IE533 Homework 3

February 10, 2023

NAME: Duo Zhou

NETID: duozhou2

1 1

The bubble sort algorithm is a popular method for sorting n numbers in a non-decreasing order of their magnitudes. The algorithm maintains an ordered set of the numbers $\{a_1, a_2, ..., a_n\}$ that it rearranges through a sequence of several passes over the set. In each pass, the algorithm examines every pair of elements (a_k, a_{k+I}) for each $k = 1, ..., (n-1)$, and if the pair is out of order (i.e., $a_k > a_{k+1}$), it swaps the positions of these elements. The algorithm terminates when it makes no swap during one entire pass.

Show that the algorithm performs at most n passes and runs in $O(n^2)$ time. For every n, construct a sorting problem (i.e., the initial ordered set of numbers $\{a_1, a_2 ..., a_n\}$ so that the algorithm performs $\Omega(n^2)$ operations. Conclude that the bubble sort is an $O(n^2)$ algorithm.

1.1 Solution

Illustrate by python code as below:

[1]:
```
def bubbleSort(_list):
    n = len(_list)
    # Go through all elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n-i-1):
            if _list[j] > _list[j+1] :
                _list[j], _list[j+1] = _list[j+1], _list[j]
    return _list

test = [11, 27, 76, 24, 46, 72, 78, 25, 36]

print(bubbleSort(test))
```

```
[11, 24, 25, 27, 36, 46, 72, 76, 78]
```

Optimal time complexity: $O(n)$. Tt means that the sorting is completed after traversing once and finding no elements that can be swapped, we only need to implement the loop of i.

Worst time complexity: $O(n^2)$. It means in the worst-case scenario, the algorithm has to perform n-1 passes, since each pass requires $n-1$ comparisons and swaps, the total number of comparisons in the worst-case scenario is $(n-1)+(n-2)+...+2+1 = n(n-1)/2$, which is roughly proportional to n^2.

2 2

Suggest an $O(m+n)$ algorithm for identifying all components of a (possibly) disconnected graph. Design the algorithm so that it will assign a label 1 to all nodes in the first component, a label 2 to all nodes in the second component, and so on.

2.1 Solution

One algorithm to identify all components of a graph and label each component is a Depth-First Search (DFS) or Breadth-First Search (BFS) based approach, which would result in the same time and space complexity.

The time complexity of this algorithm is $O(m+n)$, where m is the number of edges in the graph and n is the number of nodes. This is because in the worst case, the algorithm will visit each node and each edge once. The DFS stack requires $O(n)$ space, so the overall space complexity is also $O(n)$.

Illustrate by python code as below:

[2]:
```
def DFS(adj_matrix, start):
    n = len(adj_matrix)
    # start a vertex on the stack first, mark it as visited
    stack = []
    result = []
    visited = [False] * n

    stack.append(start)
    visited[start] = True

    while stack:
        # pop a vertex at the top of the stack, add the neighboring unvisited␣
↪vertices of that vertex to the stack and mark them all as visited
        node = stack.pop()
        result.append(node)
        # Repeat the previous step until the stack is empty
        for i in range(n):
            if adj_matrix[node][i] and not visited[i]:
                stack.append(i)
                visited[i] = True

    return result

def BFS(adj_matrix, start):
```

```
    # the starting vertex start goes into the queue first and is marked as␣
↪visited
    n = len(adj_matrix)
    queue = []
    result = []
    visited = [False] * n

    queue.append(start)
    visited[start] = True

    while queue:
        # the first vertex out of the queue, and the adjacent unvisited vertices␣
↪neighbors of that vertex into the queue, marking them all as visited
        node = queue.pop(0)
        result.append(node)
        # repeat the previous step until the queue is empty
        for i in range(n):
            if adj_matrix[node][i] and not visited[i]:
                queue.append(i)
                visited[i] = True

    return result
```

[3]:
```
adj_matrix = [[0, 1, 0, 0, 0, 0],
              [0, 0, 1, 1, 0, 0],
              [0, 0, 0, 0, 1, 0],
              [0, 0, 0, 0, 0, 1],
              [0, 0, 0, 0, 0, 1],
              [0, 0, 0, 0, 0, 0]]
s = 0

print(DFS(adj_matrix,s))
print(BFS(adj_matrix,s))
```

```
[0, 1, 3, 5, 2, 4]
[0, 1, 2, 3, 4, 5]
```

3 3

In an acyclic network $G = (N, A)$ with a specified source node s, let $\alpha(i)$ denote the number of distinct paths from node s to node i. Give an $O(m)$ algorithm that determines $\alpha(i)$ for all $i \in N$. (Hint: Examine nodes in a topological order).

3.1 Solution

In an acyclic network, nodes can be ordered such that for every directed edge (u, v), node u appears before node v in the order. This order is called a topological order, and it can be found using a topological sorting algorithm. Once the topological order is found, we can use it to determine the number of distinct paths from the source node s to each node i in the network.

