| **EX.N0 : 06** | **RECURSIVE BEST-FIRST SEARCH** |
| --- | --- |
| **DATE : 03.04.2024** |

**AIM:**

To implement the Recursive Best-First Search using python.

**ALGORITHM:**

Step 1: Start with an initial node and add it to a priority queue.

Step 2:  While the priority queue is not empty:

Step 3: Pop the node with the lowest estimated cost.

Step 4: If the popped node is a goal state, you've found the solution.

Step 5:  Generate child nodes and estimate their costs.

Step 6: Sort child nodes by their estimated costs.

Step 7: Recursively apply RBFS to the child with the lowest estimated cost.

Step 8: If a node returns a failure, update its cost to be higher than the next best child.

Step 9:  Continue until a solution is found or all nodes are explored.

Step 10: RBFS terminates when a solution is found or when all nodes have been explored.

**PROGRAM:**

class Node:

def \_\_init\_\_(self, state, parent=None, cost=0, heuristic=0):

self.state = state

self.parent = parent

self.cost = cost # g(n)

self.heuristic = heuristic # h(n)

self.f\_cost = cost + heuristic # f(n) = g(n) + h(n)

def \_\_lt\_\_(self, other):

return self.f\_cost < other.f\_cost

def \_\_repr\_\_(self):

return f"Node({self.state}, f\_cost={self.f\_cost})"

def rbfs(problem, node, f\_limit):

print(f"RBFS called with node: {node}, f\_limit: {f\_limit}"

if problem.is\_goal(node.state):

return node, 0 # Solution found

successors = []

for successor in problem.get\_successors(node.state):

s = Node(successor['state'], node, node.cost + successor['cost'], successor['heuristic'])

successors.append(s)

if not successors:

return None, float('inf') # Dead end

successors.sort()

while successors:

best = successors[0]

if best.f\_cost > f\_limit:

return None, best.f\_cost

alternative = successors[1].f\_cost if len(successors) > 1 else float('inf')

result, best.f\_cost = rbfs(problem, best, min(f\_limit, alternative))

if result is not None:

return result, best.f\_cost

successors.sort()

return None, float('inf')

class Problem:

def \_\_init\_\_(self, initial\_state, goal\_state, successors):

self.initial\_state = initial\_state

self.goal\_state = goal\_state

self.successors = successors

def is\_goal(self, state):

return state == self.goal\_state

def get\_successors(self, state):

return self.successors[state]

defrecursive\_best\_first\_search(problem): start\_node=Node(problem.initial\_state,None,0,problem.get\_successors(problem.initial\_state)[0]['heuristic'])

result, \_ = rbfs(problem, start\_node, float('inf'))

return result

# Example usage:

initial\_state = 'A'

goal\_state = 'G'

successors = {

'A': [{'state': 'B', 'cost': 1, 'heuristic': 5}, {'state': 'C', 'cost': 4, 'heuristic': 2}],

'B': [{'state': 'D', 'cost': 2, 'heuristic': 2}, {'state': 'E', 'cost': 5, 'heuristic': 1}],

'C': [{'state': 'F', 'cost': 1, 'heuristic': 4}],

'D': [{'state': 'G', 'cost': 3, 'heuristic': 0}],

'E': [],

'F': [{'state': 'G', 'cost': 1, 'heuristic': 0}],

'G': []

}

problem = Problem(initial\_state, goal\_state, successors)

solution = recursive\_best\_first\_search(problem)

# Reconstruct path

path = []

node = solution

while node:

path.append(node.state)

node = node.parent

path.reverse()

print("Path to goal:", path)

**OUTPUT:**

****

**RESULT:**

Thus to implement the Recursive Best-First Search Algorithm using python has been executed successfully.