## Design and Analysis of Algorithms

## CS375 Spring 2016

## Theory Assignment 3

Release Date: 3/18/2016

Due: 4/6/2016 (Wednesday) at start of class

Remember to include the following statement at the start of your answers with a signature by the side.

“I have done this assignment completely on my own. I have not copied it, nor have I given my solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of **0** for the involved assignment for my first offense and that I will receive a grade of **“F” for the course** for any additional offense.”

All solutions of theory assignments must be typed (no handwritten solutions) and submitted in hard copy. Advance electronic submission to the TA is acceptable if the student is expected to miss the class on the due date.

1. [10%] Suppose the capacity of the knapsack is 30 and the set of items S ={(*item1,* 5, $50), (*item2*, 20, $140), (*item3*, 10, $60), (*item4*, 10, $80)} where each element of set S represents (item, weight, profit). Find an optimal solution for the fractional knapsack problem using the greedy algorithm introduced in class. Show both the order in which the items are selected and the optimal solution you find.

Let Rn = Value/Capacity of each item

R1 = $10; R2 = $7; R3 = $6; R4 = $8

Select item1, Total Value = $50; Remaining Capacity = 25;

Select item 4, Total Value = $130; Remaining Capacity = 15;

Select ¾ of item 2, Total Value = $235, Remaining Capacity = 0;

Optimal Solution: item1+item4+3/4 of item 2; Total Value is $235

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | A | P | P | L | E |
|  | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | ↑0 | ↖ 1 | ↖ 1 | ←1 | ←1 |
| L | 0 | ↑0 | ↑1 | ↑1 | ↖ 2 | ←2 |
| A | 0 | ↖ 1 | ↑1 | ↑1 | ↑2 | ↑2 |
| T | 0 | ↑1 | ↑1 | ↑1 | ↑2 | ↑2 |
| E | 0 | ↑1 | ↑1 | ↑1 | ↑1 | ↖ 3 |

1. [30%] Find a longest common subsequence (LCS) between two strings X = APPLE and Y = PLATE using the dynamic programming algorithm discussed in class. Provide your solution steps in a table that includes the solutions for all possible subproblems and directed arrows (diagonal, left, and up arrows) needed to find an LCS in the end. [20%]. Use the recursive method discussed in class to find an LCS based on the information stored in the table. [5%] Note: show both an LCS and the path that leads to the LCS.

LCS: PLE

1. [10%] Briefly describe how to extend the depth first search (DFS) algorithm to determine whether a directed graph has a cycle (You may give a sketch of pseudo code and highlight the lines that are different from the original DFS algorithm. Comment your pseudo code to make it easy to understand).

DFS-VISIT(G,u, circle)

time ++

u.d = time

u.color = GREY

for each v in G.Adj[u]

if v.color == GREY //if v’s color is GREY, that means it has been discovered earlier,

circle = TRUE; // in the same path, thus it contains a circle

return circle; // return TRUE immediately

if v.color == WHITE

v.pi = u

DFS-VISIT(G,v)

u.color = BLACK

time ++

u.f = time

return circle;

DFS-FIND\_CIRCLE (G)

for each u in G.V //initialization

u.color = WHITE

u.pi = NIL

time = 0;

circle = FALSE; //declare a variable to track

for each u in G.V

if (circle == TRUE) return TRUE; //Once there is a circle, stop looking

if u.color == WHITE

circle = DFS-VISIT(G, u, circle)

return FLASE; //if circle has never been marked as TRUE

1. [15%] In graph theory, a **connected component** of an undirected graph is a subgraph in which any two vertices are reachable to each other (i.e., connected by at least one path), but the subgraph is not connected to any additional vertices in the supergraph. Briefly describe how to extend the breadth first search (BFS) algorithm to determine the number of connected components in an undirected graph (You may give a sketch of pseudo code and highlight the lines that are different from the original BFS algorithm. Comment your pseudo code to make it easy to understand).

