Date: 10 - 01 - 2026
Assignment No.: 1b

Name: Varghese K James
Reg No.: 3122245001192

Excercise 1b – Uninformed Search Strategy - Uniform Cost Search

Aim: Implement Uniform Cost Search (UCS)

Source Code

```
import heapq

class Search_problem:
    def start_node(self):
        pass
    def is_goal(self, node):
        pass
    def neighbors(self, node):
        pass
    def heuristic(self, node):
        return 0

class Edge:
    def __init__(self, from_node, to_node, action=None, cost=1):
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost = cost
        if self.cost < 0:
            raise ValueError("Edge cost must be non-negative.")

    def __repr__(self):
        action_str = f", action={self.action}" if self.action is not None else ""
        return f"Edge({self.from_node} -> {self.to_node}{action_str}, cost={self.cost})"

class Search_problem_from_explicit_graph(Search_problem):
    def __init__(self, nodes, edges, start, goals, heuristics=None):
        self.nodes = set(nodes)
        self.edges_list = edges
        self.start = start
        self.goals = set(goals)
        self.heuristics_map = heuristics if heuristics else {}

        self.adjacency_list = {node: [] for node in self.nodes}
        for edge in self.edges_list:
            if edge.from_node in self.adjacency_list:
                self.adjacency_list[edge.from_node].append(edge)

    def start_node(self):
        return self.start

    def is_goal(self, node):
        return node in self.goals

    def neighbors(self, node):
        return self.adjacency_list.get(node, [])
```

```
    def heuristic(self, node):
        return self.heuristics_map.get(node, 0)


class Path:
    def __init__(self, start_node=None, parent_path=None, edge=None):
        if parent_path is None:
            self.last_node = start_node
            self.parent_path = None
            self.incoming_edge = None
            self.cost = 0
        else:
            self.last_node = edge.to_node
            self.parent_path = parent_path
            self.incoming_edge = edge
            self.cost = parent_path.cost + edge.cost

    def end(self):
        return self.last_node

    def __repr__(self):
        nodes = []
        current = self
        while current:
            nodes.append(str(current.last_node))
            current = current.parent_path
        nodes.reverse()
        path_str = " -> ".join(nodes)
        return f"Path([{path_str}], total_cost={self.cost})"


class Searcher:
    def __init__(self, problem):
        self.problem = problem

    def search(self):
        print("Starting Search")
        start = self.problem.start_node()
        initial_path = Path(start_node=start)

        frontier = []
        tie_breaker_index = 0

        heapq.heappush(frontier, (0, tie_breaker_index, initial_path))
        tie_breaker_index += 1

        explored = set()

        while frontier:
            path_cost, _, current_path = heapq.heappop(frontier)

            print(f"\nExploring: {current_path}")

            current_node = current_path.end()
```

```
            if current_node in explored:
                print(f"  Node {current_node} already explored, skipping.")
                continue

            explored.add(current_node)

            if self.problem.is_goal(current_node):
                print("  Goal condition met!")
                return current_path

            neighbors = self.problem.neighbors(current_node)
            if not neighbors:
                print("  No neighbors found.")

            for edge in neighbors:
                print(f"  Processing neighbor via: {edge}")

                child_node = edge.to_node
                if child_node not in explored:
                    new_path = Path(parent_path=current_path, edge=edge)

                    heapq.heappush(frontier, (new_path.cost, tie_breaker_index,
new_path))
                    tie_breaker_index += 1

        return None


if __name__ == "__main__":
    nodes_set = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'}

    edges_list = [
        Edge('A', 'C', cost=3),
        Edge('A', 'B', cost=2),
        Edge('A', 'D', cost=4),
        Edge('C', 'J', cost=7),
        Edge('B', 'E', cost=2),
        Edge('B', 'F', cost=3),
        Edge('F', 'D', cost=2),
        Edge('D', 'H', cost=4),
        Edge('H', 'G', cost=3),
        Edge('J', 'G', cost=4)
    ]

    heuristics = {
        'A': 7,
        'B': 5,
        'C': 9,
        'D': 6,
        'E': 3,
        'F': 5,
        'G': 0,
        'H': 3,
        'J': 4
    }
```

```
problem_instance = Search_problem_from_explicit_graph(
    nodes=nodes_set,
    edges=edges_list,
    start='A',
    goals={'G'},
    heuristics=heuristics
)

searcher = Searcher(problem_instance)
result_path = searcher.search()

print()
if result_path:
    print("Success:", result_path)
else:
    print("No path found.")
```

Name: Varghese K James
Reg No.: 3122245001192

Input/Output

```
vicfic@NAG3:~/stud/ai/ex2$ python ucs.py
Starting Search

Exploring: Path([A], total_cost=0)
  Processing neighbor via: Edge(A -> C, cost=3)
  Processing neighbor via: Edge(A -> B, cost=2)
  Processing neighbor via: Edge(A -> D, cost=4)

Exploring: Path([A -> B], total_cost=2)
  Processing neighbor via: Edge(B -> E, cost=2)
  Processing neighbor via: Edge(B -> F, cost=3)

Exploring: Path([A -> C], total_cost=3)
  Processing neighbor via: Edge(C -> J, cost=7)

Exploring: Path([A -> D], total_cost=4)
  Processing neighbor via: Edge(D -> H, cost=4)

Exploring: Path([A -> B -> E], total_cost=4)
  No neighbors found.

Exploring: Path([A -> B -> F], total_cost=5)
  Processing neighbor via: Edge(F -> D, cost=2)

Exploring: Path([A -> D -> H], total_cost=8)
  Processing neighbor via: Edge(H -> G, cost=3)

Exploring: Path([A -> C -> J], total_cost=10)
  Processing neighbor via: Edge(J -> G, cost=4)

Exploring: Path([A -> D -> H -> G], total_cost=11)
  Goal condition met!

Success: Path([A -> D -> H -> G], total_cost=11)
vicfic@NAG3:~/stud/ai/ex2$
```

Date: 10 - 01 - 2026
Assignment No.: 1b

Name: Varghese K James
Reg No.: 3122245001192

Learning Outcomes

- Implemented the Uniform Cost Search (UCS) algorithm using Python's heapq.
- Learn to properly represent graph search problems using recursive path structures and abstract base classes.
- Understood the importance of priority queues in efficiently managing the search frontier to guarantee optimal solutions