COLLEGE OF INFORMATION AND COMMUNICATION TECHNOLOGIES(CoICT)

IS 365

ARTIFICIAL INTELLIGENCE

GROUP ASSIGNEMENT

STUDENT NAME	**REGISTRATION NO**	**CONTRIBUTION**
NDAGULA, FRANK JONAS	2021-04-09387	Predicate logic, Rule based system, Probabilistic systems
KARATA, HERI MUSSA	2021-04-03523	Breadth First Search, Depth First Search
KANOKOLEKE,REHEMA BAKARI	2019-04-03560	Uniform Cost Search, Propositional Logic
MCHOMVU,SALIM I	2021-04-06692	Hill climbing, Dynamic programming
MGAYA, DENIS PASCAL	2021-04-06875	Heuristic Search, A * Search

1. **How different search algorithms are used search problem solving**

   a) **Breadth First Search (BFS) (HERI MUSSA KARATA)**

   Breadth-First Search is a fundamental search algorithm used in computer science for traversing or searching tree or graph data structures.

   i) Description

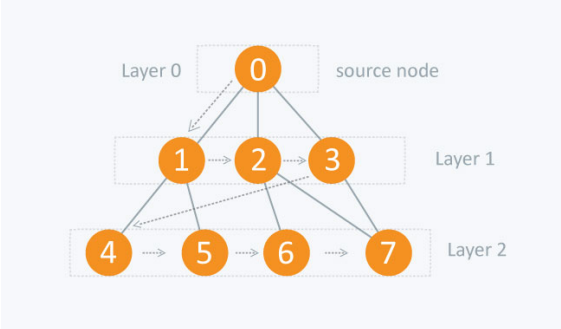
   BFS explores all the nodes at the present depth level before moving on to nodes at the next depth level. It uses a queue data structure to keep track of the nodes to be explored next. The algorithm starts at the root (or an arbitrary node in the case of a graph) and explores the neighbour nodes at the present depth prior to moving on to nodes at the next depth level.

   ii) How it works:

   BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

   As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

   1) First move horizontally and visit all the nodes of the current layer
   2) Move to the next layer



iii) Algorithm Pseudocode:

```
function BFS (graph, start_node):

# Create a queue for BFS and enqueue the start_node

queue = new Queue ()
queue.enqueue(start_node)

# Create a set to keep track of visited nodes

visited = set ()
visited.add(start_node)

# While there are nodes to process in the queue

while not queue.is_empty():

# Dequeue a node from the front of the queue

current_node = queue.dequeue()

# Process the current node
print(current_node)

# Get all adjacent nodes of the dequeued node
for neighbour in graph[current_node]:

# If an adjacent node has not been visited
if neighbour not in visited:
```

```
        # Enqueue the neighbour
        queue.enqueue(neighbour)

        # Mark the neighbour as visited
        visited.add(neighbour)
```

iv) Applications of Breadth First Search:

- Copying garbage collection, Cheney's algorithm.
- Finding the shortest path between two nodes u and v, with path length measured by the total number of edges (an advantage over depth–first search).
- Testing a graph for bipartiteness.
- Minimum Spanning Tree for an unweighted graph.
- Web crawler.
- Finding nodes in any connected component of a graph.
- Ford–Fulkerson method for computing the maximum flow in a flow network.
- Serialization/Deserialization of a binary tree vs. serialization in sorted order allows the tree to be reconstructed efficiently.

v) Example:

Chess Knight problem: Given a chessboard, find the shortest distance (minimum number of steps) taken by a knight to reach a given destination from a given source.

