ECE 592 Homework 3

Kudiyar Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)

October 16, 2017

Note that the entries are links.

Contents

List of Figures

Listings

1 Computational Complexity

1.1 (a) $n^2 + 100n = \theta(n^2)$

For all $n > 0$, $n^2 + 100n > 1 * n^2$. Thus, $c_1 = 1$
Let's say $c_2 = 2$. Then, $n^2 + 100n \leq 2n^2$
$100n \leq n^2$
$n \geq 100$
$1n^2 \leq n^2 + 100n \leq 2n^2$ for all $N_0 \geq 100$

1.2 (b) $(log(n))^3 = O(n)$

Let's say $N_0 \geq 10$. Then,
$(log(10))^3 = c10$
$c = \frac{1}{10}$, thus for all $N_0 \geq 10$
$0 \leq log(n))^3 \leq \frac{1}{10}n$
So $log(n)^3$ is on the order of $O(n)$.

1.3 (c) $n^{0.5} = \Omega((log(n))^3)$

For $N_0 \geq 10000$, $c * (log(10000))^3 = \sqrt{(10000)}$
$64c = 100$, then $c = \frac{100}{64}$.
$0 \leq \frac{100}{64}(log(n))^3 \leq n^{0.5}$ for all $N_0 \geq 10000$

2 Searching Problem

2.1 (a) Linear Time Search

Listing 1: Pseudocode to search through a list of values.

```
clear; clc; close all;

prompt = {'Input sequence of numbers:', 'Value:'};
% Store input
array = inputdlg(prompt);
x = str2num(array{1,1});
v = str2num(array{2,1});

% step through array in linear time
found = 0;
for i=1:length(x)
   if(v == x(i));
       fprintf('%d\n', i);
           found = 1;
       break;
   end
end

% if not found
if (~found)
        fprintf('Value not found in sequence.');
end
```

2.2 (b) Elements Accessed

On average we will have $\frac{N+1}{2}$ elements to be searched because $1 \leq index \leq N$ since they are equally likely to appear the mean will be $\frac{1+N}{2}$. The worst case will be $\mathcal{O}(N)$, or N, in which case the algorithm has to iterate through all elements in order to find the value. Both the average and worst cases will have $\Theta(N)$ as both depend on N.

2.3 (c) Alternate Data Structure

An alternate data structure that would provide fast searches would be a sorted array that's pre-allocated in memory. The complexity of an individual search through a sorted list is$O(log(N))$. The search would be completed by recursively choosing the middle element of sub-vectors of the array. When the value is larger than the middle element, the search algorithm would choose the center of the elements to the right of the original middle element. If the value being sought was smaller, the algorithm would search through the left sub-array. It would proceed recursively until the element was found.

Note that there is a set-up cost to sorting the list. The complexity of sorting is $O(Nlog(N))$. But this is an up-front cost that wouldn't be nearly as expensive as using linear time searches from the beginning. In addition, insert sort could be used to add or remove elements from the already sorted list quickly.

3 Paths and Simple Paths

A simple path is defined as **a path which contains no repeated vertices.** Both directed and undirected graphs can exist as fully connected graphs, where each vertex is connected to each other. i.e. there is a path from any vertex to another. Likewise, both directed and undirected graphs can be defined as multi-graphs, where multiple (redundant) edges can connect vertices.

If we have a graph $G(V, E)$ consisting of vertices V and edges E, where a path exists between vertices v_j and v_k, we can show that there exists a simple path that also connects the two vertices.

