COMP20007 Desigs of Algorithms, Semester 1 2016
Adrian Hendrawan Putra
ID: 743206
Assignment 1 Report

Preface
In this assignment, we were to implement two sorting algorithms for directed graphs from the Wikipedia. The algorithms are the "*Depth-First Search*" (**DFS**) and the "*Kahn's Algorithm*" (**Kahn**).

Data Collection
To collect data sample, the implemented algorithms are tested against random graphs of varying sizes generated by the provided *daggen.c* program with following parameter set:

Parameter Set 1					
Graph Order: |V|					
100	200	500	800		
Edge Chance (%)					
15	30	45	60	75	90

Parameter Set 2							
Graph Order: |V|							
1000	1100	1200	1300	1400			
Edge Chance (%)							
10	20	30	40	50	60	70	80

For each pair of parameter (Graph Order, Edge Chance), 20 graphs were generated and tested against the program. Each implementation (**DFS** and **Kahn** respectively) has been tested, and verified against a total of 1280 randomly generated graphs.

It is important to note that due to the percentage-based edge generation, the vertex to edge ratio does not remain constant as the number of vertex in a graph increases. With that in mind, a smoother edge chance transition is used (on 'Parameter Set 2') when generating graphs with higher number of vertices.

The collection of data set (generated by the provided *main.c*), consisting of: *edge count*, *vertex count,* and time ('*clock ticks*'), is then plotted in a 'time vs |E|+|V|' graph (where |E| is the number of edges, and |V| is the number of vertices).

Pathological Parameter								
Graph Order: |V|								
100	200	500	800	1000	1100	1200	1300	1400
Edge Chance (%)								
0								

For the pathological graph, the following parameter has been used.

The generated graphs are a list of vertices with no edges.

Analysis: DFS
The DFS topological sorting algorithm starts by 'visiting' a vertex *v* and marking it temporarily. It will then 'traverse' through all of the reachable vertex *u* from *v* (corresponding to the edges of *v*), and vertex *v* will be marked permanently and be added to the sorted list *L* after the 'traversal'.

It is important to note that the algorithm will not 're-traverse' permanently marked vertices, which means that a vertex will only be 'traversed' during the first 'visitation' of that vertex.

Since marking and unmarking a vertex is a constant-time operation, and adding an element to the head of a linked list is also a constant-time operation; the expected average running time for **DFS** is linear in respect to the number of edges and the number of vertices in a graph. In other words, the running time of **DFS** is asymptotic to **O(|V|+|E|)**.

DFS:
L ← Empty list containing sorted vertices
for each unmarked vertex *v* in graph **do:**
    visit(*v*)
return *L*.

function visit(vertex *v*)**:**
if *v* is temporarily marked **then:**
    stop. graph is cyclical
if *v* is not marked **then:**
    mark *v* temporarily
    for each vertex *u* reachable from *v* **do:**
        visit(*u*)
    mark *v* permanently
    unmark *v* temporarily
    add *v* to head of *L*

As observed on *fig.1a*, the plotted graph shows a very convincing linear growth in time with respect to |V| and |E|. A sample on a pathological input can be seen on *fig.1b*. The graph also shows a linear growth.

Kahn:

L ← Empty list containing sorted vertices
S ← List of vertices with no incoming edges
while *S* is non-empty **do:**
    remove a vertex *v* from *S*
    add *v* to the tail of *L*
    for each vertex *u* with an edge *e* from *v* to *u* **do:**
        remove edge *e* from the graph
        if *u* has no other incoming edge **then:**
            insert *u* into *S*
if graph still has edges **then:**
    stop. graph is cyclical
else:
    return *L*.

Analysis: Kahn

The Kahn's topological sorting algorithm (which will be referred to as "**Kahn**") starts by having a list *S*, containing 'source vertices' (no incoming edges).

The algorithm will then iterate through the vertices in S, by adding the vertex *v* to the tail of the sorted list *L*, and removing all of the edges connected from *v.* If the other vertex *u* has no other incoming edges, *u* is then inserted into *S*.