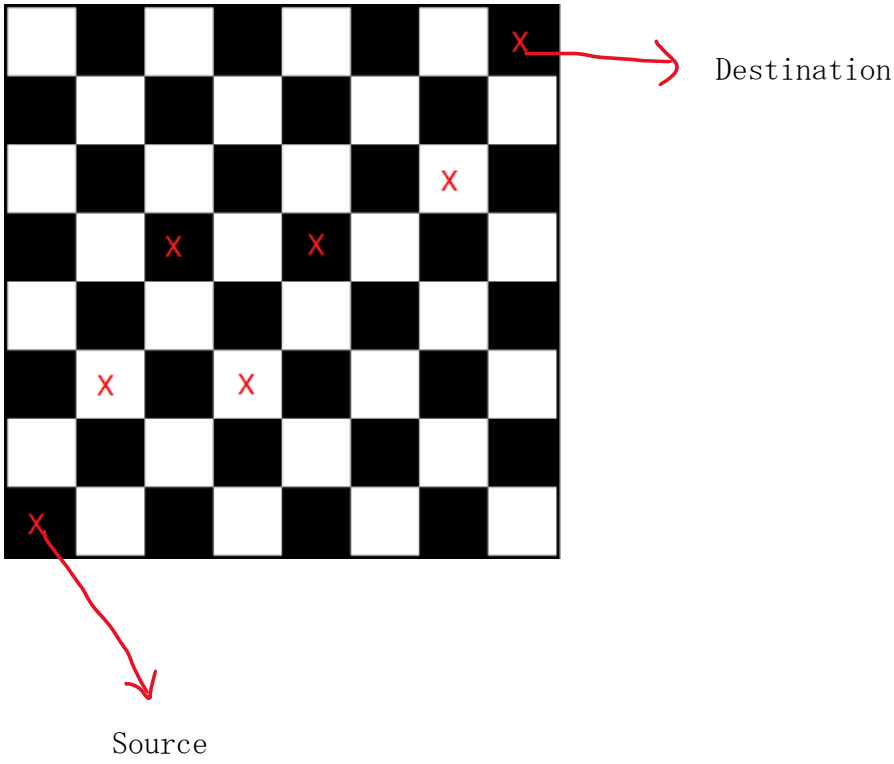
Consider:

8 by 8-piece board = 64 total pieces,

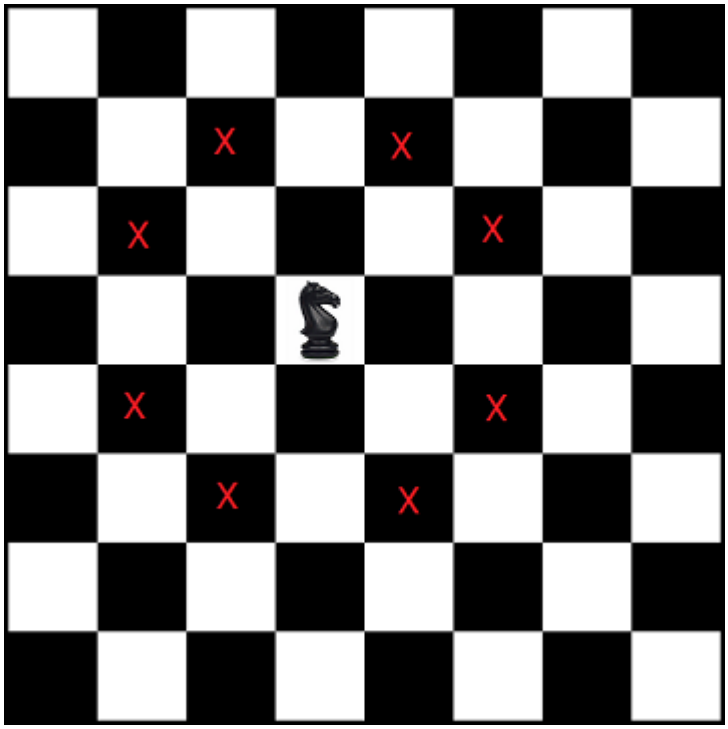
Source = (0, 7)

Destination = (7, 0)

The knight's movement is illustrated in the following figure:



A knight can move in eight possible directions from a given cell, as illustrated in the following figure:



We can find all the possible locations the knight can move to from the given location by using the array that stores the relative position of knight movement

from any location. For example, if the current location is (x, y), we can move to `(x + row[k], y + col[k]) for 0 <= k <= 7` using the following array:

```
row[] = [ 2, 2, -2, -2, 1, 1, -1, -1 ]

col[] = [ -1, 1, 1, -1, 2, -2, 2, -2 ]
```

So, from `position (x, y)` knight's can move to:

```
(x + 2, y - 1)

(x + 2, y + 1)

(x - 2, y + 1)

(x - 2, y - 1)

(x + 1, y + 2)

(x + 1, y - 2)

(x - 1, y + 2)

(x - 1, y - 2)
```