The time complexity of this algorithm is $O(m)$, where m is the number of edges in the network. This is because each edge is processed exactly once and takes constant time to update the array α. The space complexity is $O(n)$, where n is the number of nodes in the network, to store the array α and the topological order of the nodes.

Illustrate by python code as below:

[4]:
```
def sort_dag(adj_matrix):
    n = len(adj_matrix)
    sorted_vertices = []
    visited = [False] * n

    def DFS(node):
        nonlocal sorted_vertices
        visited[node] = True
        for i in range(n):
            if adj_matrix[node][i] and not visited[i]:
                DFS(i)
        sorted_vertices.append(node)

    for i in range(n):
        if not visited[i]:
            DFS(i)

    sorted_vertices.reverse()
    new_matrix = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if adj_matrix[j][i]:
                new_matrix[i][j] = 1

    return new_matrix, sorted_vertices
```

[5]:
```
def count_paths(adj_matrix, s):
    n = len(adj_matrix)
    # get the topological order of the vertices using sort_dag
    _, sorted_vertices = sort_dag(adj_matrix)
    # initialize an array to store the count of different paths from s to each␣
 ↪vertex
    count = [0] * n
    count[s] = 1

    for i in sorted_vertices:
        for j in range(n):
            if adj_matrix[i][j]:
```

```
                count[j] += count[i]

    return count
```

[6]:
```
alpha = count_paths(adj_matrix, s)
print(alpha)
```

```
[1, 1, 1, 1, 1, 2]
```

4 4

This part of the homework will also implement some of the algorithms learnt in class. You may use Python, Java, or C++ to execute this. Custom packages cannot be used to replace the algorithms tested.

4.1 a.

Write a function is_dag(G) which returns whether the input directed graph G is a DAG. You should implement a search algorithm to answer this. G should be in the form of an adjacency matrix.

4.1.1 Solution

Based on the DFS code above, the basic idea is to keep track of visited nodes and nodes in the recursion stack during the search. If there is a back edge (an edge connecting a node to one of its ancestors in the search tree), then the graph is not a DAG.

[7]:
```
def is_dag(adj_matrix):
    n = len(adj_matrix)
    visited = [False] * n
    rec_stack = [False] * n

    def is_dag_util(v, visited, rec_stack):
        visited[v] = True
        rec_stack[v] = True
        for i in range(n):
            if adj_matrix[v][i] == 1:
                if not visited[i]:
                    if is_dag_util(i, visited, rec_stack):
                        return True
                elif rec_stack[i]:
                    return True
        rec_stack[v] = False
        return False

    for i in range(n):
        if not visited[i]:
            if is_dag_util(i, visited, rec_stack):
                return False
```

```
    return True
```

[8]:
```
is_dag(adj_matrix)
```

[8]: True

4.2 b.

Write a function make_dag(G) which takes as input a directed graph G and removes the least number of edges to return a new graph G' which is a DAG. Hint: use the function from the previous question to determine whether changes are necessary.

[9]:
```
def make_dag(adj_matrix):
    n = len(adj_matrix)
    new_matrix = [[adj_matrix[i][j] for j in range(n)] for i in range(n)]

    for i in range(n):
        for j in range(n):
            if adj_matrix[i][j]:
                new_matrix[i][j] = 0
                if not is_dag(new_matrix):
                    new_matrix[i][j] = 1

    return new_matrix
```

[10]:
```
adj_matrix_1 = [[0, 1, 0, 0],
                [0, 0, 1, 0],
                [1, 0, 0, 1],
                [0, 1, 0, 0]]

is_dag(adj_matrix_1)
```

[10]: False

[11]:
```
make_dag(adj_matrix_1)
```

[11]: [[0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

4.3 c.

Write a function sort_dag(G) which takes as input a DAG G and returns an equivalent topologically sorted graph $G^{'}$.

4.3.1 Solution

Use the function constructed in Question3:

[12]:
```
sort_dag(adj_matrix_1)
```

[12]: ([[0, 0, 1, 0], [1, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0]], [0, 1, 2, 3])

[13]:
```
sort_dag(adj_matrix)
```

[13]:
```
([[0, 0, 0, 0, 0, 0],
  [1, 0, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 0],
  [0, 0, 1, 0, 0, 0],
  [0, 0, 0, 1, 1, 0]],
 [0, 1, 3, 2, 4, 5])
```

4.4 d.

Given G is the graph produced in 5c), without actually running the code (honor system) if we let H = sort_dag(make_dag(G)), is it true that H=G? Why or why not)

4.4.1 Solution

No, $H \neq G$ after running H = sort_dag(make_dag(G)).

The function make_dag() removes edges from the input graph G to obtain a DAG G'. Therefore, G' is a subgraph of G, which has a different structure than G.

The function sort_dag() takes as input a DAG G' and returns an equivalent topologically sorted graph G''.

Therefore, G'' has the same vertices as G', but with the edges rearranged to be in topological order.

Verify with code:

[14]:
```
G = adj_matrix_1

H, vertex_sort = sort_dag(make_dag(G))

if G == H:
    print(True)
else:
    print(False)
```

```
False
```