Matrix Max Sum Path with DFS

#algorithms/firecode/matrix
#algorithms/firecode/dfs

Given an m x n matrix (2d list) filled with non-negative integers, use depth first search to find the maximum sum along a path from the top-left of the grid to the bottom-right. Return this maximum sum. The direction of movement is limited to right and down.

Example:

```
Input Matrix :

    1 2 3
    4 5 6
    7 8 9

Output  : 1 + 4 + 7 + 8 + 9 = 29
```

Note: This problem has a more efficient solution based on Dynamic Programming techniques. We'll be exploring those in future problems - so don't fret just yet!

Solution O(v + e)

```python
class Node:
    def __init__(self, total, i, j):
        self.total = total
        self.i = i
        self.j = j

# Collections module has already been imported.
def matrix_max_sum_dfs(grid):
    if not grid or not grid[0]: return 0
    downs = len(grid) - 1
    rights = len(grid[0]) - 1
    stack = [Node(grid[0][0], 0, 0)]
    maxx = 0
    while stack:
        node = stack.pop()
        if node.i < downs:
            stack.append(Node(node.total+grid[node.i+1][node.j], node.i+1,
node.j))
        if node.j < rights:
            stack.append(Node(node.total+grid[node.i][node.j+1], node.i,
node.j+1))
        if node.i == downs and node.j == rights:
            maxx = max(node.total, maxx)
    return maxx
```

Flip it!

#algorithms/firecode/matrix

You are given an m x n 2D image matrix (`List of Lists`) where each integer represents a pixel. Flip it in-place along its vertical axis.

Example:

```
Input image :
1 0
1 0

Modified to :
0 1
0 1
```

Solution -- O(n^2)

```
def flip_vertical_axis(matrix):
    m = len(matrix)
    n = len(matrix[0])
    if n < 2:
        return matrix
    for i in range(m):
        for j in range(n//2):
            temp = matrix[i][n-1-j]
            matrix[i][n-1-j] = matrix[i][j]
            matrix[i][j] = temp
    return temp
```

Find the Transpose of a Square Matrix

#algorithms/firecode/matrix

You are given a square 2D image matrix (`List of Lists`) where each integer represents a pixel. Write a method `transpose_matrix` to transform the matrix into its Transpose - in-place. The transpose of a matrix is a matrix which is formed by turning all the rows of the source matrix into columns and vice-versa.

Example:

```
Input image :
1 0
1 0

Modified to:
1 1
0 0
```

Solution

```
def transpose_matrix(matrix):
    m = len(matrix)
    n = len(matrix[0])
    for i in range(m):
        for j in range(n-i):
            matrix[i][j+i], matrix[j+i][i] = matrix[j+i][i], matrix[i][j+i]
```

Horizontal Flip

#algorithms/firecode/matrix

You are given an m x n 2D image matrix (`List of Lists`) where each integer represents a pixel. Flip it in-place along its horizontal axis.

Example:

```


Input image :
              1 1
              0 0
Modified to :
              0 0
              1 1
```

Solution -- O(m)

```
def flip_horizontal_axis(matrix):
    for i in range(len(matrix)//2):
        row = matrix[i]
        matrix[i], matrix[len(matrix)-1-i] = matrix[len(matrix)-1-i], matrix[i]
```

Dijsktra Shortest Path

#algorithms/graphs/bfs

The algorithm we are going to use to determine the shortest path is called "Dijkstra's algorithm." Dijkstra's algorithm is an iterative algorithm that provides us with the shortest path from one particular starting node to all other nodes in the graph. Again this is similar to the results of a breadth first search.