Note that in BFS, all cells having the shortest path as 1 are visited first, followed by their adjacent cells having the shortest path as 1 + 1 = 2 and so on… so if we reach any node in BFS, its shortest path = shortest path of parent + 1. So, the destination cell's first occurrence gives us the result, and we can stop our search there. `The shortest path cannot exist from some other cell for which we haven't reached the given node yet. If any such path were possible, we would have already explored it.`

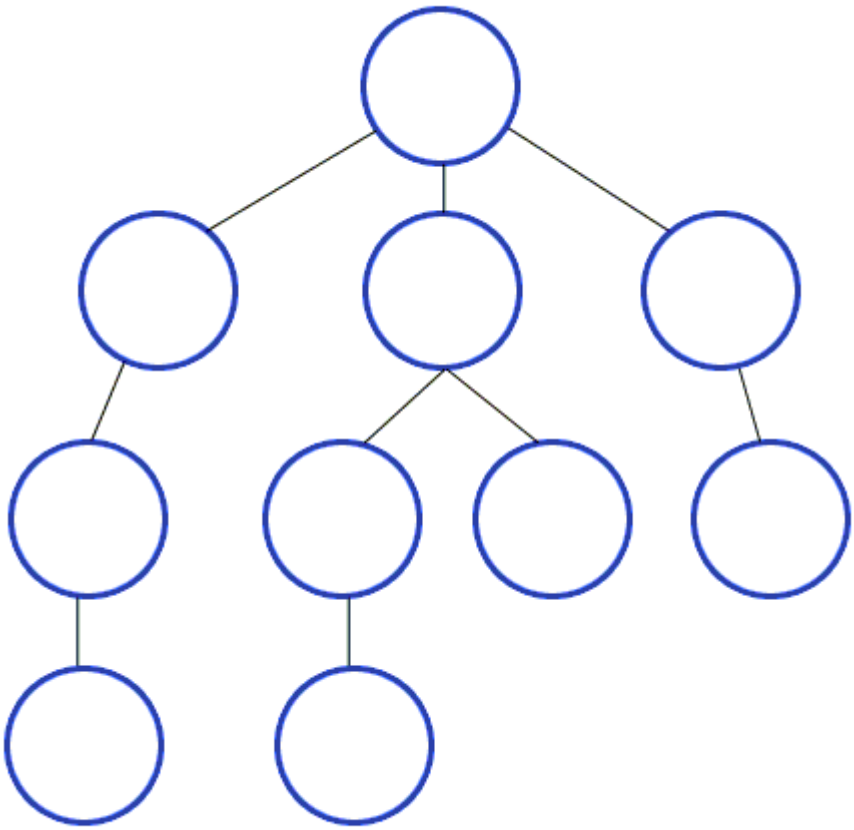
b) Depth First Search(HERI MUSSA KARATA)

i) Description

Depth-First Search is another fundamental search algorithm used in computer science for traversing or searching tree or graph data structures. Unlike Breadth-First Search (BFS), which explores nodes level by level, DFS explores as far down a branch as possible before backtracking.

ii) How it works:

The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.



iii) Pseudocode

```
DFS(graph, start_node):
// Mark the starting node as visited
visited = {}
visited[start_node] = true

// Visit the starting node
visit(start_node)

// Get all adjacent nodes of the starting node
adjacent_nodes = graph.get_adjacent_nodes(start_node)

// Loop through all the adjacent nodes
for next_node in adjacent_nodes:
// If the adjacent node is not visited, mark it as visited and perform
DFS on it
if next_node not in visited:
visited[next_node] = true
DFS(graph, next_node)
```

iv) Applications of Breadth First Search:

- Graph Traversal: DFS is primarily used for traversing graphs, visiting each vertex and edge in a systematic manner. It helps identify connected components, cycles, and paths between nodes.

- Maze Solving: DFS is employed to navigate through mazes, systematically exploring each pathway until reaching the exit. It ensures thorough exploration of all possible routes.

- Topological Sorting: DFS can perform topological sorting of directed acyclic graphs (DAGs), crucial in scheduling tasks with dependencies or resolving build dependencies.

