py1

September 21, 2024

Uninformed search technique(BFS,DFS,IDDFS)

[ ]:
```

```

[ ]:
```
from collections import deque

def bfs(g, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node)
            visited.add(node)
            for n in g[node]:
                if n not in visited:
                    queue.append(n)

# New graph with different values
g = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'E'],
    'D': ['B', 'F'],
    'E': ['B', 'C'],
    'F': ['D']
}

bfs(g, 'A')
```

[ ]:
```
def dfs(g, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            print(node)
```

```
            visited.add(node)
            stack.extend(reversed(g[node]))

# New graph with different values
g = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'G'],
    'F': ['C'],
    'G': ['E']
}

dfs(g, 'A')
```

[ ]:
```
def iddfs(g, start, max_depth):
    def dfs(node, depth, visited):
        if depth == 0:
            return
        print(node)
        visited.add(node)
        for neighbor in g[node]:
            if neighbor not in visited:
                dfs(neighbor, depth - 1, visited)

    for depth in range(1, max_depth + 1):
        print(f"Depth {depth}:")
        visited = set()
        dfs(start, depth, visited)
        print()

# New graph with letters as nodes
g = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'G'],
    'F': ['C'],
    'G': ['E']
}

# Start IDDFS from node 'A' with a maximum depth limit of 3
iddfs(g, 'A', 3)
```

Informed search technique(A*,Best first search)

[ ]:
```python
import heapq

def astar(graph, start, goal, h):
    queue = [(h[start], start, 0)]
    visited = set()
    while queue:
        f, node, g = heapq.heappop(queue)
        if node in visited:
            continue
        print(node)
        visited.add(node)
        if node == goal:
            break
        for neighbor, cost in graph[node]:
            if neighbor not in visited:
                new_g = g + cost
                new_f = new_g + h[neighbor]
                heapq.heappush(queue, (new_f, neighbor, new_g))

# New graph (neighbor, cost) pairs
graph = {
    'A': [('B', 2), ('C', 5)],
    'B': [('D', 3), ('E', 1)],
    'C': [('E', 2)],
    'D': [('F', 4)],
    'E': [('F', 2)],
    'F': []
}

# New heuristic function
h = {
    'A': 6,
    'B': 4,
    'C': 3,
    'D': 5,
    'E': 2,
    'F': 0
}

astar(graph, 'A', 'F', h)
```

[ ]:
```python
import heapq

def best_first_search(graph, start, goal, h):
    queue = [(h[start], start)]
    visited = set()
```

```python
    while queue:
        _, node = heapq.heappop(queue)
        if node in visited:
            continue
        print(node)
        visited.add(node)
        if node == goal:
            break
        for neighbor, _ in graph[node]:
            if neighbor not in visited:
                heapq.heappush(queue, (h[neighbor], neighbor))

# New graph (neighbor, cost) pairs
graph = {
    'A': [('B', 3), ('C', 6)],
    'B': [('D', 2), ('E', 4)],
    'C': [('E', 3)],
    'D': [('F', 5)],
    'E': [('F', 2)],
    'F': []
}

# New heuristic function
h = {
    'A': 7,
    'B': 5,
    'C': 4,
    'D': 6,
    'E': 3,
    'F': 0
}

best_first_search(graph, 'A', 'F', h)
```

Adversal search techniques(Alpha-beta,Min-Max)

[ ]:
```python
#========Min-Max Algorthim===============#
import math

def minimax (curDepth, nodeIndex,
             maxTurn, scores,
             targetDepth):


    if (curDepth == targetDepth):
        return scores[nodeIndex]
```

```
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                     True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                     True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

[ ]:

```
#===========Alpha-Beta Pruning========#

MAX, MIN = 1000, -1000


def minimax(depth, nodeIndex, maximizingPlayer,
                        values, alpha, beta):

        if depth == 3:
                return values[nodeIndex]

        if maximizingPlayer:

                best = MIN

                for i in range(0, 2):

                        val = minimax(depth + 1, nodeIndex * 2 + i,
                                                False, values, alpha, beta)
                        best = max(best, val)
                        alpha = max(alpha, best)

                        if beta <= alpha:
                                break

                return best

        else:
```

```
            best = MAX

            for i in range(0, 2):

                    val = minimax(depth + 1, nodeIndex * 2 + i,
                                                    True, values, alpha,␣
 ↪beta)
                    best = min(best, val)
                    beta = min(beta, best)

                    if beta <= alpha:
                            break

            return best

if __name__ == "__main__":

        values = [3, 5, 6, 9, 1, 2, 0, -1]
        print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```