Assume that there only exists a path that contains repeated vertices between v_j and v_k. The simple path is found as the first subset of the path that doesn't have repeated vertices. Take the following example:

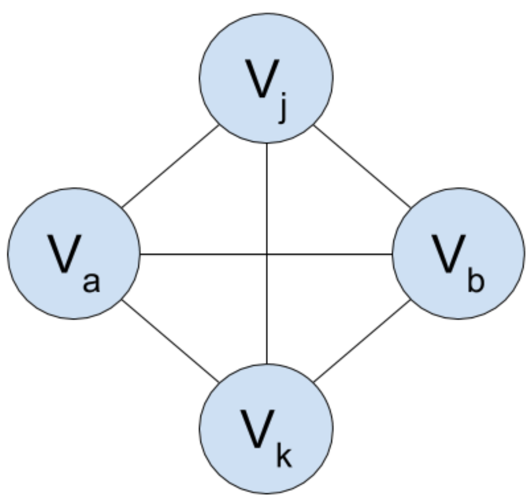

Figure 3.1: A graph with cycles has at least one simple path between vertices v_j and v_k.

The path between v_j and v_k can be represented in an infinite combination of ways, assuming vertices can be duplicated in a path. For instance, the path

$$p_1 = (V_j, V_a, V_b, V_j, V_k) \tag{3.1}$$

Reveals a simple path between V_j and V_k as simply the two nodes: $p_{simple} = (V_j, V_k)$. If there needed to be another connection through V_b to reach V_k, the simple path would just have V_b between the two nodes. As long as a path exists between V_j and V_k, even one with duplicate nodes, a simple path can be extracted from it by virtue of its ultimate connection.

4 Mathematical Induction

The following function represents the runtime of a sub-problem:

$$T(n) = \begin{cases} 2 & n = 2 \\ 2T(\frac{n}{2}) + n & n = 2^k,\ k > 1 \end{cases} \tag{4.1}$$

To show that $T(n) = n \times log_2(n)$, we use mathematical induction.

4.1 Basis Case

We start with two basis cases, for $k = 1, 2$.

$\underline{k = 1}$: $T(2) = 2 = 2log_2(2) = 2$. Thus the relation holds for $k = 1$.

$\underline{k = 2}$: $T(4) = 2T(2) + 4 = 4 + 4 = 8 = 4log_2(4) = 8$. Thus the relation holds for $k = 2$.

4.2 Inductive Hypothesis

Given that the basis case holds for small values of n, we hypothesize that the relationship holds for $T(2^k)$ where $k = K$.

4.3 Inductive Step

Assuming that the hypothesis holds, then it should also hold for $k = K + 1$.

$\underline{k = K + 1}$:

$$\begin{aligned} T(2^{K+1}) &= 2T(\frac{2^{K+1}}{2}) + 2^{K+1} \\ &= 2T(2^K) + 2^{K+1} \\ &= 2(2T(2^{K-1}) + 2^K) + 2^{K+1} \\ &= 2^2T(2^{K-1}) + 2 * 2^{K+1} \\ &\text{Note that this proceeds recursively until } T(2) \text{ is reached} \\ &= 2^K T(2) + (K)2^{K+1} \\ &= (K+1)2^{K+1} = (2^{K+1})log_2(2^{K+1}) = (2^{K+1})(K+1) \end{aligned} \tag{4.2}$$

We've shown that the relationship holds for $k = K + 1$, so it must hold for any positive integer k that $T(n = 2^k) = n \times log_2(n)$.

5 Dijkstra's Algorithm

5.1 Dijkstra's Algorithm

v	A	B	C	D	E	F
A	0_a	3_a	5_a	9_a	inf_a	inf_a
B	0_a	3_a	5_a	7_b	10_b	inf_a
C	0_a	3_a	5_a	7_c	10_b	inf_c
B	0_a	3_a	5_a	7_b	9_d	9_d
C	0_a	3_a	5_a	7_c	9_d	9_d

The bottom two rows indicate that the shortest paths are $p_1 = A, B, D, F$ and $p_2 = A, C, D, F$.

5.2 Bellman-Ford Algorithm

B	A	C	D	E	F		
0	3	3	4	7	6	$\rightarrow$	iteration 1
0	3	3	4	6	6	$\rightarrow$	iteration 2

Note that this algorithm 'runs' faster than Dijkstra's Algorithm.