- Detecting Cycles: DFS can detect cycles in a graph by identifying back edges during traversal. This is valuable for applications where cycles need to be avoided, such as in network routing or deadlock detection.

- Pathfinding: DFS can be used to find paths between nodes in a graph, including shortest paths if appropriate criteria are applied. This is useful in routing algorithms, puzzle-solving, and navigation systems.

- Solving Puzzles: DFS is employed in solving various puzzles and games, such as Sudoku, Minesweeper, and the Eight Queens problem. It systematically explores possible solutions until a valid one is found.

- Network Analysis: DFS can analyze networks, such as social networks or computer networks, by exploring connections between nodes and identifying patterns or clusters within the network.

- Generating Mazes: DFS can be used to generate mazes, where each cell in the maze is a vertex and edges represent passages between adjacent cells. It ensures that all cells are reachable and that the maze has no unreachable areas or loops.

- Artificial Intelligence: DFS is used in AI algorithms, such as depth-limited search and iterative deepening depth-first search, for solving problems in areas like planning, scheduling, and game playing.

v) Example

Depth First Search (DFS) is used in various real-world applications such as maze solving, where it explores paths deeply before backtracking. It's useful in network analysis for identifying connected components, and in pathfinding algorithms for maps and games. DFS also helps in topological sorting to resolve task dependencies, and it can solve puzzles like Sudoku by exploring possible solutions deeply. Additionally, web crawlers use DFS to traverse websites by following links from one

page to the next. These applications demonstrate the versatility of DFS in solving complex problems by systematically exploring all potential paths.

c) Heuristic Search - Best-First Search (MGAYA DENIS PASCAL)

i) Description

Best-First Search (BFS) is a heuristic search algorithm that explores a graph by expanding the most promising node chosen according to a specified rule. It uses an evaluation function to rank nodes, guiding the search towards the goal more efficiently than uninformed search algorithms.

ii) How It Works

BFS uses a priority queue to keep track of nodes to be explored. The priority is determined by a heuristic function that estimates the cost from the current node to the goal. Nodes with lower estimated costs are expanded first. The search continues until the goal node is reached or all possible nodes have been explored.

iii) Pseudocode

```
function best_first_search(start, goal, heuristic):

open_list = PriorityQueue()

closed_list = set()

open_list.put((heuristic(start, goal), start))


while not open_list.empty():

_, current_node = open_list.get()


if current_node == goal:
```

```
return reconstruct_path(current_node)


closed_list.add(current_node)


for neighbor in neighbors(current_node):
if neighbor in closed_list:
continue
if neighbor not in open_list or heuristic(neighbor, goal) < heuristic(current_node, goal):
open_list.put((heuristic(neighbor, goal), neighbor))
neighbor.parent = current_node


return None


function reconstruct_path(node):
path = []
while node:
path.append(node)
node = node.parent
return path.reverse()
```

iv) Application

BFS is often used in scenarios where we need to find the shortest path or the least cost path to a goal. It's applied in pathfinding algorithms in games, AI for decision-making, and robotics for navigation.

v) Example

Imagine a robot navigating a grid to find the shortest path from its starting position to a target position. Each cell in the grid represents a node. The heuristic function could be the Manhattan distance between the current cell and the target cell. BFS would explore the cells with the lowest estimated cost first, guiding the robot efficiently towards its target.

d) A* Search (MGAYA DENIS PASCAL)

i) Description

A* Search is a popular and efficient pathfinding algorithm that combines the strengths of Dijkstra's Algorithm and Best-First Search. It uses both the actual cost to reach a node and an estimated cost to reach the goal, ensuring both optimal and efficient search.

ii) How It Works

A* Search uses a priority queue to explore nodes. Each node is evaluated based on the function f(n)=g(n)+h(n), where g(n) is the actual cost from the start node to the current node and h(n) is the heuristic cost estimate from the current node to the goal. Nodes with the lowest f(n) value are expanded first. The algorithm continues until the goal is reached or all possible nodes are explored.

