Advanced Algorithm HW7

Seungho Jang

October 2020

1 Question 1

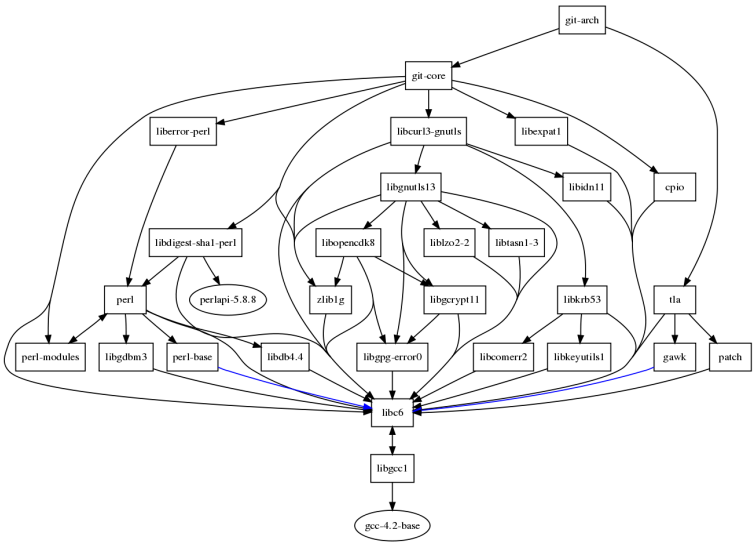

Figure 1: GCC Library

I believe that topological sort algorithm will solve for the package installation because topological sorting algorithm is applicable for DAG(Directed Acyclic Graph), which means that there is no cycle in directed graph. If there is cycle, topological sorting algorithm cannot be applicable for this problem. Also, the topological sorting algorithm is to see which work should be done first to do next work. For exmaple, if libdopencdk8 is inside of libgnut ls 13, and there is zlib1 g. If we want to excute zib1 g, that means that we have to install libgnutls 13 so that we can excute zib1 g, and so on.

2 Question 2

Topological sorting B is correct. When we look at the the figure below. This is done by looking at 0 incoming edges, get rid of its vertex, and put that vertex into queue.

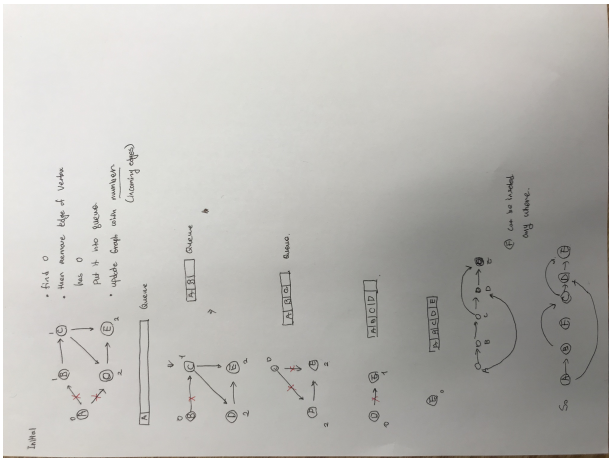

Figure 2: Topological Sorting Algorithm

Note that F can be inserted anywhere, but the reason why A cannot be there because edge going from D to E backward at the same time edge is not going back to E.

3 Question 3

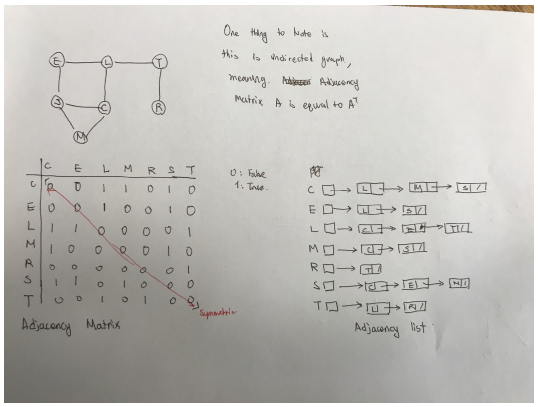

Figure 3: Adjancecy list and matrix

4 Question 4

Advantage of adjacency list representation over adjacency matrix is that it can save memory more. For example, adjacency matrix for directed graph will take a lot of space(memory), requiring $\theta(V_2)$ because it has to save N by N binary element to describe the graph. That means adjacency list only have to store the linked list for node, and result will be less than n_2. One of the disadvantage of adjacency list is that it's hard to depict the sparse graph. Also, it would be hard to search whether an vertex is present in each linked list. That means it will requires n(number of vertex) steps to search at max. It is not completely sure why NetworkX uses, but we can make an assumption that it would be heavier computation and too much memory is taken if they were to use adjacency matrix for their library.

5 Question 5

The code for part a) can be found in this link : Incidence Matrix

b) As shown in the code, diagonal value represents the number of vertex, and each diagonal element in degree matrix represent the number of edges that is connected(incoming and outgoing) to each vertex. For example, 3 represents Node A and has three connected edges. Non-diagonal value represent the number edges that is connected to v[i] to v[j] if v is denoted as vertex.