Solution

```
from collections import defaultdict
class Graph:
  def init(self):
    self.nodes = set() # set object
    self.edges = defaultdict(list)
    self.distances = {}

def add_nodes(self, value):
    for i in value:
        self.nodes.add(i) # add element into set

def add_edge(self, from_node, to_node, distance):
    self.edges[from_node].append(to_node)
    self.edges[to_node].append(from_node) # dict to neighbour nodes
    self.distances[(from_node, to_node)] = distance # dict for distance
    self.distances[(to_node, from_node)] = distance
```

```
def dijsktra(graph, initial):
    visited = {initial: 0}
    path = {}

    nodes = set(graph.nodes)

    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node

        if min_node is None:
              break

        nodes.remove(min_node)
        current_weight = visited[min_node]

        for edge in graph.edges[min_node]:
            weight = current_weight + graph.distances[(min_node, edge)]
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge] = min_node

    return visited, path
```

```

g = Graph()
g.add_nodes([i+1 for i in range(8)])
g.add_edge(1, 2, 4)
g.add_edge(1, 3, 1)
g.add_edge(2, 4, 3)
g.add_edge(2, 5, 7)
g.add_edge(4, 8, 3)
g.add_edge(5, 8, 4)
g.add_edge(3, 6, 3)
g.add_edge(3, 7, 2)
g.add_edge(6, 7, 1)
print "nodes:", g.nodes
print "edges:", g.edges
print "distances: ", g.distances
```

Rotate a Square Image Counterclockwise

#algorithms/firecode/matrix

You are given a square 2D image matrix (List of Lists) where each integer represents a pixel. Write a method `rotate_square_image_ccw` to rotate the image counterclockwise - in-place. This problem can be broken down into simpler sub-problems you've already solved earlier! Rotating an image counterclockwise can be achieved by taking the transpose of the image matrix and then flipping it on its horizontal axis.

Example:

```
Input image :
1 0
1 0
Modified to :
0 0
1 1
```

Solution

```python
def rotate_square_image_ccw(matrix):
    for i in range(len(matrix)):
        for j in range(i):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
    for i in range(len(matrix)//2):
        matrix[i], matrix[len(matrix)-1-i] = matrix[len(matrix)-1-i], matrix[i]
```

Snake

#algorithms/firecode/matrix

Let's have some fun with 2D Matrices! Write a method `find_spiral` to traverse a 2D matrix (List of Lists) of `int` s in a clockwise spiral order and append the elements to an output

`List` of `Integer` s.

Example:

```
Input Matrix :
    [1, 2, 3]
    [4, 5, 6]
    [7, 8, 9]

Output List:[1, 2, 3, 6, 9, 8, 7, 4, 5]
```

Solution -- O(n*m)

```
def find_spiral(matrix):
    output = []
    if not matrix or not matrix[0]: return output
    m = len(matrix)
    n = len(matrix[0])
    rounds = max(m, n)//2 + max(m,n) % 2

    for k in xrange(rounds):
        for j in xrange(k, n-k):
            output.append(matrix[k][j])
        if k == n-1-k:
            return output
        for i in xrange(k+1, m-k):
            output.append(matrix[i][n-1-k])
        if k == m-1-k:
            return output
        for j in reversed(xrange(k, n-1-k)):
            output.append(matrix[m-1-k][j])
        for i in reversed(xrange(k+1, m-1-k)):
            output.append(matrix[i][k])

    return output
```

Recovering IPv4 Addresses

#algorithms/dfs

You are given a `String` containing at least 4 numbers that represented an IPv4 Address, but the separator data - i.e. the dots that separate each Byte in a 4 Byte Ipv4 address, has been lost. Write a method - `generate_ip_address` that takes in this `String` and returns a `List` of strings containing all possible IPv4 Addresses that can be generated from the given sequence of decimal integers.

![](https://www.firecode.io/assets/problems/ipv4.png)
Note:

<small>- The IP Addresses for this problem are written in the decimal dot notation.<br>&nbsp;- You must use all the digits in the input String<br>&nbsp;- The order in which the IP Addresses are returned does not matter<br>&nbsp;- 0.0.0.1 and 0.0.0.01 may be considered 2 distinct possibilities. i.e. do not ignore leading or trailing 0s.</small>

Examples:

`generate_ip_address("0001")` ==> `["0.0.0.1"]`

`generate_ip_address("0010")` ==> `["0.0.1.0"]`

`generate_ip_address("25525511135")` ==> `["255.255.11.135", "255.255.111.35"]`

Solution

```
class Node:
    # Constructor
    def __init__(self, prev, next, level):
        self.prev = prev
        self.next = next
        self.level = level

def generate_ip_address(input):
    # Return type should be a List.
    if len(input)/4 > 12 or len(input) < 4:
        return None
    ips = []
    stack = [Node("", input, 0)]

    while stack:
        node = stack.pop()
        if node.level < 4:
            if node.next is "":
                continue
            elif len(node.next) < 3:
                index = len(node.next)
            else:
                index = 3 if int(node.next[:3]) < 256 else 2
            for i in xrange(index):
                num = node.next[:i+1]
                rest = node.next[i+1:]
                stack.append(Node(node.prev+"."+num, rest, node.level + 1))
        elif node.next is "":
            ips.append(node.prev[1:])

    return ips
```

Power of 4

#algorithms/firecode

Write a method to determine whether a given integer (zero or positive number) is a power of 4 without using the multiply or divide operator. If it is, return `True`, else return `False`.

Examples:
`is_power_of_four(5)` ==> `False`

`is_power_of_four(16)` ==> `True`

Solution -- O(n) time; O(1) space

```python
def is_power_of_four(number):
    num = 1
    while num < number:
        num <<= 2
    return num == number
```

Merge Sort

#algorithms/sorting

Recursive -- O(nlogn)

```python
def merge(left, right):
   if not len(left) or not len(right):
       return left or right

   result = []
   i, j = 0, 0
   while (len(result) < len(left) + len(right)):
       if left[i] < right[j]:
           result.append(left[i])
           i+= 1
       else:
           result.append(right[j])
           j+= 1
       if i == len(left) or j == len(right):
           result.extend(left[i:] or right[j:])
           break

   return result

def mergesort(list):
   if len(list) < 2:
       return list
```

```
  middle = len(list)/2
  left = mergesort(list[:middle])
  right = mergesort(list[middle:])

  return merge(left, right)

if __name__ == "__main__":
  print mergesort([3,4,5,1,2,8,3,7,6])
```

Iterative -- O(nlogn)

```
def merge_sort_iterative(data):
  """ gets the data using merge sort and returns sorted."""

  for j in range(1, len(data)):
    j *= 2
    for i in range(0,len(data),j):
      data_1 = data[i:i+(j/2)]
      data_2 = data[i+(j/2):j-i]
      l = m = 0
      while l < len(data_1) and m < len(data_2):
        if data_1[l] < data_2[m]:
          m += 1
        elif data_1[l] > data_2[m]:
          data_1[l], data_2[m] = data_2[m], data_1[l]
          l += 1
      data[i:i+(j/2)], data[i+(j/2):j-i] = data_1, data_2

  return data
```

Binary Search Tree Traversal

#algorithms/bst/traversal

Recursive

Preorder

```
def printPreorder(root):
    if root:
        print(root.val),
        printPreorder(root.left)
        printPreorder(root.right)
```

Inorder

```
def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val),
        printInorder(root.right)
```

Postorder

```
def printPostorder(root):
    if root:
        printPostorder(root.left)
        printPostorder(root.right)
        print(root.val),
```

Iterative

Preorder

```
def iterativePreorder(root):

    # Base CAse
    if root is None:
        return

    # create an empty stack and push root to it
    stack = [root]
    #  Pop all items one by one. Do following for every popped item
    #   a) print it
    #   b) push its right child
    #   c) push its left child
    # Note that right child is pushed first so that left
    # is processed first */
    while stack:

        # Pop the top item from stack and print it
        node = stack()
        print node.data,

        # Push right and left children of the popped node
        # to stack
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```

Inorder

```

# Iterative function for inorder tree traversal
def inOrder(root):
    # Set current to root of binary tree
    current = root
    stack = [] # initialze stack
    while stack and current:
        # Reach the left most Node of the current Node
        if current:
            # Place pointer to a tree node on the stack
            # before traversing the node's left subtree
            stack.append(current)
            current = current.left
        # BackTrack from the empty subtree and visit the Node
        # at the top of the stack; however, if the stack is
        # empty you are done
        else:
            if stack:
                current = s.pop()
                print current.data,
                # We have visited the node and its left
                # subtree. Now, it's right subtree's turn
                current = current.right
```

Postorder with two stacks