iii) Pseudocode

```
function A_star_search(start, goal, heuristic):

open_list = PriorityQueue()

closed_list = set()

start.g = 0

start.f = heuristic(start, goal)

open_list.put((start.f, start))


while not open_list.empty():
```

```
_, current_node = open_list.get()

if current_node == goal:
return reconstruct_path(current_node)

closed_list.add(current_node)

for neighbor in neighbors(current_node):
if neighbor in closed_list:
continue
tentative_g = current_node.g + cost(current_node, neighbor)

if tentative_g < neighbor.g or neighbor not in open_list:
neighbor.g = tentative_g
neighbor.f = neighbor.g + heuristic(neighbor, goal)
neighbor.parent = current_node
open_list.put((neighbor.f, neighbor))

return None

function reconstruct_path(node):
path = []
while node:
path.append(node)
```

```
node = node.parent
return path.reverse()
```

iv) Application

A* Search is widely used in various fields requiring efficient and optimal pathfinding. It is used in video games for character movement, robot navigation, route planning for autonomous vehicles, and network routing.

v) Example

Consider a game where a character needs to find the shortest path through a maze to reach a goal. The maze can be represented as a grid of nodes. The heuristic function could be the Euclidean distance between the current node and the goal node. A* Search would explore nodes by evaluating both the cost to reach the current node and the estimated cost to the goal, ensuring the character finds the shortest and most efficient path.

By using these algorithms, complex search problems can be solved efficiently, ensuring optimal and near-optimal solutions in various real-world applications.

e) Hill Climbing Algorithm (MCHOMVU SALIM I)

i) Description

Hill climbing is a heuristic search algorithm used for mathematical optimization problems. It is an iterative algorithm that starts with an arbitrary solution to a problem and then attempts to find a better solution by incrementally changing a single element of the solution. The algorithm continues to make changes as long as improvements are found.

ii) How It Works

Hill climbing works by evaluating the neighboring solutions of the current solution and moving to the neighbor with the highest improvement. If no improvement is found, the algorithm terminates, having reached a local optimum. There are different variations, such as simple hill climbing, steepest ascent hill climbing, and stochastic hill climbing.

Steps:

1. Start with an initial solution.
2. Evaluate the neighboring solutions.
3. Move to the neighbor with the best improvement.
4. Repeat steps 2 and 3 until no improvement is found.

iii) Pseudocode

```
function HillClimbing(initialState):

currentNode = initialState

loop do:

neighbor = getBestNeighbor(currentNode)

if neighbor.value <= currentNode.value:

return currentNode

currentNode = neighbor


function getBestNeighbor(node):

bestNeighbor = null

for each neighbor of node:
```

if bestNeighbor is null or neighbor.value > bestNeighbor.value:

bestNeighbor = neighbor

return bestNeighbor

iv) Application

Hill climbing is used in various optimization problems where finding an exact solution is difficult due to the vast search space. Some applications include:

- Scheduling problems
- Pathfinding algorithms in AI
- Game development for making strategic decisions
- Robotics for motion planning

v) Example

Consider a simple problem of finding the maximum value of the function f(x)=−(x−3)2+9f(x) = -(x-3)^2 + 9f(x)=−(x−3)2+9. The hill climbing algorithm would:

1. Start with an initial guess, say x=0x = 0x=0.
2. Evaluate the function at neighboring points, x=1x = 1x=1 and x=−1x = -1x=−1.
3. Move to the neighbor with the highest value, say x=1x = 1x=1.
4. Repeat the process until no better neighbors are found. In this case, it will converge to x=3x = 3x=3, where f(x)f(x)f(x) is maximum.

Example Code

Here's a simple Python implementation:

```
def hill_climbing(f, x_start, step_size, max_iter):

x = x_start

for _ in range(max_iter):

next_x = x + step_size

prev_x = x - step_size

if f(next_x) > f(x):

x = next_x

elif f(prev_x) > f(x):

x = prev_x

else:

break

return x



# Example function: f(x) = -(x-3)^2 + 9

def f(x):

return -(x-3)**2 + 9



# Start at x = 0

result = hill_climbing(f, x_start=0, step_size=0.1, max_iter=100)

print("Maximum value is at x =", result)
```

In this example, the algorithm will move towards x=3x = 3x=3 and find the maximum value of the function.