It is important to note that if the graph is acyclic, the algorithm will eventually remove all of the edges in the graph by iterating through all of its vertices. Therefore if the cost of all other operation is constant, Kahn's algorithm will also have an expected lower bound of **O(|V|+|E|)**.

However, due to the provided singly linked list implementation from *list.c*, it is important to realize that the cost of operation of adding vertex *v* to the tail of list *L* is bounded by **Θ(n)**, where **n** is the length of the linked list L. Furthermore, the operation of removing an edge *e* by using the provided '*del()'* function form *list.c* is equivalent to searching for a specific element in a singly linked list, which requires the traversing through the linked list. In other words, it is bounded by **Ω(n)**, where **n** is the length of the linked list.

Since the list *L* will eventually grow to the size of |V|, and that every vertex will have to have its edge removed, a naïve implementation of the **Kahn's Algorithm** will result in a non-optimal average operational cost asymptotic to **O($|V|^2 + |E|^2$)**. This can be observed on *fig.3*, where the curve is very reflective of a polynomial graph of degree 2. A minor adjustment in the implementation can improve the average running time up to the expected linear lower bound of **O(|V|+|E|)**.

The adjustments are as follows:

- The list *L* is constructed in reverse, allowing the constant operational cost of adding vertex *v* to the head of *L*.
- The list *L* is then reversed before returning. Reversing a list is bounded by **Θ(n)**, where **n** is the number of vertices |V| on the graph. The list has only have to be reversed once and thus only contributes an operational cost that grows linearly to the number of vertices |V|.
- An auxiliary array to keep track incoming-edges of each vertex can be constructed by traversing each vertex, and iterating through all of its edges. This operation is also bounded by **Θ(n+m)**, where **n** is the number of vertices |V| and **m** is the number of edges |E| in our graph. This also only contributes an operational cost that grows linearly to the number of vertices |V|, and the number of edges |E|.
- The auxiliary array allows a constant time operation. Instead of removing an edge *e*, the number of incoming edges of the vertex *u* gets decremented by one.
- When checking for a cyclical graph, the auxiliary array can be traversed to check whether any vertex still has any edges. The elements of the array is exactly the same as the number of vertices |V|.

The result of these adjustments can be observed on *fig.2a*, where the plotted graph shows a convincing linear growth in time with respect to |V| and |E|. A sample on a pathological input can be seen on *fig.2b*. The graph also shows a linear growth.

