A* Search and

Recursive Best-First Search

Aim:

4) Implement the A* Search algorithm for solving a path finding problem.
5) Implement the Recursive Best-First Search algorithm for the same problem.
6) Compare the performance and effectiveness of both algorithms.

Theory:

Part 1: A* Search algorithm:

A* (A-star) is a widely used search algorithm that combines the best features of both Dijkstra's algorithm and heuristic search. It is commonly applied to solve the path-finding problem in graphs or grids, where the goal is to find the shortest path from a start node to a goal node.

The A* algorithm works by maintaining two main values for each node: the cost to reach the node from the start node (known as g-value), and an estimate of the cost from the node to the goal node (known as h-value). It uses a priority queue, typically implemented as a min-heap, to prioritize the nodes for exploration based on their f-value, which is the sum of the g-value and h-value.

The A* algorithm follows these steps:

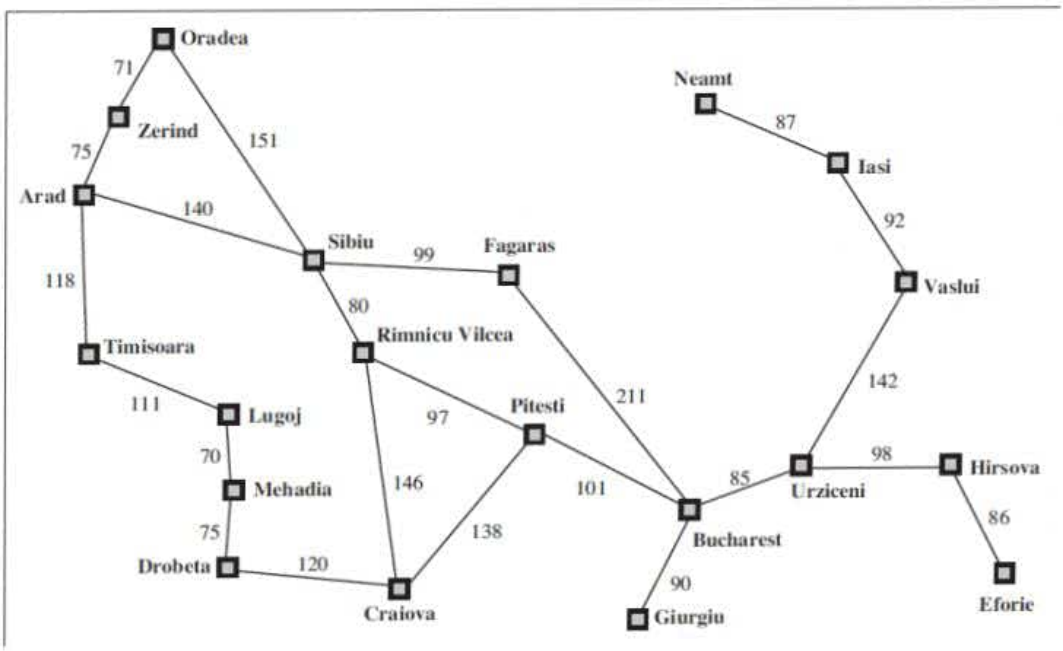
a) Initialize the open list, closed list, and set the g-value of the start node to 0.
b) Calculate the h-value for each node in the graph or grid based on a heuristic function. The heuristic function estimates the cost from each node to the goal node. Common heuristic functions include Euclidean distance, Manhattan distance, or any other admissible and consistent heuristic.
c) Enqueue the start node to the open list with its f-value as the priority.
d) Repeat the following steps until the open list becomes empty or the goal node is reached:
    I. Dequeue the node with the lowest f-value from the open list. This node becomes the current node.
    II. If the current node is the goal node, the algorithm terminates, and the path has been found.
    III. Add the current node to the closed list to mark it as visited.
    IV. Explore the neighboring nodes of the current node:
        i. Calculate the tentative g-value for each neighbor by adding the cost to reach the neighbor from the current node to the g-value of the current node.
        ii. If the neighbor is not in the closed list or its tentative g-value is lower than its current g-value:
            1) Update the g-value of the neighbor to the new lower value.
            2) Calculate the f-value of the neighbor by adding its g-value and h-value.
            3) If the neighbor is not in the open list, enqueue it with its f-value as the priority.
            4) If the neighbor is already in the open list, update its priority if the new f-value is lower.
            5) Set the parent of the neighbor to the current node.
e) If the open list becomes empty before reaching the goal node, there is no path available.

f) Once the goal node is reached, reconstruct the path by following the parent pointers from the goal node to the start node.