f) Dynamic Programming Algorthim (MCHOMVU SALIM I)

i) Description

Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them down into simpler subproblems. It is particularly useful for problems exhibiting overlapping subproblems and optimal substructure properties. By storing the results of subproblems, DP avoids redundant computations, significantly reducing the time complexity compared to naive approaches.

ii) How It Works

Dynamic Programming works by following these steps:

1. **Characterize the structure of an optimal solution.**
2. **Define the value of an optimal solution recursively in terms of the values of smaller subproblems.**
3. **Compute the value of an optimal solution (typically in a bottom-up fashion).**
4. **Construct an optimal solution from the computed information.**

DP can be implemented using either a **top-down approach** (with memoization) or a **bottom-up approach** .

iii) Pseudocode

Here's a pseudocode for solving the Fibonacci sequence using dynamic programming:

Top-Down Approach (Memoization)

```
function Fibonacci(n, memo):
if n in memo:
return memo[n]
if n <= 1:
return n
memo[n] = Fibonacci(n-1, memo) + Fibonacci(n-2, memo)
return memo[n]

function Fibonacci(n):
memo = {}
return Fibonacci(n, memo)
```

Bottom-Up Approach (Tabulation)

```
function Fibonacci(n):
if n <= 1:
return n
dp = array of size (n+1)
dp[0] = 0
dp[1] = 1
for i from 2 to n:
dp[i] = dp[i-1] + dp[i-2]
```

return dp[n]

iv) Application

Dynamic Programming is used in a wide range of applications, including:

- **Optimization problems:** Finding the shortest path, the longest common subsequence, the knapsack problem, etc.
- **Bioinformatics:** Sequence alignment, gene prediction.
- **Economics:** Portfolio optimization, resource allocation.
- **Operations research:** Scheduling, network flow.

v) Example

Consider the problem of finding the minimum number of coins needed to make a certain amount of money given a set of coin denominations. This problem can be solved using dynamic programming.

Here's a simple Python implementation:

```
def min_coins(coins, amount):
# Initialize DP table with a value higher than the possible minimum
dp = [float('inf')] * (amount + 1)
dp[0] = 0  # Base case: 0 coins are needed to make the amount 0

for coin in coins:
for x in range(coin, amount + 1):
```

```
dp[x] = min(dp[x], dp[x - coin] + 1)

return dp[amount] if dp[amount] != float('inf') else -1

# Example usage
coins = [1, 2, 5]
amount = 11
result = min_coins(coins, amount)
print("Minimum number of coins needed:", result)
```

In example above :

- We have coins of denominations 1, 2, and 5.
- We aim to make the amount 11.
- The algorithm finds the minimum number of coins needed, which is 3 (5+5+1).

This dynamic programming solution efficiently computes the minimum number of coins by storing intermediate results and building up to the final solution.

g) Uniform Cost Search (UCS) (KANOKOLEKE REHEMA BAKARI)

i) Description

Uniform Cost Search (UCS) is an uninformed search algorithm that expands the least cost node first. It's essentially Dijkstra's algorithm applied for searching, ensuring the shortest path is always chosen by considering the cumulative cost from the start node.

ii) How It Works

UCS uses a priority queue to explore nodes. The priority is based on the cumulative cost from the start node to the current node. The algorithm expands the node with the lowest cumulative cost first. It continues until the goal node is reached or all nodes have been explored. UCS guarantees finding the least-cost path if the cost is positive for all edges.

iii) Pseudocode

```
function uniform_cost_search(start, goal):

open_list = PriorityQueue()

closed_list = set()

open_list.put((0, start))

while not open_list.empty():

cost, current_node = open_list.get()

if current_node == goal:

return reconstruct_path(current_node)

closed_list.add(current_node)

for neighbor, step_cost in neighbors(current_node):

if neighbor in closed_list:
```

```
continue

new_cost = cost + step_cost

if neighbor not in open_list or new_cost < neighbor.cost:

neighbor.cost = new_cost

open_list.put((new_cost, neighbor))

neighbor.parent = current_node


return None


function reconstruct_path(node):

path = []

while node:

path.append(node)

node = node.parent

return path.reverse()
```

iv) Application

UCS is suitable for scenarios where the path with the lowest cost is required, irrespective of the number of steps. It is used in network routing, robot pathfinding where energy cost or time cost is considered, and logistics planning where minimizing the overall cost is essential.

v) Example

Consider a delivery robot navigating a city grid. Each intersection is a node, and the streets connecting them have different travel costs (e.g., based on distance or traffic). UCS ensures the robot finds the path with the lowest travel cost to reach the destination. For instance, if the robot starts at point A and needs to reach point B, UCS will explore all possible paths and expand the least costly nodes first, ensuring the most cost-effective route is taken.

