cse210 ass02

Palash A. (*e17cse069*)

16 April, 2019

1 Knuth Morris Pratt algorithm

Watch the video on YouTube or scan the QR code below:

2 Greedy vs Dynamic

2.1 Greedy Algorithm

Greedy programming is used in optimization problems following a simple, intuitive strategy. The problem is broken down into sub problems and the best solution at that step is chosen. Greedy algorithm may not give the optimal solution to every problem and in general is used if the following two conditions are met:
1. **Greedy Choice**: A global optimal solution can be obtained by choosing the optimal solution at each step.
2. **Optimal substructure**: The global optimal solution solution to the problem contains the optimal solutions to every sub problem.

2.1.1 Example

Huffman Encoding [4] is where the greedy approach is useful. Another algorithm that uses the greedy approach is Djikstra's algorithm to find the shortest path between two vertices.

2.2 Dynamic Programming

Dynamic programming is a problem solving approach where solutions to simpler, similar sub problems are stored to build the solution to a more complex problem. This bottom up approach is similar to recursion (where base cases allow us to determine final value) and works best when the new values are dependent on the previously calculated values. Dynamic programming has one condition that the problem must meet to be used that it should have overlapping sub problems. If not, divide and conquer may be used to solve the problem where memoization is not required increasing space efficiency of the solution.

2.2.1 Example

Dynamic Programming is exceptionally useful in solving problems such as BACKPACK PROBLEM (*Given a set of treasures with known values and weights, which ones should be picked to maximize profit within the capacity of the backpack*), LONGEST COMMON SUBSEQUENCE (*Given two sequences, which is the longest subsequence common to both*), and a better way to compute FIBONACCI NUMBERS than recursion.

2.3 Comparison

The main difference between the two is the **greedy choice property**. Greedy programming chooses the most optimal solution based on the choices so far and not on future or all choices. Whereas, dynamic programming is exhaustive and is guaranteed to find the optimal solution. In other words, greedy algorithms do not reconsider choices.

For example, consider the following tree where the problem is to find the path with the max sum of node values.

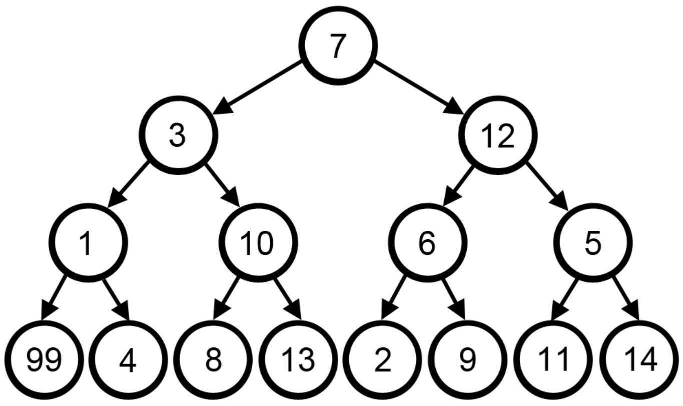

Figure 1: Find the path with max sum

The path followed by the greedy approach is $7 - 12 - 6 - 9$ (by choosing the maximum value at each step). However, using the DP approach, the tree problem can be broken down as such:
$solution = 7 + max(left\_subtree, right\_subtree)$, which can be further broken down for the respective subtrees until the leaf nodes are reached. On woking its way back up, this approach will follow the path $99 - 1 - 3 - 7$ yielding 110 as the solution.

Hence, the path with the highest value is clearly $7 - 3 - 1 - 99$. The greedy approach fails in this problem to find the global optimal solution as it does not reconsider the choices and works from top down as compared to DP which solves the problem exhaustively and finds the optimal solution.

3 Floyd Warshall Algorithm

The Floyd Warshall algorithm is a shortest path algorithm for a graph that finds the shortest path (based on weights) between all pairs of vertices.

