COMP9312 Advanced Graph Traversal

Outline

- Graph traversal
- Complex graph structure
- Disjoint set algorithm
- Connected Components
- Efficiency on Medium Dataset

Exercise 1: Graph Traversal

- Breadth-first traversal

  Considering the implementation of a breadth-first traversal on a graph:
  - Choose any vertex, mark it as visited and push it onto queue
  - While the queue is not empty
    - Pop the top vertex v from the queue
    - For each vertex adjacent to v that has not been visited:
      - Mark it as visited
      - Push it onto the queue

It continues until the queue is empty
Note: if there are no unvisited vertices, the graph is connected.

The size of the queue is at most $O(|V|)$

```python
# BFS
# A bad implementation
def BFS(u):
    visited = [False] * n
    queue = []
    # q = deque()

    queue.append(u)
    visited[u] = True
    while queue:
        s = queue.pop(0)
        # this line shift all items forward
        # q.popleft()

        print(s)
        for i in range(offset[s],offset[s+1]):
            nbr_of_s = csr_edges[i]
            if visited[nbr_of_s]: continue
            queue.append(nbr_of_s)
            # q.append()
            visited[nbr_of_s] = True
```

Exercise 1: Graph Traversal

- Depth-first traversal

  Considering the implementation of a depth-first traversal on a graph:
  - Choose any vertex, mark it as visited
  - From that vertex:
    - If find an unvisited adjacent vertex not visited yet, move to that vertex
    - Otherwise, go back to the last vertex that still has unvisited adjacent vertices.
  - Continue until there are no visited vertices with unvisited adjacent vertices.

  Two implementation
  - Recursive approach (a statement in a function calls itself repeatedly.)
  - Iterative approach (a loop executes repeatedly until the controlling condition becomes false)

Exercise 1: Graph Traversal

- Depth-first traversal
  - Recursive approach (a statement in a function calls itself repeatedly.)
  - Iterative approach (a loop executes repeatedly until the controlling condition becomes false)

```python
# DFS recursive
visited = [False] * n
def DFS_recursive(u):
    print(u)
    visited[u] = True
    for i in range(offset[u],offset[u+1]):
        nbr_of_u = csr_edges[i]
        if visited[nbr_of_u]: continue
        DFS_recursive(nbr_of_u)
```

```python
# DFS iterative
def DFS_iterative(u):
    visited = [False] * n
    stack = []
    stack.append(u)

    while (len(stack)):
        s = stack.pop()
        if(visited[u]):
            continue;

        visited[u] = True
        for i in range(offset[s],offset[s+1]):
            nbr_of_s = csr_edges[i]
            if visited[nbr_of_s]: continue
            stack.append(nbr_of_s)
```

Exercise 1: Graph Traversal

- Implement BFS(G, v) and DFS(G, v)
- Load the graph in Figure 1 and check the connectivity from the following node pairs: (A, B), (A, C), (A, D), (A, E).

  For example, using BFS: if the target vertex is in the traversal, then it is connected.

Now, you have 10 minutes to implement Q1.

Exercise 2: Complex Graph Structures

- Change the definition of class `SimpleGraph to Directed weighted graph` by yourself. The new class should support the load of directed weighted graphs and allow self-loop and multiple edges between two vertices.

```python
class DirectedWeightedGraph(object):
    def __init__(self, edge_list):
        self.vertex_dict = {}
        self.adj_list_in = []
        self.adj_list_out = []
        self.vertex_num = 0
        for [src, dst, weight] in edge_list:
            self.add_edge(src, dst, weight)

    def add_vertex(self, name):
        id = self.vertex_num
        self.vertex_dict[name] = id
        self.vertex_num += 1
        self.adj_list_in.append(list())
        self.adj_list_out.append(list())

    def add_edge(self, vertex1, vertex2, weight):
        if vertex1 not in self.vertex_dict.keys():
            self.add_vertex(vertex1)
        if vertex2 not in self.vertex_dict.keys():
            self.add_vertex(vertex2)
        self.adj_list_out[self.vertex_dict[vertex1]].append([vertex2, weight])
        self.adj_list_in[self.vertex_dict[vertex2]].append([vertex1, weight])
```

```python
def sumIndegree(G, u):
    # compute and print the sum of the in-degree of each vertex in G
    return
```

Exercise 2: Complex Graph Structures

- Implement a function, in which when inputting any vertex *v*, it outputs the sum of the weights of all the indegree edges of *v*. Please refer to the function **sumIndegree(G, u)**.

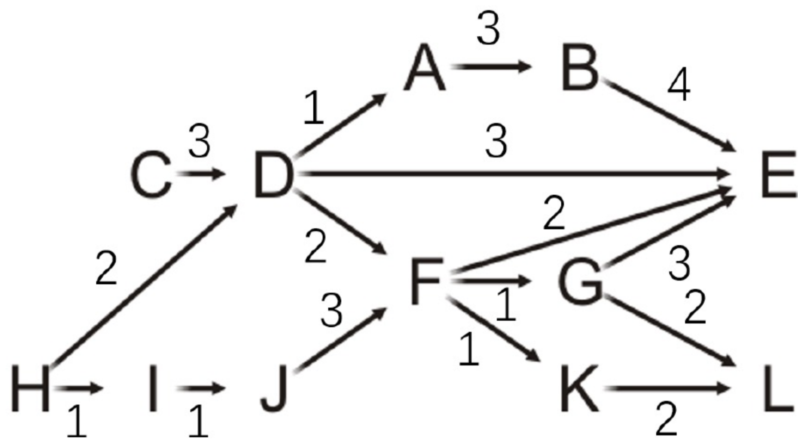


Figure. 3

Input	Output
A	1
C	0
E	12
G	1

Now, you have 5 minutes to implement Q2.