2. Knowledge representation in AI is about how information is structured so that machines can use it to simulate human reasoning and make decisions. Here are some key methodologies:

a. Propositional Logic (KANOKOLEKE REHEMA BAKARI)

i) Description

Propositional logic (or Boolean logic) represents knowledge using propositions, which are statements that can either be true or false. It is the simplest form of logic used in AI for knowledge representation.

ii) Methodology

- **Syntax**: Propositions are atomic units denoted by letters (e.g., P, Q, R). Complex statements are formed using logical connectives such as AND (∧), OR (∨), NOT (¬), and IMPLIES (→).
- **Semantics**: Each proposition is assigned a truth value (true or false). The truth value of complex statements is determined based on the truth values of their components.

Example

Consider a simple knowledge base about the weather:

- PPP: It is raining.
- QQQ: The ground is wet.
- Rule: P→QP \rightarrow QP→Q (If it is raining, then the ground is wet).

If PPP is true (it is raining), then QQQ must also be true (the ground is wet).

b. Predicate Logic (NDAGULA, FRANK JONAS)

i) Description

Predicate logic (or first-order logic) extends propositional logic by introducing quantifiers and predicates, allowing for the representation of more complex statements involving objects and their relationships.

ii) Methodology

- **Syntax**: Predicates represent properties or relations among objects (e.g., Loves(John, Mary)). Variables (e.g., x, y) and quantifiers (e.g., ∀ for "for all", ∃ for "there exists") are used to create more expressive statements.
- **Semantics**: Predicates are interpreted over a domain of discourse. The truth of a predicate depends on the assignment of objects to variables.

iii) Example

Consider a knowledge base about family relationships:

- Loves(John,Mary)Loves(John, Mary)Loves(John,Mary): John loves Mary.
- ∀x∀y(Loves(x,y)→Happy(x))\forall x \forall y (Loves(x, y) \rightarrow Happy(x))∀x∀y(Loves(x,y)→Happy(x)): For all x and y, if x loves y, then x is happy.

This allows the representation of more nuanced knowledge compared to propositional logic.

c. Rule-Based Systems (NDAGULA FRANK JONAS)

i) Description

Rule-based systems use a set of "if-then" rules to represent knowledge. These systems apply rules to a set of facts to infer new facts or make decisions.

ii) Methodology

- **Syntax**: Rules are written in the form of "if condition then action". The condition part specifies when the rule applies, and the action part specifies what to do when the condition is met.
- **Inference Engine**: The system's inference engine matches rules against the knowledge base to derive new information.

iii) Example

Consider an expert system for medical diagnosis:

- Rule: If a patient has a fever and a cough, then they might have the flu.
- Facts: Patient A has a fever. Patient A has a cough.
- Inference: The system concludes that Patient A might have the flu based on the rule.

d. Probabilistic Systems (NDAGULA FRANK JONAS)

i) Description

Probabilistic systems represent knowledge using probabilities, allowing for the handling of uncertainty and incomplete information.

ii) Methodology

- **Bayesian Networks**: A common probabilistic model that represents variables and their conditional dependencies using a directed acyclic graph (DAG). Each node represents a variable, and edges represent probabilistic dependencies.
- **Markov Models**: Used for modeling sequences of events, where the probability of each event depends only on the previous event (Markov property).

iii. Example

Consider a Bayesian network for diagnosing diseases based on symptoms:

- Variables: Disease (D), Symptom 1 (S1), Symptom 2 (S2).

- Probabilities: P(D)P(D)P(D), P(S1|D)P(S1|D)P(S1|D), P(S2|D)P(S2|D)P(S2|D).

Given observed symptoms, the network can infer the probability of different diseases using Bayes' theorem.