15/10/24.

DATE: PAGE:

Iterative Deepening Search:

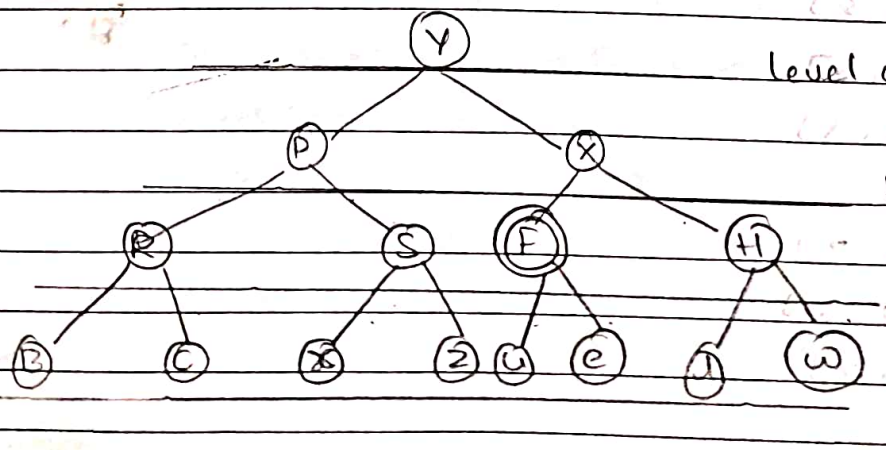
1. Define a function called iterative deepening search with a parameter graph, start node and a goal node
2. Define a function depth with a parameter node, goal and depth.
3. if depth == 0:
    if node == goal:
        return [node]
    else:
        return None.

or else it will go to the depth of the graph and it will check the child nodes.

3. while true:
    result = ids(start, goal, graph)
    if result is None.
        depth = depth + 1.

4. Define a function called user_input. take a users input for number of edges, and give a edges and give a start node and the goal node.

5. In the main function it will check the path is found or not.



DATE: PAGE:

Iteration	States
0	Y
1	Y -> P -> X
2	Y -> P -> X -> R -> S -> F
.	

8 Puzzle using A*

1	2	3
8		4
7	6	5

initial state

2	8	1
	4	3
7	6	5

goal state

1. Define a heuristic function to calculate the counting number of misplaced state and target we are using a zip function to combine the state and the target ~~function~~.
2. Define a evaluation function state level contain the current state and we call a ~~matte~~ heuristic function to calculate then we are combining the h(n) and g(n).
3. The we will take a directions left, right, up and down.
4. Display the state in 3x3 grid format.
   It will range from 1 to 9 take a empty cells as 0.
5. Define a function A* with parameters state, target.

```
ass = [[src, 0]]
visited = []
Iterations = 0.
```

DATE: PAGE:

6. We will check the conditions and iteration will be incremented.

7. It will display the current state where the target is there.

8. Append a ~~to~~ visited states to the list and give a initial state and goal state. and call a function.

Proceed

1) Iterative deepening search.

```
def ids(graph, start, goal):
    def depth_limited_search(node, goal, depth):
        if depth == 0:
            if node == goal:
                return [node]
            else:
                return None
        else depth > 0:
            for child in graph.get(node, []):
                result = depth_limited_search(child,
                                              goal, depth-1)
                if result is not None:
                    return [node] + result.
            return None
    depth = 0:
    while True:
        result = depth_limited_search(start, goal, depth)
        if result is not None:
            return result.
        depth += 1
```

DATE: PAGE:

```
def get_user_input_graph():
    graph = { }
    num_edges = int(input("Enter the number of edges:"))
    print("Enter each edge in format 'node1 node2':")
    for _ in range(num_edges):
        node1, node2 = input().split()
        if node1 in graph:
            graph[node1].append(node2)
        else:
            graph[node1] = [node2]
        if node2 in graph:
            graph[node2].append(node1)
        else:
            graph[node2] = [node1]
    return graph
def main():
    graph = get_user_input_graph()
    start_node = input("Enter the starting node: ")
    goal_node = input("Enter the goal node: ")
    path = ids(graph, start_node, goal_node)
    if path:
        print(f"Path found: {' -> '.join(path)}")
    else:
        print("No path found")
if __name__ == "__main__":
    main()
```

Output:

```
Enter the number of edges: 14
Enter each edge in the format 'node1 node2':
Y P
```

YX
PR
PS
XF
XH
RB
RC
SX
SZ
FU
FE
HL
HW

Enter the starting node: Y
Enter the goal node: F
Path found: Y -> X -> F

2) A*

```
def H_n(state, target):
    return sum(x != y for x, y in zip(state, target))
def F_n(state_with_lvl, target):
    state, lvl = state_with_lvl
    return H_n(state, target) + lvl
def possible_moves(state_with_lvl, visited_states):
    state, lvl = state_with_lvl
    b = state.index(0)
    directions = []
    pos_moves = []
    if b <= 5: directions.append('d')
    if b >= 3: directions.append('u')
```

```
if b%3>0: directions.append('l')
if b%3<2: directions.append('r')
for move in directions:
    temp = gen(state, move, b)
    if temp not in visited_states:
        pos_moves.append([temp, lvl+1])
return pos_moves
def gen(state, move, b):
    temp=state.copy()
    if move=='l': temp[b], temp[b-1]= temp[b-1], temp[b]
    if move=='r': temp[b], temp[b+1] = temp[b+1], temp[b]
    if move=='u': temp[b], temp[b-3] = temp[b-3], temp[b]
    if move=='d': temp[b], temp[b+3] = temp[b+3], temp[b]
    return temp
def display_state(state):
    print("current state:")
    for i in range(0,9,3):
        print(state[i:i+3])
    print()
def astar(src, target):
    ass=[[src, 0]]
    visited_states=[]
    iterations=0
    while ass:
        iterations+=1
        current = min(ass, key=lambda x: f-n(x, target))
        ass.remove(current)
        display_state(current[0])
        if current[0]==target:
            return f'Found with {iterations} iterations'
        visited_states.append(current[0])
```

DATE: PAGE:

```
        arr.extend(possible_moves(current, visited_states))
    return 'not found'
src = [1, 2, 3, 8, 0, 4, 7, 6, 5]
target = [2, 8, 1, 0, 4, 3, 7, 6, 5]
print(astar(src, target))
```

output:

```
[1, 2, 3]      [1, 2, 3]      [8, 1, 2]
[8, 0, 4]      8, 4, 5        0, 4, 3
[7, 6, 5]      7, 6, 0]       7, 6, 5];

[1, 2, 3       [0, 1, 3]      [2, 8, 3
0, 8, 4        [8, 2, 4]      0, 1, 4
7, 6, 5]       [7, 6, 5]      7, 6, 5]

[1, 2, 3       [1, 3, 0       [2, 8, 3,
8, 4, 0        8, 2, 4        1, 4, 0
7, 6, 5]       7, 6, 5]       7, 6, 5]

[1, 2, 0       [0, 1, 2       [2, 8, 0
8, 4, 3        8, 4, 3        1, 4, 3
7, 6, 5]       7, 6, 5]       7, 6, 5]

[1, 0, 2       [2, 0, 3]      [1, 8, 0
8, 4, 3        1, 8, 4        [2, 3, 4
7, 6, 5]       7, 6, 5]       1, 8, 0
                              7, 6, 5]

[1, 2, 3       [8, 1, 3
8, 6, 4        0, 2, 4        [2, 8, 1
7, 0, 5]       7, 6, 5]       0, 4, 3
                              7, 6, 5]
```