The A* algorithm is both complete (able to find a solution if one exists) and optimal (guaranteed to find the shortest path) under certain conditions. The heuristic used must be admissible, meaning it never overestimates the actual cost to reach the goal node. Additionally, the heuristic must be consistent (or monotonic), meaning the estimated cost from a node to its successor plus the heuristic value of the successor is always less than or equal to the estimated cost from the current node to the goal node.

A* is widely used in various applications such as path planning, robotics, navigation systems, and game AI due to its efficiency and optimality. It efficiently explores the search space by prioritizing nodes that are most likely to lead to the goal while considering the actual cost from the start node.

We consider the Map of Romania problem and solve it using A* Search



Python code:

```
import heapq
# Define the map of Romania with distances between cities
romania_map = {
   'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
   'Zerind': {'Arad': 75, 'Oradea': 71},
   'Timisoara': {'Arad': 118, 'Lugoj': 111},
   'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
   'Oradea': {'Zerind': 71, 'Sibiu': 151},
   'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
   'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
   'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
   'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
   'Drobeta': {'Mehadia': 75, 'Craiova': 120},
   'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
   'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
   'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
   'Giurgiu': {'Bucharest': 90},
```

```
    'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}
}

class Node:
    def __init__(self, city, cost, parent=None):
        self.city = city
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost

def heuristic(node, goal):
    return 0  # No need for heuristic in this case

def astar_search(graph, start, goal):
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.city == goal.city:
            path = []
            while current_node:
                path.append(current_node.city)
                current_node = current_node.parent
            return path[::-1]  # Reverse the path to get it from start to goal

        closed_set.add(current_node.city)

        for neighbor, distance in graph[current_node.city].items():
            if neighbor not in closed_set:
                new_cost = current_node.cost + distance
                new_node = Node(neighbor, new_cost, current_node)
                heapq.heappush(open_list, new_node)

    return None  # No path found

start_city = 'Arad'
goal_city = 'Bucharest'

start_node = Node(start_city, 0)
goal_node = Node(goal_city, 0)
```

```
path = astar_search(romania_map, start_node, goal_node)
if path:
    print("Path found:", path)
else:
    print("No path found")
```

Output: **['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']**

Part 2: Recursive Best-First Search algorithm:

Recursive Best-First Search (RBFS) is an informed search algorithm used for finding the shortest path from a start node to a goal node in a graph or tree. RBFS is a memory-bounded variant of the A* algorithm that avoids the need for storing the entire search tree explicitly.

RBFS utilizes a heuristic function that estimates the cost from each node to the goal node. It maintains a priority queue of nodes to be explored, sorted based on their f-values, which is the sum of the cost to reach the node from the start node (g-value) and the estimated cost from the node to the goal node (h-value).

The RBFS algorithm follows these steps:

a) Initialize the recursive function RBFS with the current node, goal node, and a limit value (initially set to infinity).
b) Check if the current node is the goal node. If it is, return the path containing only the current node, indicating success.
c) Generate the successors of the current node and calculate their f-values using a heuristic function.
d) If there are no successors, return None to indicate failure.
e) Loop over the successors in order of their f-values:
    i. Recursively call RBFS on the successor with the minimum f-value, the goal node, and the minimum of the current limit and the f-value of the next best successor. This recursive call effectively explores the subtree rooted at the selected successor.
    ii. If the recursive call returns a path, return the path concatenated with the current node, indicating success.
    iii. If the recursive call returns None, update the limit to the maximum f-value among the successors. This limit represents the threshold beyond which RBFS will not explore further.
    iv. If all successors have been explored and none of them resulted in a path, return None to indicate failure.
    v. The RBFS algorithm continues to recursively explore the graph/tree, using the minimum limit from the failed recursive calls as the new limit for subsequent iterations. This allows RBFS to "backtrack" and reconsider previously rejected nodes if a better path is discovered.

RBFS terminates when a path is found or when it exhaustively explores all nodes without finding a path. The algorithm is memory-efficient since it only keeps track of the current path and the best f-value encountered so far.

RBFS is especially useful in scenarios where memory is limited, and it prioritizes exploring promising nodes based on their f-values, leading to efficient search in large state spaces.

