National University of Computer and Emerging Sciences

Artificial Intelligence

Lab Activity

Fast School of Computing

FAST-NU, Lahore, Pakistan

Objectives

- Breadth First Search (BFS)
- Depth First Search (DFS)
- Using DFS & BFS to find # of islands

Your Task: Understand and test the following problems then propose your own solutions to finding # of islands problem using BFS & DFS.

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all the vertices of a graph in breadth-first order, i.e., it explores all the vertices at the present depth before moving on to the vertices at the next depth level. It uses a queue data structure to keep track of vertices to be explored.

Key Points:

- BFS starts traversal from the root node (or any arbitrary node in case of a graph) and visits nodes in a level by level manner.
- It visits all the nodes of a level before going to the next level.
- BFS uses a queue data structure for traversal.

Depth-First Search (DFS)

Depth-First Search (DFS) is another graph traversal algorithm that explores all the vertices of a graph in depth-first order, i.e., it explores as far as possible along each branch before backtracking. It uses a stack data structure, either explicitly or implicitly through recursion, to keep track of vertices to be explored.

Key Points:

- DFS starts traversal from the root (or any arbitrary node in case of a graph) and explores as far as possible along each branch before backtracking.
- DFS uses a stack data structure for traversal, either explicitly or implicitly through recursion.

Both BFS and DFS have their own use cases and are used in different scenarios based on the problem at hand. For instance, BFS can be used to find the shortest path in a 2D grid, while DFS can be used to traverse trees, graphs, and more. It's important to understand the properties of both and how to implement them to solve a wide range of problems. Happy learning!

Finding the number of islands

Given a binary 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 4 islands.

Example:

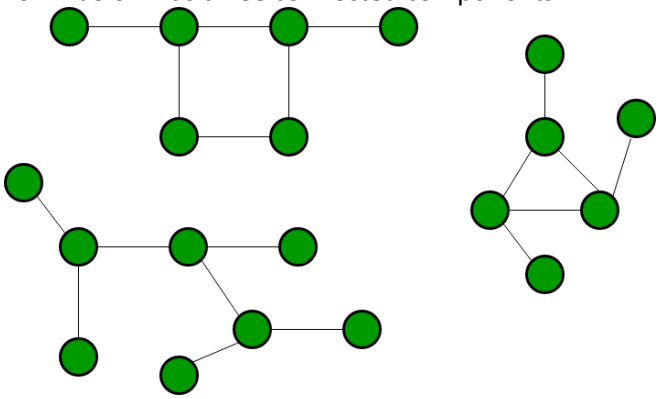
Input: Mat[][]= {{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 0}}

Output: 4

This is a variation of the standard problem: "Counting the number of connected components in an undirected graph".

Before we go to the problem, let us understand what is a connected component. A connected component of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.

For example, the graph shown below has three connected components.



A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph. Such a graph with only one connected component is called a Strongly Connected Graph.

This problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?

A group of connected 1s forms an island. For example, the below matrix contains 4 islands

```
{ { 1 , 1, 0, 0, 0 },
  { 0, 1, 0, 0, 1 },
  { 1 , 0, 0, 1, 1 },
  { 0 , 0, 0, 0, 0 },
  { 1, 0, 1, 1, 0 } }
```

Finding the number of islands using an additional Matrix:

The idea is to keep an additional matrix to keep track of the visited nodes in the given matrix, and perform DFS to find the total number of islands

Follow the steps below to solve the problem:

- Initialize a boolean matrix visited of the size of the given matrix to false.
- Initialize count = 0, to store the answer.
- Traverse a loop from 0 till ROW
  - Traverse a nested loop from 0 to COL
    - If the value of the current cell in the given matrix is 1 and is not visited
      - Call DFS function
        - Initialize rowNbr[] = { -1, -1, -1, 0, 0, 1, 1, 1 } and colNbr[] = { -1, 0, 1, -1, 1, -1, 0, 1 } for the neighbour cells.
        - Mark the current cell as visited
        - Run a loop from 0 till 8 to traverse the neighbor
          - If the neighbor is safe to visit and is not visited
            - Call DFS recursively on the neighbor.
      - Increment count by 1
- Return count as the final answer.