Disjoint Sets Algorithm

- Attempt 1: Quick find
  - SetName[i] = name of the set containing element i
  - Complexity:
    Find: O(1), Union: O(n)

```
Initialize(int N)
  SetName = new int [N+1];
  for (int e=1; e<=N; e++)
    SetName[e] = e;

Union(int i, int j)
  for (int k=1; k<=N; k++)
    if (SetName[k] == j)
        SetName[k] = i;

int Find(int e)
  return SetName[e];
```

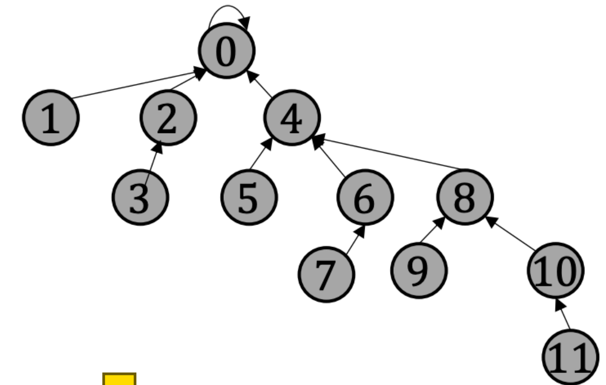
- Attempt2: Smart Union
  - Link smaller tree to the larger one
  - Complexity: Find/Union: O(logn)

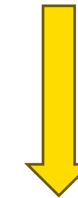
```
Initialize(int N)
  setsize = new int[N+1];
  parent = new int [N+1];
  for (int e=1; e <= N; e++)
    parent[e] = 0;
    setsize[e] = 1;

int Find(int e)
  while (parent[e] != 0)
    e = parent[e];
  return e;

Union(int i, int j)
   i = find(i);
   j = find(j)'
  if setsize[i] < setsize[j]
  then
    setsize[j] += setsize[i];
    parent[i] = j;
  else
    setsize[i] += setsize[j];
    parent[j] = i ;
```

Disjoint Sets Algorithm

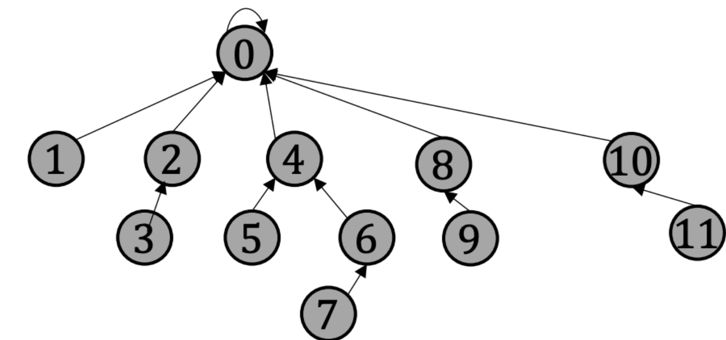
- Ultimate Union-Find: Path compression

```
int Find(int e)
  if (parent[e] == 0)
    return e
  else
    parent[e] = Find(parent[e])
    return parent[e]
```





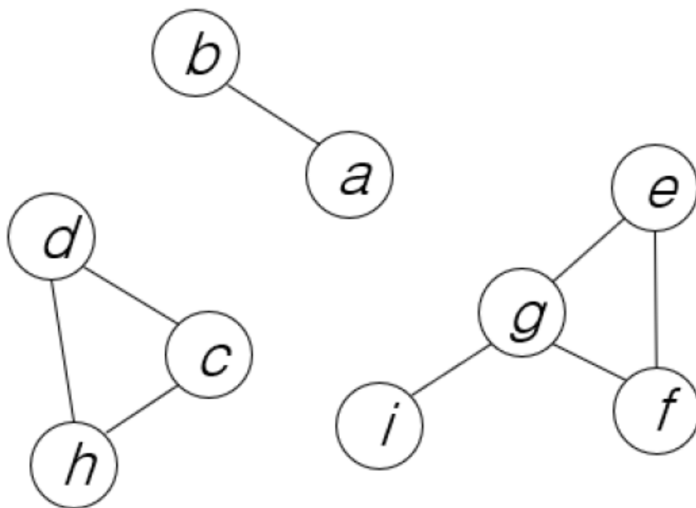
Find(10)



- Any single find can still be O(log(n)), but later finds on the same path are faster.
- Amortized O(1) time for each Union or Find

Exercise 3: Connected Components

- Load the graph in Figure using UndirectedGraph
- The function ConnectedComponents(G, method) will compute all the connected components
- Implement the class QuickFind and UnionFind
- Output the connected component each vertex belongs to and the total number of connected components. (Each connected component can be represented by any unique identifier, e.g., the root vertex in the disjoint structure or the vertex's sequence number in the 'vertex_dict' of 'UndirectedGraph'.)



Now, you have 10 minutes to implement Q3:

Exercise 4: Efficiency on Medium dataset

- Use the dataset in the dataset_30k.txt which contains 30000 nodes and three components.
- Download the dataset from github in colab using the command "!git clone https://github.com/guaiyoui/COMP9312.git". And use "!ls" to check if the download is successful.

```
!git clone https://github.com/guaiyoui/COMP9312.git

Cloning into 'COMP9312'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

```
[22] !ls

COMP9312  drive  sample_data
```

Q & A