Graphs 44

Saturday, 21 October 2023 4:48 AM

① BFS of a graph

```
def bfs(graph, v, adj):
    ans = []
    visited = [False] * len(adj)
    q = deque()
    q.append(v)
    visited[v] = True
    while q:
        curr = q.popleft()
        ans.append(curr)
        for _ in adj[curr]:
            if visited[_] == False:
                q.append(_)
                visited[_] = True
    return ans
```

② DFS of a graph

```
def dfs(graph, v, adj):
    visited = [False] * len(adj)
    ans = []
    def dfs_helper(v, adj, visited, ans):
        visited[v] = True
        ans.append(v)
        for _ in adj[v]:
            if visited[_] == False:
                dfs_helper(_, adj, visited, ans)
    dfs_helper(v, adj, visited, ans)
    return ans
```

③ Detect cycle in a directed graph

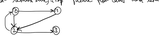
— To find a cycle in graph we can use DFS technique. It is based on idea that there is a cycle in a graph only if there is a back-edge. (x: a node, points to one of its ancestors) present in graph.

— To detect a backedge, we need to keep track of nodes visited till now and nodes that are in current recursion stack (a current path that we are visiting).

— If we reach a node that is already present in recursion stack, there is a cycle present in the graph.

note: if a graph is disconnected then get the DFS tree for each and check for a cycle in individual trees by checking back edges.

Steps:
1) Create a recursive DFS that has following — current vertex, visited array and recursion stack.
1) Mark current node as visited and also mark index in recursion stack.
2) Iterate, a loop for all vertices and for each vertex, call recursive func if it is not yet visited (to make sure that if there is a forest of graphs, we check each forest).
— In each recursion, find adjacent vertices of current vertex, then note not visited:
——If an adjacent vertex is already visited in recursion stack, return True.
——Otherwise, call recursion for the adjacent vertex.
— While returning from recursion call, remove the current node, from recursion stack, to represent that current node, is no longer in path for backedge search.
4) If any of the function returns True, stop function, print cycle and return True.



```
def isCyclic(adj, V, adj):
    visited = [False] * len(adj)
    recStack = [False] * len(adj)
    for node in range(len(adj)):
        if visited[node] == False:
            if cycle_dfs(node, visited, recStack) == True:
                return True
    return False

def cycle_dfs(node, visited, recStack):
    visited[v] = True
    recStack[v] = True
    for node in adj[v]:
        if visited[node] == False:
            if cycle_dfs(node, visited, recStack) == True:
                return True
        elif recStack[node] == True:
            return True
    recStack[v] = False
    return False
```

class solution:

```
def isCyclic(adj, V, adj):
    visited = [False] * len(adj)
    recStack = [False] * len(adj)
    for node in range(len(adj)):
        if visited[node] == False:
            if cycle_dfs(node, visited, recStack) == True:
                return True
    return False

def dfs(self, v, visited, recStack):
    visited[v] = True
    recStack[v] = True
    for node in adj[v]:
        if visited[node] == False:
            if self.dfs(node, visited, recStack) == True:
                return True
        elif recStack[node] == True:
            return True
    recStack[v] = False
    return False
```

O(V+E) time.
O(V) space.

④ Detect cycle in undirected graphs

BFS Approach

Steps: Iterate over all nodes of graph and keep visited array

1) Run BFS on given subgraph connected to current node, and pass parent of current node, in each recursion:
—— call visited[node] = T
—— Iterate over all adjacent nodes.
——If not visited, run BFS and return True if return True.
——Else, if adjacent node, is visited and not parent of current node, then return True.
——Return False.

```
def bfs(self, node, parent, visited):
    visited[node] = True
    q = deque()
    q.append(node)
    while q:
        curr = q.popleft()
        for adjNode in adj[curr]:
            if visited[adjNode] == False:
                q.append(adjNode)
                visited[adjNode] = True
                if self.bfs(adjNode, curr, visited):
                    return True
            elif adjNode != parent:
                return True
    return False

def isCyclic(self, V, adj):
    vis = [0]*V
    for i in range(V):
        if visited[i] == 0:
            if self.bfs(i, -1, vis):
                return True
    return False
```

DFS Approach

We also use DFS traversal. For every node, we have to check nodes such that it is already visited and current node is not parent, then there is a cycle.

— We check through each node in adjacent list, nor any visited.

```
def bfs(self, node, parent, vis):
    vis[node] = 1
    q = deque()
    q.append(node)
    while q:
        curr = q.popleft()
        for adjNode in adj[curr]:
            if vis[adjNode] == 0:
                q.append(adjNode)
                vis[adjNode] = 1
                if self.bfs(adjNode, curr, vis):
                    return True
            elif adjNode != parent:
                return True
    return False

def isCyclic(self, V, adj):
    vis = [0]*V
    for i in range(V):
        if vis[i] == 0:
            if self.bfs(i, -1, vis):
                return True
    return False
```

⑤ Topological Sort

Steps:
1) Create a recursive func.
2) Mark current node as visited.
3) Traverse all adjacent nodes and, if any node is visited array then True.
3) If any adjacent node is cycle, then call recursive func for that node.
If it returns True, then return True.
5) If no adjacent node is cycle or has not adjacent then, mark current node as BLACK and return False.

```
def isCyclic(self, V, adj):
    color = [0]*len(adj)
    for node in range(len(adj)):
        if color[node] == 0:
            if self.cycle_dfs(node, adj, color) == True:
                return True
    return False

def cycle_dfs(self, v, adj, color):
    color[v] = 1
    for node in adj[v]:
        if color[node] == 1:
            return True
        elif color[node] == 0:
            if self.cycle_dfs(node, adj, color) == True:
                return True
    color[v] = 2
    return False
```

O(V) space.
O(V+E) time.