6 Question 6

Main difference between bfs to dfs is how they store the data into queue or stack. The modified bfs could act like dfs as shown below:

```
Modified_BFS(G, s, dest)
 for each vertex u ∈ G.V - {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
 s.color = GRAY
 s.d = 0
 s.π = NIL
 Q = ∅
 push (Q, s)
 while (Q ≠ ∅)
    u = POP(Q, s)
    if u == dest
       break;
    for each v ∈ G.Adj[u]
       if v.color == WHITE
          v.color = GRAY
          v.d = u.d + 1
          v.π = u
          push (Q, s)
    u.color = BLACK
```

Figure 4: Modified BFS

7 Question 7

The running time of DFS and BFS is O(V+E). Discuss how aggregate analysis (ammortized analysis) applies to derive this running time.

Reference used : CLR(Introduction to Algorithm)

1. Aggregate Analysis on BFS

   - There are two operations in BFS: dequeue and enqueue. Each reachable vertex can only be discovered and then enqueued once, A vertex's outgoing edges only get explored when it is dequeued, which happens only once. Which means both would take O(1), so then the total time in the queue operation is O(V) where V represents number of vertex.
   - because how the BFS operates, expanding search through adjacent vertices. the sum of the edges is $\theta(E)$, then the time that takes to scans through all adjacency vertex is O(E).
   - Then, the total time will be O(V + E) considering O(V) for initialization.

DFS is also the same except the operations that involved with DFS are push and pop. Both operations takes O(1), one at a time. Then the total time will be O(V). The DFS operates adjacent adjacency vertex but searching the deepest node first.

8 Question 8

The code can be found in here : Incidence Matrix

The dfs trace/tree, discovered/finished is done for A and B in figure below,

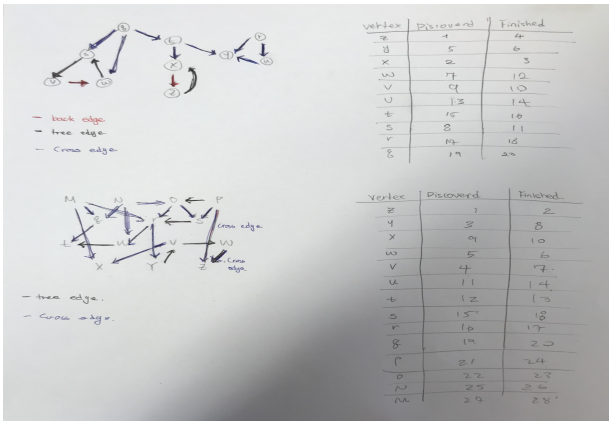

Figure 5: DFS Tree and Discovered-finished-Time

The topological order for A and B is shown below

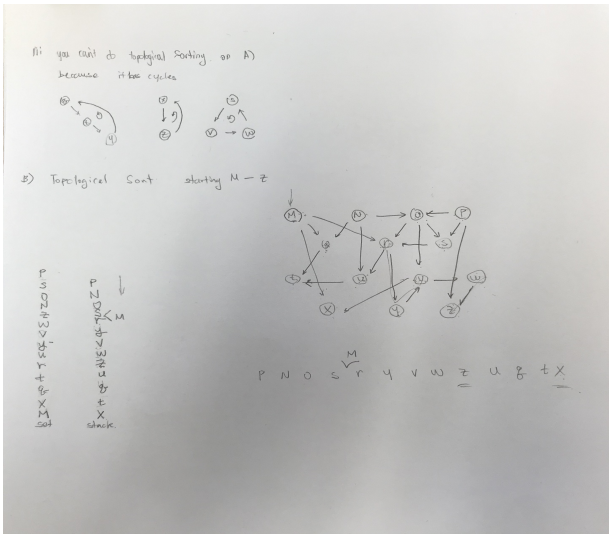

Figure 6: DFS Tree and Discovered-finished-Time

The topological order from **Networkx** is done below, to note position of z and x can be changed depending the order of searching.

```
import networkx as nx
DG = nx.DiGraph()
DG.add_edge('m','x')
DG.add_edge('m','q')
DG.add_edge('m','r')
DG.add_edge('n','q')
DG.add_edge('n','u')
DG.add_edge('n','o')
DG.add_edge('o','r')
DG.add_edge('o','s')
DG.add_edge('p','o')
DG.add_edge('p','s')
DG.add_edge('p','z')
DG.add_edge('q','t')
DG.add_edge('r','u')
DG.add_edge('r','y')
DG.add_edge('s','r')
DG.add_edge('u','t')
DG.add_edge('v','x')
DG.add_edge('v','w')
DG.add_edge('w', 'z')
DG.add_edge('y','v')

# Visualize the graph
nx.draw(DG, alpha = 0.5, with_labels = True)
# Run topological sorting, and print the order
print(list(nx.topological_sort(DG)))
```

```
['p', 'n', 'o', 's', 'm', 'r', 'y', 'v', 'w', 'z', 'x', 'u', 'q', 't']
```

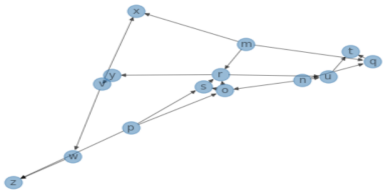

Figure 7: DFS Tree and Discovered-finished-Time