Below is the code implementation of the above approach:

```
# Program to count islands in boolean 2D matrix
class Graph:

	def __init__(self, row, col, g):
		self.ROW = row
		self.COL = col
		self.graph = g

	# A function to check if a given cell
	# (row, col) can be included in DFS
	def isSafe(self, i, j, visited):
		# row number is in range, column number
		# is in range and value is 1
		# and not yet visited
		return (i >= 0 and i < self.ROW and
				j >= 0 and j < self.COL and
				not visited[i][j] and self.graph[i][j])
```

```
# A utility function to do DFS for a 2D
# boolean matrix. It only considers
# the 8 neighbours as adjacent vertices

def DFS(self, i, j, visited):

        # These arrays are used to get row and
        # column numbers of 8 neighbours
        # of a given cell
        rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1]
        colNbr = [-1, 0, 1, -1, 1, -1, 0, 1]

        # Mark this cell as visited
        visited[i][j] = True

        # Recur for all connected neighbours
        for k in range(8):
                if self.isSafe(i + rowNbr[k], j + colNbr[k], visited):
                        self.DFS(i + rowNbr[k], j + colNbr[k], visited)

# The main function that returns
# count of islands in a given boolean
# 2D matrix

def countIslands(self):
        # Make a bool array to mark visited cells.
        # Initially all cells are unvisited
        visited = [[False for j in range(self.COL)]for i in range(self.ROW)]

        # Initialize count as 0 and traverse
        # through the all cells of
        # given matrix
        count = 0
        for i in range(self.ROW):
                for j in range(self.COL):
                        # If a cell with value 1 is not visited yet,
                        # then new island found
                        if visited[i][j] == False and self.graph[i][j] == 1:
                                # Visit all cells in this island
                                # and increment island count
                                self.DFS(i, j, visited)
                                count += 1
```

```
                return count


graph = [[1, 1, 0, 0, 0],
                [0, 1, 0, 0, 1],
                [1, 0, 0, 1, 1],
                [0, 0, 0, 0, 0],
                [1, 0, 1, 0, 1]]


row = len(graph)
col = len(graph[0])

g = Graph(row, col, graph)

print("Number of islands is:")
print(g.countIslands())
```

Output

```
Number of islands is: 5
```

Time complexity: O(ROW x COL), where ROW is the number of rows and COL is the number of columns in the given matrix.
Auxiliary Space: O(ROW x COL), for creating an additional visited matrix.

Finding the number of islands using DFS:

The idea is to modify the given matrix, and perform DFS to find the total number of islands

Follow the steps below to solve the problem:

- Initialize count = 0, to store the answer.
- Traverse a loop from 0 till ROW
  - Traverse a nested loop from 0 to COL
    - If the value of the current cell in the given matrix is 1
      - Increment count by 1
      - Call DFS function
        - If the cell exceeds the boundary or the value at the current cell is 0
          - Return.
        - Update the value at the current cell as 0.
        - Call DFS on the neighbor recursively
- Return count as the final answer.

Below is the code implementation of the above approach:

```
# Program to count islands in boolean 2D matrix
class Graph:
```

```
def __init__(self, row, col, graph):
        self.ROW = row
        self.COL = col
        self.graph = graph

# A utility function to do DFS for a 2D
# boolean matrix. It only considers
# the 8 neighbours as adjacent vertices
def DFS(self, i, j):
        if i < 0 or i >= len(self.graph) or j < 0 or j >= len(self.graph[0]) or self.graph[i][j] != 1:
                return

        # mark it as visited
        self.graph[i][j] = -1

        # Recur for 8 neighbours
        self.DFS(i - 1, j - 1)
        self.DFS(i - 1, j)
        self.DFS(i - 1, j + 1)
        self.DFS(i, j - 1)
        self.DFS(i, j + 1)
        self.DFS(i + 1, j - 1)
        self.DFS(i + 1, j)
        self.DFS(i + 1, j + 1)

# The main function that returns
# count of islands in a given boolean
# 2D matrix
def countIslands(self):
        # Initialize count as 0 and traverse
        # through the all cells of
        # given matrix
        count = 0
        for i in range(self.ROW):
                for j in range(self.COL):
                        # If a cell with value 1 is not visited yet,
                        # then new island found
                        if self.graph[i][j] == 1:
                                # Visit all cells in this island
                                # and increment island count
                                self.DFS(i, j)
                                count += 1
```

```
                    return count


graph = [
          [1, 1, 0, 0, 0],
          [0, 1, 0, 0, 1],
          [1, 0, 0, 1, 1],
          [0, 0, 0, 0, 0],
          [1, 0, 1, 0, 1]
]


row = len(graph)
col = len(graph[0])

g = Graph(row, col, graph)

print("Number of islands is:", g.countIslands())
```

