Addis Ababa University

Addis Ababa Institute of Technology

School of Electrical and Computer Engineering

Algorithm Analysis and Design Assignment #2

By: Kelem Negasi

ID: ATR/8467/07

Date of submission: **01/03/2019**

Submitted to Menilik B.

Report

1. Greedy Algorithms

A greedy algorithm is an algorithmic strategy that makes the best optimal choice at each small stage with the goal of this eventually leading to a globally optimum solution. This kind of algorithm works for problems whose local optimal solution leads to global optimal solution. This means taking small optimal solutions leads to general optimal solution of the problem.

1.1. Fractional Knapsack

Problem: Given a set of items and total capacity of a knapsack, find the maximal value of fractions of items that fit into the knapsack.

Solution:

To solve this problem first we have to examine the nature of the problem the kind of algorithm we can use in solving it. As we can see this problem has a property that a local optimal solution can lead to the general optimal solution. Therefore we can use the greedy algorithm to solve it.

Here are the procedures followed in the algorithm:

i. Accept the number of items, the maximum capacity of the knapsack and the list of values and the respective weights of the items.
ii. Select the item with the maximum value to weight ratio from the list of items.
iii. Compare the weight of the item selected in step 1 with the remaining capacity of the knapsack.
iv. If the capacity of the knapsack is greater than the weight of the selected item then take the whole item and update the weight of that item and the value.
v. If the capacity of the knapsack of is smaller than the given weight of the selected item then take a fraction of that item that is as much of the remaining capacity. Then update the weight of the item and the value accordingly
vi. Do the above steps until the knapsack gets full by including as much items as possible.
vii. When the knapsack is full or when all the items are taken iteratively (if possible) then display the final result.

Here is some sample input and output:

```
run:
3 50
60 10
100 20
120 30
the maximum value possible is 240.0000
```

Explanation here we take the whole item 1 and item 2 and two third of item 3

1.2. Minimum Dot product

Problem:

The dot product of two sequences a_1, a_2. . . an and b_1, b_2, . . . , b_n of the same length is equal to.

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n.$$

Given two sequences a_1, a_2..., an and b_1, b_2…b_n, find a permutation π of the second sequence such that the dot product of a_1, a_2... a_n and $b_{\pi 1}$, $b_{\pi 2}$... $b_{\pi n}$ is minimum.

Solution:

This problem can also be solved using greedy algorithm. This mean to get the minimum product we have to first multiply the largest value in the sequences
a_1, a_2... a_n, and multiply it with the minimum value in the b_1, b_2…b_n, sequence to get the smallest result. By following the same rule in the remaining sequences we can finally get the smallest sum of products. This gives us the solution to the minimum dot product problem. Now to implement this solution in the algorithm we follow the following the procedures.

Procedures:

i. Accept the size of the vector and sequences of numbers from the user
ii. Sort the sequence of values in the first vector i.e. $a_1, a_2 \ldots a_n$ in ascending order:
iii. Sort the sequence of values in the second vector i.e. $b_1, b_2 \ldots b_n$ in descending order
iv. Then perform the dot product operation and display the final result.

Sample Input and output:

```
run:
5
3 5 1 -3 2
6 3 1 2 5
4
```

2. Dynamic Programming

2.1. Primitive Calculator

Problem:

You are given a primitive calculator that can perform the following three operations with the current number x: multiply x by 2, multiply x by 3, or add 1 to x. Your goal is given a positive integer n, find the minimum number of operations needed to obtain the number n starting from the number 1. Given an integer n, compute the minimum number of operations needed to obtain the number n starting from the number 1.

Solution:

In solving this problem we have to use dynamic programming with memoization in bottom up fashion because it is an optimization problem where sub problems overlap in their solution.

To write a dynamic programming algorithm first let see the problem in very carefully. Our goal is to find the minimum number of operation to reach a number n using the given basic three operations.

Let's express the number of operations to reach n as cost of n i.e. C(n). The question now is how to express C(n) in terms of C(n/2), C(n/3) and C(n-1). The first thing to do is finding the recurrence.

For n =2, 3 it is simple because its cost is 1 since it can computed using the basic operation from 1. And the cost for n=1 it is 0 because it doesn't need any operation to reach it.

For n=4

4 =2*2= (4/2)*2

Or 4=3+1= (4-1) +1

The cost now becomes:

C(4) = C(4/2)+1 or C(4) = C(4-1)+1 because it needs only one operation after 2 or 3 that is multiply by 2 or add 1 respectively.

For n=5:

5= 4+1 = (5-1)+1

C(5) = C(5-1) +1

For n=6

6 = (6/2)*2

0r (6/3)*3

or C(6-1)+1

C(6) =C(6/2)+1

Or C(6) = C(6/3)+1

Or C(6) =C(5) +1

As we can see from the above expressions the cost of n could be defined as follows.

If n is divisible by both 2 and 3
C(n) =C(n/2)+1 or C(n/3)+1 or C(n-1)+1
If n is only divisible by 2
C(n) =C(n/2)+1 or C(n-1)+1
Other wise
C(n) = C(n-1)+1

The minimum cost of n is therefore the minimum of the cost expressions which it can be expressed by.