⑥ Topological Sort (DFS)

```
def topologicalSort(self, V, adj):
    stack = []
    visited = [False] * len(adj)
    for node in range(len(adj)):
        if visited[node] == False:
            self.dfs(node, visited, stack)
    return stack[::-1]

def dfs(self, v, visited, stack):
    visited[v] = True
    for node in adj[v]:
        if visited[node] == False:
            self.dfs(node, visited, stack)
    stack.append(v)
```

O(V+E) time.
O(V) space.

⑦ Flood-fill Algorithm

Using recursion

Idea is simple, we replace color of current pixel, the seed pixel, with a corresponding pixel.

floodfill(screen[M][N], x, y, prev, new):
1) If x, y is outside, the screen return
2) If color of screen[x][y] is not same as prev, then return
3) Recur for north, south, east and west:

```
floodfill(screen, x-1, y, prev, new)
floodfill(screen, x+1, y, prev, new)
floodfill(screen, x, y-1, prev, new)
floodfill(screen, x, y+1, prev, new)
```

```
def floodFill(self, image, sr, sc, newColor):
    def helper(x, y, prev, new):
        if x < 0 or x >= len(image) or
            y < 0 or y >= len(image[0]) or
            image[x][y] != prev:
            return
        image[x][y] = new
        helper(x-1, y, prev, new)
        helper(x+1, y, prev, new)
        helper(x, y-1, prev, new)
        helper(x, y+1, prev, new)
    prev = image[sr][sc]
    helper(sr, sc, prev, new)
    return image
```

O(image) time

O(image) space

Backtrack check, checked done to recursion

⑧ Flood-fill Algorithm

Using BFS Approach

1. Create a queue, and enqueue color of source pixel, the seed pixel.
2. Create another queue.
3. While queue is not empty, dequeue the seed pixel, then return it to queue.
4. Check the four directions, to find if any of them are same as the seed pixel, if yes, then enqueue the pixel into queue, and make it as old color.
5. Keep doing so until we find the target and we have traversed all nodes.

```
def floodFill(self, image, r, c, newColor):
    def bfs(x, y):
        if x < 0 or x >= len(image) or
            y < 0 or y >= len(image[0]):
            return False
        if image[x][y] == newColor:
            return True
        if image[x][y] != oldColor:
            return False
        image[x][y] = newColor
        return True
    oldColor = image[r][c]
    q = deque()
    q.append((r, c))
    while q:
        x, y = q.popleft()
        if bfs(x, y):
            for i in range(-1, 2):
                for j in range(-1, 2):
                    if i == 0 and j == 0:
                        continue
                    if 0 <= x+i < len(image) and
                        0 <= y+j < len(image[0]):
                        q.append((x+i, y+j))
    return image
```

⑨ Clone an Undirected Graph

— DFS traversal: for the graph and while visiting a node, make a clone node of it (a copy of original node). If a node has connections which is already visited then do nothing for a clone node.

—— Maintaining a mapping to maintain old node which have already been made.
Key clone (reference) address of original node.
Value clone → " " of clone node.

—— Visit to connect clone nodes:
While visiting neighboring nodes of a node if get corresponding clone node then, take out clone/clone1. Visit all neighboring nodes for a node and for each neighbor find corresponding clone node (if not clone node yet) and then add this neighboring node of clone/clone1.

```
def cloneGraph(self, node):
    m = {}
    def dfs(cur):
        if cur in m:
            return m[cur]
        clone = Node(cur.val)
        m[cur] = clone
        for nei in cur.neighbors:
            clone.neighbors.append(dfs(nei))
        return clone
    return dfs(node) if node else None
```

```
def cloneGraph(self, node):
    if node:
        clone = {node: Node(node.val)}
        q = [node]
        while q:
            curr = q.pop(0)
            for nei in curr.neighbors:
                if nei not in clone:
                    clone[nei] = Node(nei.val)
                    q.append(nei)
                clone[curr].neighbors.append(clone[nei])
        return clone[node]
    return None
```

Time: O(V+E)
Space: O(V)

⑩ Word Ladder (length of shortest chain to reach a target word)

— Idea is to use BFS. To find shortest path through BFS, start from start word and push into queue. And now, target is found by that time, return level of BFS.

— In each step of BFS, we can get all words that can be formed step from many steps.

1. Load from given word list
2. Put word in queue.
3. Run a loop until queue is empty.
4. Traverse all words that differ by one character in it and push to queue.
5. Keep doing so until we find the target and we have traversed all words.

```
def ladderLength(self, beginWord, endWord):
    s = set(wordList)
    if endWord not in s:
        return 0
    s.remove(endWord)
    q = deque()
    q.append((beginWord, 1))
    while q:
        word, step = q.popleft()
        if word == endWord:
            return step
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                if word[i] != c:
                    new_word = word[:i] + c + word[i+1:]
                    if new_word in s:
                        s.remove(new_word)
                        q.append((new_word, step + 1))
    return 0
```

```
def ladderLength(self, beginWord, endWord):
    if endWord not in wordList:
        return 0
    wordSet = set(wordList)
    queue = [(beginWord, 1)]
    while queue:
        word, level = queue.pop(0)
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                if word[i] != c:
                    new_word = word[:i] + c + word[i+1:]
                    if new_word == endWord:
                        return level + 1
                    if new_word in wordSet:
                        wordSet.remove(new_word)
                        queue.append((new_word, level + 1))
    return 0
```