState, GoalState, Path),
    write('Path to goal: '), nl,
    write(Path).

% Helper predicate to find the nth element in a list
nth0(0, [H|_], H).
nth0(N, [_|T], E) :-
    N > 0,
    N1 is N - 1,
    nth0(N1, T, E).

% Helper predicate to sum a list of numbers
sumlist([], 0).
sumlist([H|T], Sum) :-
    sumlist(T, Sum1),
    Sum is Sum1 + H.
```