Now:
Minimum cost of n which is $C(n)_{min}$ is given by:
$C(n)_{min}$ = min(C(n/2)+1), C(n/3)+1, C(n-1)+1) if n is divisible by 3
$C(n)_{min}$ =min(C(n/2)+1, C(n-1)+1) if n is divisible by 3
$C(n)_{min}$ = C(n-1)+1 if n is odd

Now the procedures of the algorithm are:

i. Starting from n=2 check if each number is divisible by 2 and/or by 3 until we reach the given number n.
ii. Then find the minimum of the costs to find reach n/2, n-1 and/or n/3 at each step
iii. Then add that number in to the predecessor of n for each number until we reach n
iv. Finally display the cost of n and the list of the sequences of numbers to reach

Sample Input and output:

```
run:
55
5
1 3 9 27 54 55 BUILD SUCCESSFUL (to
```

3. Paths In Graphs

3.1. Computing the Minimum Number of Flight Segments

Problem:

You would like to compute the minimum number of flight segments to get from one city to another one. For this,
you construct the following undirected graph: vertices represent cities, there is an edge between two vertices
whenever there is a flight between the corresponding two cities. Then, it suffices to find a shortest path from one of the given cities to the other one.
Generally:
Given an undirected graph with n vertices and m edges and two vertices u and v, compute the length of a shortest path between u and v (that is, the minimum number of edges in a path from u to v).

Solution:

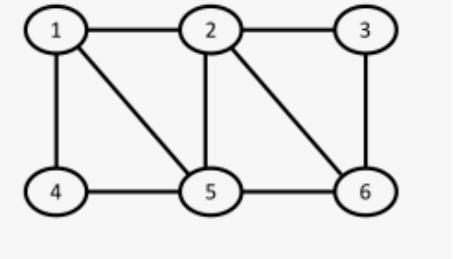
This is a problem of finding the shortest path in a graph. The kind of graph given here is undirected and unweighted graph. Breadth first algorithm is an appropriate way to find the shortest path because in breadth first search the first time a node discovered the distance to that node gives us the shortest path.

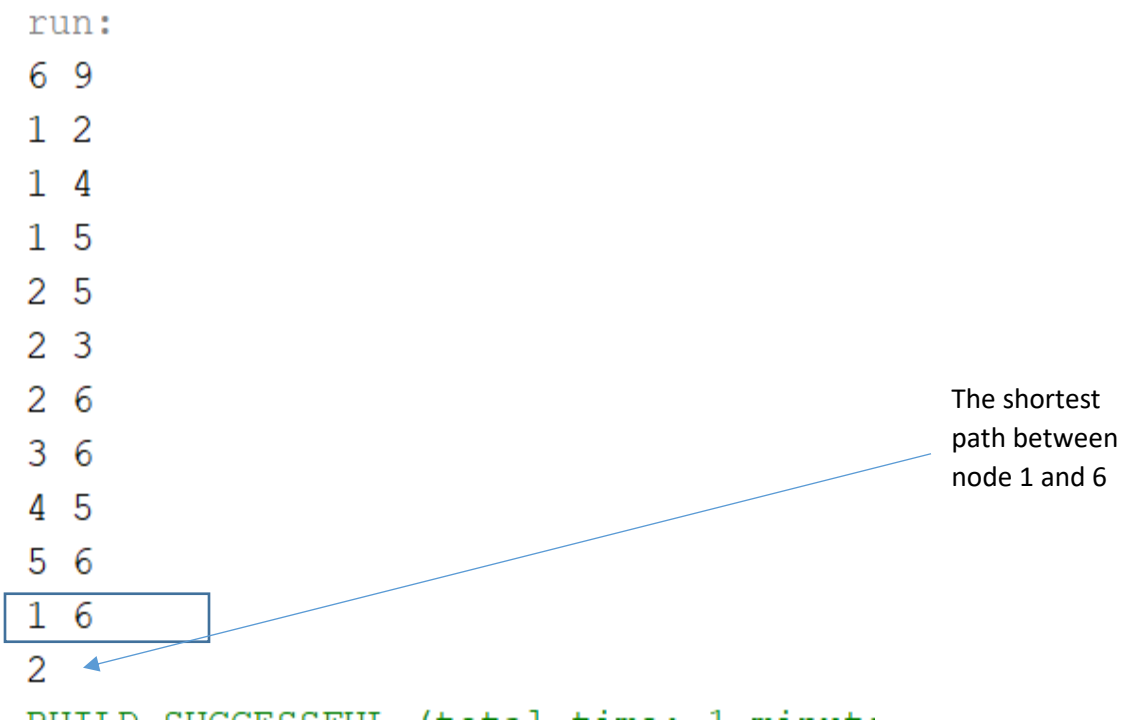
The procedures for the algorithm:

i. Accept graph information from the user i.e.
   a. First accept the number of nodes and number of edges from the user
   b. Accept each node and its adjacent node consecutively
   c. Since it is undirected graph add the reverse adjacency internally. E.g. if node2 is adjacent to node1 as given from the user, then since the reverse is also true, add node1 as adjacent of node2. Do this until all nodes are inputted from the user
   d. Accept the starting and destination node
ii. Initialize the distance to all the nodes as infinity except the starting node which is initialized as 0
iii. Add the starting node a to que
iv. Then until the que is empty do the following
   a. Dequeue a node and find all the adjacent nodes o the dequeued node. If a node is undiscovered then add it the queue
   b. For each adjacent node update the distance by adding 1 from the parent node
   c. If destination is found stop traversing

v. Finally display the minimum distance which is the minimum number of flight segments because it is unweighted edge.
Note:

Sample Input and output:





Note:
An input validation that implements the given constraints is not implemented. I have assumed that the user is aware of the constraints and how the inputs are given to the program.