Output

```
Number of islands is: 5
```

Time complexity: O(ROW x COL), where ROW is the number of rows and COL is the number of columns in the given matrix.
Auxiliary Space: O(ROW * COL), as to do DFS we need extra auxiliary stack space

Finding the number of islands using DFS:

A DFS solution for islands is already discussed. This problem can also solved by applying BFS() on each component. In each BFS() call, a component or a sub-graph is visited. We will call BFS on the next un-visited component. The number of calls to BFS() gives the number of connected components.

A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard BFS(), where we process all adjacent vertices, we process 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

Algorithm:

1. Initialize a boolean matrix of the same size as the given matrix to keep track of visited cells.
2. Traverse the given matrix, and for each unvisited cell that is part of an island, perform BFS starting from that cell.
3. In the BFS algorithm, enqueue the current cell and mark it as visited. Then, while the queue is not empty, dequeue a cell and enqueue its unvisited neighbors that are part of the same island. Mark each of these neighbors as visited.
4. After BFS is complete, increment the island count by 1.
5. Repeat steps 2-4 until all unvisited cells have been processed.
6. Return the total island count.

Below is the code implementation of the above approach:

```
# A BFS based solution to count number of
# islands in a graph.
from collections import deque

# A function to check if a given cell
# (u, v) can be included in DFS
def isSafe(mat, i, j, vis):

	return ((i >= 0) and (i < 5) and
				(j >= 0) and (j < 5) and
		(mat[i][j] and (not vis[i][j])))

def BFS(mat, vis, si, sj):

	# These arrays are used to get row and
	# column numbers of 8 neighbours of
	# a given cell
	row = [-1, -1, -1, 0, 0, 1, 1, 1]
	col = [-1, 0, 1, -1, 1, -1, 0, 1]

	# Simple BFS first step, we enqueue
	# source and mark it as visited
	q = deque()
	q.append([si, sj])
	vis[si][sj] = True

	# Next step of BFS. We take out
	# items one by one from queue and
	# enqueue their unvisited adjacent
	while (len(q) > 0):
		temp = q.popleft()

		i = temp[0]
		j = temp[1]

		# Go through all 8 adjacent
		for k in range(8):
			if (isSafe(mat, i + row[k], j + col[k], vis)):
				vis[i + row[k]][j + col[k]] = True
				q.append([i + row[k], j + col[k]])

# This function returns number islands (connected
# components) in a graph. It simply works as
```

```
# BFS for disconnected graph and returns count
# of BFS calls.
def countIslands(mat):

	# Mark all cells as not visited
	vis = [[False for i in range(5)]
					for i in range(5)]
	# memset(vis, 0, sizeof(vis));

	# 5all BFS for every unvisited vertex
	# Whenever we see an univisted vertex,
	# we increment res (number of islands)
	# also.
	res = 0

	for i in range(5):
		for j in range(5):
			if (mat[i][j] and not vis[i][j]):
				BFS(mat, vis, i, j)
				res += 1

	return res

# Driver code
if __name__ == '__main__':

	mat = [ [ 1, 1, 0, 0, 0 ],
			[ 0, 1, 0, 0, 1 ],
			[ 1, 0, 0, 1, 1 ],
			[ 0, 0, 0, 0, 0 ],
			[ 1, 0, 1, 0, 1 ]]

	print ("Number of islands is:",countIslands(mat))
```

Output

```
Number of islands is: 5
```

Time complexity: O(ROW * COL) where ROW is the number of ROWS and COL is the number of COLUMNS in the matrix.
Auxiliary Space: O(ROW * COL) because of the visited array.