Ex no : 13

Solve any problem using Best first search.

Aim:

Write the program to implementation of Best First Search Algorithm

Algorithm:

- Initialize the open list (priority queue) with the start node.
- Initialize the closed list (visited nodes) as empty.
- While the open list is not empty:
  a) Remove the node with the lowest heuristic value from the open list.
  b). If this node is the goal, return the path.
  c). Otherwise, generate all successors of the current node. D
- If the successor is not in the open list or closed list, add it to the open list and record its parent.
- If the successor is in the open list with a higher cost, update its cost and parent.
- If the open list is empty and no goal is found, return failure.

Source code:

```
import heapq

class Node:
    def __init__(self, state, parent, cost, heuristic):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return self.heuristic < other.heuristic
```

```
def best_first_search(start, goal, heuristic_fn,
get_neighbors_fn):
    open_list = []
    closed_list = set()
    start_node = Node(start, None, 0, heuristic_fn(start,
goal))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.state == goal:
            return reconstruct_path(current_node)

        closed_list.add(current_node.state)

        for neighbor, cost in
get_neighbors_fn(current_node.state):
            if neighbor in closed_list:
                continue

            neighbor_node = Node(neighbor, current_node,
current_node.cost + cost, heuristic_fn(neighbor, goal))
            for open_node in open_list:
                if open_node.state == neighbor and
open_node.cost <= neighbor_node.cost:
                    break
            else:
                heapq.heappush(open_list, neighbor_node)

    return None

def reconstruct_path(node):
    path = []
```

```python
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]
def manhattan_distance(state, goal):
    return abs(state[0] - goal[0]) + abs(state[1] - goal[1])
def get_neighbors(state):
    neighbors = []
    x, y = state
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        neighbor = (x + move[0], y + move[1])
        if 0 <= neighbor[0] < 5 and 0 <= neighbor[1] < 5:
            neighbors.append((neighbor, 1))
    return neighbors
start = (0, 0)
goal = (4, 4)
path = best_first_search(start, goal, manhattan_distance,
get_neighbors)
print("Path found:", path)
```

Output:

Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

Result:

The implementation of best first search algorithm was successfully executed