3.1 Algorithm

Algorithm 1 Floyd Warshall Algorithm

```
$V \leftarrow$ (number of vertices)
$dist \leftarrow |V| * |V|$
for v in V do
    $dist[v][v] \leftarrow 0$
end for
for edge (u,v) do
    $dist[u][v] = weight(u, v)$
end for
for k from 1 to V do
    for i from 1 to V do
        for j from 1 to V do
            if $dist[i][j] > dist[i][k] + dist[k][j]$ then
                $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$
            end if
        end for
    end for
end for
```

3.2 Example

Consider the graph on the side

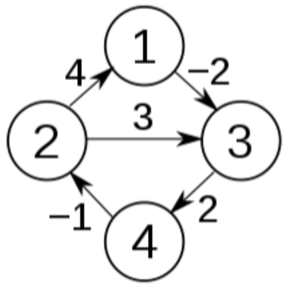

Figure 2: Floyd Warshall Example Graph

The following matrix is created after the first two for loops (∞ is inserted where no path exists),

	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0

When $k = 1$, the if condition fails for all values of i and j and hence, the table is unchanged.

When $k = 2$, the if condition passes when $(i = 2, j = 3)$ as $3 > 4 + (-2)$ and the matrix is updated as such,

	1	2	3	4
1	0	∞	-2	∞
2	4	0	*2*	∞
3	∞	∞	0	2
4	∞	-1	∞	0

The if condition also passes when $(i = 4, j = 1)$ and $(i = 4, j = 3)$ producing the following matrix:

	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	*3*	-1	*1*	0

When $k = 3$, the if condition is passed when $(i = 1, j = 4)$ and when $(i = 2, j = 4)$ producing the following matrix:

	1	2	3	4
1	0	∞	-2	*0*
2	4	0	2	*4*
3	∞	∞	0	2
4	3	-1	1	0

When $k = 4$, the if condition is passed when $(i = 1, j = 2)$, $(i = 3, j = 1)$, and $(i = 3, j = 2)$ producing the matrix:

	1	2	3	4
1	0	*-1*	-2	0
2	4	0	2	4
3	*5*	*1*	0	2
4	3	-1	1	0

Hence, the shortest paths between all vertices in Figure 3.2 is given by the following matrix:

	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

4 Huffman Encoding

Huffman encoding is an algorithm used for data compression and forms the basic idea behind file compression. Characters are stored using 8 bits. Huffman encoding reduces the size by encoding the most frequent characters with lesser bits, thereby reducing size. Huffman coding follows the PREFIX RULE which states that no code is a prefix for another code.

Huffman coding works by building a binary tree of nodes based on character frequencies, where the lowest frequency is given the highest priority. The nodes in a Huffman tree are of two types - LEAF NODES which contain the character and the frequency of that character and INTERNAL NODES which contain the total frequency till that node (counted from the leaf) and the links to the two child nodes. Priority queues are typically used to build the Huffman tree.

0 and 1 is assigned to the left and right branch respectively and the codes are assigned to each character going from the root to the leaf. Hence, the most frequent characters will be encoded with the smallest codes, reducing size. [See example 4.2]

4.1 Algorithm

Algorithm 2 Huffman Tree Pseudocode

```
Require: A list $[C]$ of all characters with their frequencies
  $n \leftarrow C.size$
  $Q \leftarrow priority\_queue()$ {The priority is based on frequency, with lower frequency implying higher priority}
  for all $c$ in $C$ do
    $Q.push(node(c))$
  end for
  while $Q.size() \neq 1$ do
    $i \leftarrow node()$
    $i.left = Q.pop()$
    $i.right = Q.pop()$
    $i.frequency = i.left.frequency + i.right.frequency$
    $Q.push(i)$
  end while
```

4.2 Example

For example, consider a string of 100 characters with the following character frequencies: $\{a : 45,\ b : 13,\ c : 12,\ d : 16,\ e : 9,\ f : 5\}$. Encoding the 100 characters would require $100 * 8 = 800$ bits.