```
def postOrderIterative(root):
    if root is None: return
    # Create two stacks
    stack_1 = []
    stack_2 = []
    # Push root to first stack
    stack_1.append(root)
    # Run while first stack is not empty
    while stack_1:
        # Pop an item from s1 and append it to s2
        node = stack_1.pop()
        stack_2.append(node)
        # Push left and right children of removed item to s1
        if node.left:
            stack_1.append(node.left)
        if node.right:
            stack_2.append(node.right)
        # Print all elements of second stack
    while stack_2:
        node = stack_2.pop()
        print node.data,
```

Postorder with one stack

```
def peek(stack):
    if stack:
        return stack[-1] # the last item
    return None
# A iterative function to do postorder traversal of
# a given binary tree
def postOrderIterative(root):
    # Check for empty tree
    if not root: return
    stack = []
    while(True):
        while (root):
             # Push root's right child and then root to stack
             if root.right:
                stack.append(root.right)
             stack.append(root)
             # Set root as root's left child
             root = root.left
        # Pop an item from stack and set it as root
        root = stack.pop()
        # If the popped item has a right child and the
        # right child is not processed yet, then make sure
        # right child is processed before root
        if (root.right and
            peek(stack) == root.right):
            stack.pop() # Remove right child from stack
            stack.append(root) # Push root back to stack
            root = root.right # change root so that the
                                 # righ childis processed next
        # Else print root's data and set root as None
        else:
            ans.append(root.data)
            root = None
        if not stack:
                break
```

Graph Traversal

#algorithms/graphs/traversal

BFS

```
# Program to print BFS traversal from a given source
```

```python
# vertex. BFS(int s) traverses vertices reachable
# from s.
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:
    # Constructor
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)
    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
    # Function to print a BFS of graph
    def BFS(self, s):
        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))
        # Create a queue for BFS
        queue = []
        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True
        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print s,
            # Get all adjacent vertices of the dequeued
            # vertex s. If a adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True



# Driver code
# Create a graph given in the above diagram
g = Graph()
```

```
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
```

Applications of BFS

We have earlier discussed Breadth First Traversal Algorithm for Graphs. We have also discussed Applications of Depth First Traversal . In this article, applications of Breadth First Search are discussed.

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent , Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm . Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

*8) * Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford-Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).

*10) * To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structure similar to Breadth First Search.

DFS

```
# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:
    # Constructor
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)
    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
    # A function used by DFS
    def DFSUtil(self,v,visited):
        # Mark the current node as visited and print it
        visited[v]= True
        print v,
        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)
    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self,v):
        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))
        # Call the recursive helper function to print
        # DFS traversal
        self.DFSUtil(v,visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)


print "Following is DFS from (starting from vertex 2)"
g.DFS(2)
```

Applications of BFS

1)For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the
contents of the stack

See this for details.

4) Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) *Finding Strongly Connected Components of a graph*. A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See_ this_ for DFS based algo for finding Strongly Connected Components)
7) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Triple Sum

#algorithms/firecode/level4

Description

Given a sorted `list` of integers, find all the unique triplets which sum up to the given target.

Note: Each triplet must be a tuple having elements (input[i], input[j], input[k]), such that i < j < k. The ordering of unique triplets within the output `list` does not matter.

```
Input : [1,2,3,4,5,6,7]
Target: 15
Output: [(2, 6, 7), (3, 5, 7), (4, 5, 6)]
```

Answer -- O(n^2) time

```python
def triple_sum(integer_list, target_number):
    answer = []
    if len(integer_list) < 3:
        return answer
    for i in xrange(len(integer_list)-2):
        j = i + 1
        k = len(integer_list)-1
        while j < k:
            summ = integer_list[i] + integer_list[j] + integer_list[k]
            if summ == target_number:
                add = (integer_list[i], integer_list[j], integer_list[k])
                if not answer or add != answer[-1]:
                    answer.append(add)
                j += 1
                k -= 1
            elif summ < target_number:
                j += 1
            else:
                k -= 1
    return answer
```