Appendix

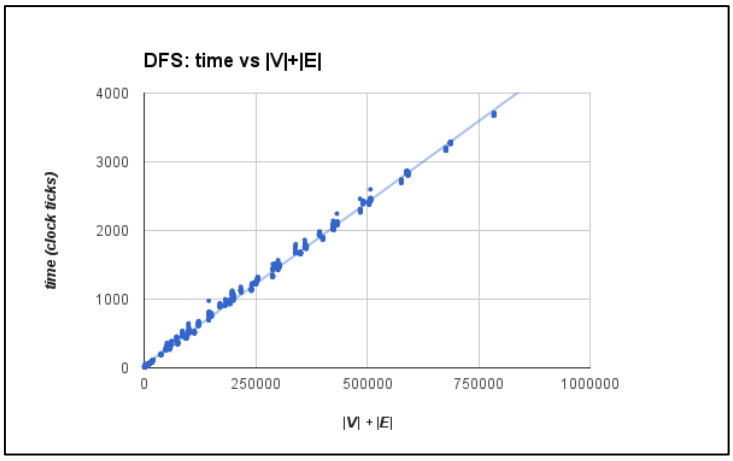

Figure 2a

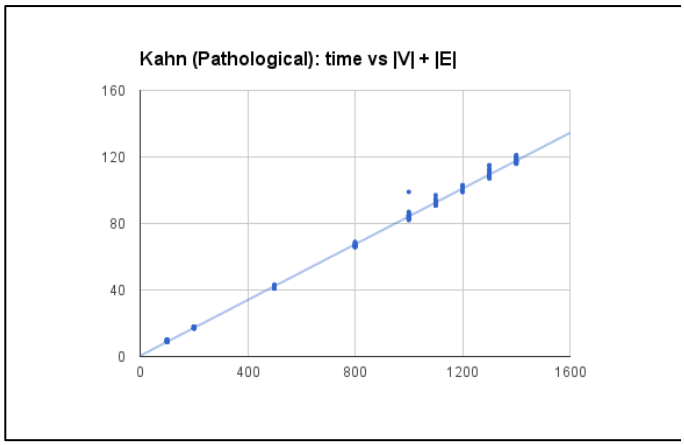

Figure 1b

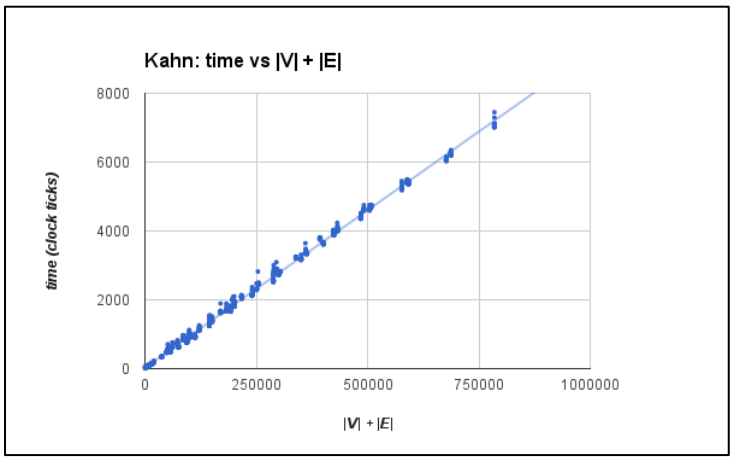

Figure 2a

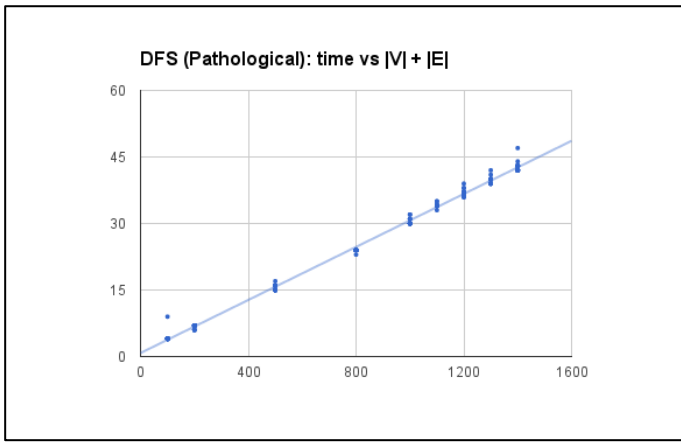

Figure 2b

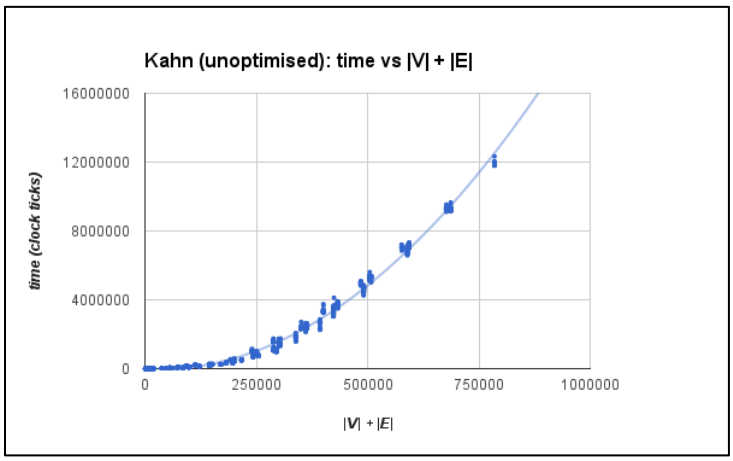

Figure 3