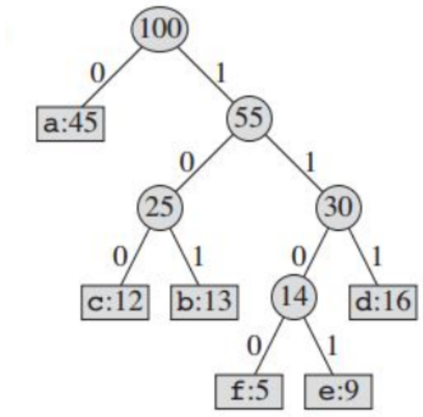

Figure 3: Huffman Tree

The Huffman tree above produces the following codes:

Character	Frequency	Code	Bits needed
a	45	0	$1 * 45 = 45$
b	13	101	$3 * 13 = 39$
c	12	100	$3 * 12 = 36$
d	16	111	$3 * 16 = 48$
e	9	1101	$4 * 9 = 36$
f	5	1100	$4 * 5 = 20$

Total bits needed: $45 + 39 + 36 + 48 + 36 + 20 = 224$

Hence, Huffman encoding reduces the bits needed to represent the string from 800 bits to 224 bits!

5 Topological Sort

Topological sorting for a directed acyclic graph is a linear ordering of vertices such that for every edge $u{-}v$, u comes before v. A certain directed acyclic graph may have multiple linear orderings based on the algorithm used to topologically sort the graph.

5.1 Example

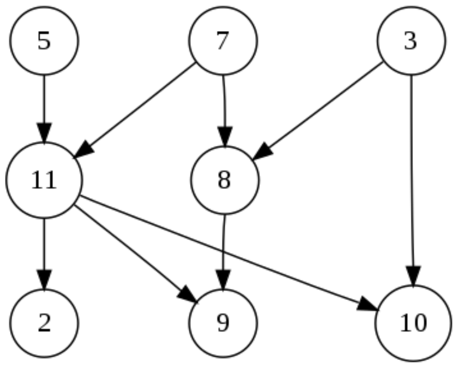

Figure 4: DAG for Topological Sort

The above graph can have multiple linear orderings based on different criteria:

- 5, 7, 3, 11, 8, 2, 9, 10 *(visual: left-to-right, top-to-bottom)*
- 5, 7, 3, 8, 11, 10, 9, 2 *(fewest edges first)*
- 3, 5, 7, 8, 11, 2, 9, 10 *(smallest numbered vertex first)*
- 3, 7, 8, 5, 11, 10, 9, 2 *(random)*

5.2 Applications

The main applications of this sort is to schedule jobs from the given jobs and their dependencies. In computer science, instruction scheduling, data serialization, and determining order of compilation (in case of makefiles) are some places where topological sorting is used.

6 Graph Search

Breadth First Search and Depth First Search traversal order on the graph given below in Figure 5

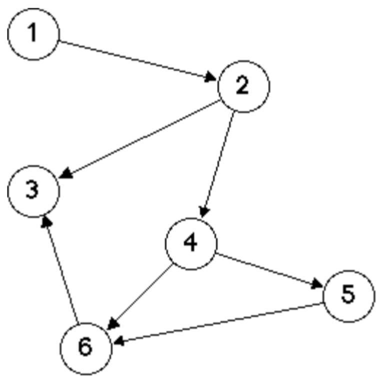

Figure 5: Given graph

6.1 Breadth First Search

Nodes to visit	Nodes Visited
1	
2	1
3,4	1,2
4	1,2,3
6,5	1,2,3,4
5	1,2,3,4,6
	1,2,3,4,6,5

Final order of traversal: [1,2,3,4,6,5]

6.2 Depth First Search

Nodes to visit	Nodes Visited
1	
2	1
3	1,2
4,6	1,2,3
6	1,2,3,4
5	1,2,3,4,6
	1,2,3,4,6,5

Final order of traversal: [1,2,3,4,6,5]