Python code for Recursive Best-First Search algorithm

```
from queue import PriorityQueue

class Node:
    def __init__(self, state, parent=None, f=float('inf')):
        self.state = state
        self.parent = parent
        self.f = f

def rbfs(start, goal):
    f_limit = float('inf')
    stack = [(Node(start, f=0), f_limit)]
    visited = set()

    while stack:
        (node, f) = stack.pop()
        visited.add(node.state)

        if node.state == goal:
            path = []
            cost = node.f
            while node is not None:
                path.append(node.state)
                node = node.parent
            return list(reversed(path)), cost

        successors = []
        for neighbor, cost in get_neighbors(node.state):
            if neighbor not in visited:
                child = Node(neighbor, parent=node)
                child.f = max(child.parent.f, cost)
                successors.append(child)

        if len(successors) == 0:
            continue

        successors.sort(key=lambda x: x.f)
        best = successors[0]

        if best.f > f_limit:
            return None, best.f

        alternative = successors[1].f if len(successors) > 1 else float('inf')
        stack.append((best, min(f_limit, alternative)))
```

```
    return None, float('inf')

def get_neighbors(state):
    # Define the successors for each state with their associated costs (simplified
example).
    successors = {
        1: [(2, 3), (3, 5)],
        2: [(1, 3), (4, 7)],
        3: [(1, 5), (5, 2)],
        4: [(2, 7), (6, 4)],
        5: [(3, 2), (7, 6)],
        6: [(4, 4), (8, 8)],
        7: [(5, 6), (8, 5)],
        8: [(6, 8), (7, 5)],
    }
    return successors.get(state, [])

if __name__ == '__main__':
    start_state = 1
    goal_state = 8

    path, cost = rbfs(start_state, goal_state)

    if path is not None:
        print(f"Optimal path from {start_state} to {goal_state}:")
        print(" -> ".join(map(str, path)))
        print(f"Total cost: {cost}")
    else:
        print("No path found.")
```

Part 3: Comparing the performance and efficiency of the two algorithms

In the comparison of efficiency and performance between the A* and RBFS algorithms, we can consider factors such as time complexity, space complexity, and search behavior as follows

Efficiency:

Time Complexity:

A*: The time complexity of A* depends on the heuristic function used. In the worst case, where the heuristic is not admissible, A* may explore more nodes than necessary. However, with an admissible heuristic, the time complexity is typically much better than blind search algorithms like Breadth-First Search or Depth-First Search.
RBFS: The time complexity of RBFS is generally exponential, as it explores nodes recursively. It explores the search space by backtracking and re-examining previously rejected nodes, which can lead to redundant exploration.
Space Complexity:

A*: A* requires additional memory to store the open list, closed list, and node information. The space complexity of A* depends on the branching factor of the graph and the size of the search space.
RBFS: RBFS has a better space complexity compared to A* since it doesn't require storing the entire search tree explicitly. RBFS only keeps track of the current path and the best f-value encountered so far, resulting in lower memory consumption.

Performance:

Search Behavior:
A*: A* is guaranteed to find the optimal path when using an admissible heuristic. It efficiently explores the search space by prioritizing nodes based on their f-values. However, if the heuristic is not admissible or inconsistent, A* may expand more nodes than necessary.
RBFS: RBFS is not guaranteed to find the optimal path. It explores the search space recursively, considering the best successor node based on f-values. However, due to its backtracking nature, RBFS may revisit nodes and spend more time exploring redundant paths.
Based on the above analysis, A* generally outperforms RBFS in terms of efficiency and performance. A* with an admissible and consistent heuristic tends to explore fewer nodes compared to RBFS. It provides an optimal path and can be more time-efficient, especially in larger search spaces. However, it requires more memory to store the open and closed lists.

On the other hand, RBFS is memory-efficient and can handle larger search spaces with limited memory. However, it may spend more time exploring redundant paths and is not guaranteed to find the optimal solution.

It's important to note that the performance of both algorithms heavily relies on the specific problem and the quality of the heuristic function used. It's advisable to carefully choose or design an appropriate heuristic for the problem domain to maximize the efficiency and performance of these algorithms.

For Video demonstration of the practical click on the link or scan the QR-code

A* - Search https://youtu.be/2aBPyxr76kQ

Recursive Depth First Search https://youtu.be/kPkr1dMxLG4