COUNT(G)

component = 0 //store the total number of components

for each vertex u in G.V //mark color for all the vertices

u.color = white

u.d = infinity // u.d to hold the distance from s to u

u.pi = NIL // u.pi denotes the predecessor of u in BS-tree

for each vertex s in G.V

if s == white

component ++

BFS(G,s)

return component

BFS(G, s) // start the search from node s in graph G

for all u in G // initialization

u.color = white;

Q = empty // Q is an FIFO queue and it is initially empty

Enqueue(Q, s)

while ( *Q ≠ φ* )

*u* = *Dequeue*(*Q*)

for each *v* in *adj*[*u*]

if (*v*.*color* == white) // *v* has not been discovered

// from another path

*v*.*color* = gray

*v*.*d* = *u*.*d* +1

*v*.*π* = *u* // make *u* the predecessor or parent

// of *v* in the breadth first tree

*Enqueue*(*Q*, *v*)

*u.color* = black // u is now fully explored

1. [10%] Enumerate the nodes in the following graph in (a) BFS order and (b) DFS order, starting from node 1.



* 1. BFS: 1 -> 2,3,4 -> 5,6,7
  2. DFS: 1->2->5; 6; 3->7; 4

1. [25%] For each node *u* in an undirected graph *G*(*V*, *E*), let *sDegree*(*u*) be the sum of the degrees of the neighbors of *u*, that is,. Given an adjacency-list implementation of a graph *G*(*V*, *E*), provide pseudo code (comment your pseudo code to make it easy to understand) for an O(|*V*|+|*E*|) algorithm that outputs for each node *u* its *sDegree(u)* [20%], and briefly analyze the time complexity of your algorithm to justify it is O(|*V*|+|*E*|). [5%]

SUM(G)

For each u in G

u.d = 0 //u.d holds the degree of each vertex.

u.color = white

//Part A: calculate degrees for each vertex

Q = empty // Q is an FIFO queue and it is initially empty

Enqueue(Q, s)

while ( *Q ≠ φ* )

*u* = *Dequeue*(*Q*)

for each *v* in *adj*[*u*]

if (*v*.*color* == white) // *v* has not been discovered

// from another path

*v*.*color* = gray

u.d ++

v.d ++

*Enqueue*(*Q*, *v*)

*u.color* = black // u is now fully explored

// end of part A

For each u in G

u.sd = 0 //u.sd holds the sum of the degrees of neighbors of u

u.color = white

//Part B: calculate the sum

Q = empty // Q is an FIFO queue and it is initially empty

Enqueue(Q, s)

while ( *Q ≠ φ* )

*u* = *Dequeue*(*Q*)

for each *v* in *adj*[*u*]

if (*v*.*color* == white) // *v* has not been discovered

// from another path

*v*.*color* = gray

u.sd = u.sd + v.d //cumulate total sum of degree

v.sd = v.sd + u.d //for both vertices at the same time

*Enqueue*(*Q*, *v*)

*u.color* = black // u is now fully explored

// end of part B

for all u in G

print u.sd

// end of the function

In this code, I simply copy and modified BFS code twice, which are Part A and Part B. Since I did not add any additional loops inside these parts, and beside the Part A and Part B, I only perform initialization and printing, which cost O (|V|). The time complexity for BFS is O(|E|+|V|), so the time complexity for this algorithm should be O(|E|+|V|)\*2+O(|V|)\*3, which is O(|E|+|V|).

1. (Bonus Question) [30%] Consider a modification to the activity selection problem in which each activity *ai* has, in addition to a start and finish time, a value *vi*. The objective is no longer to maximize the number of compatible activities scheduled, but instead to maximize the total value of the compatible activities scheduled. This is called the weighted activity selection problem. Develop a bottom-up dynamic programming solution for this problem. Your solution should have a time complexity in O(*n*2), where *n* is the total number of activities in the input.

Hint: for the original problem, a subproblem can be created based on either (i) all activities excluding the one with the latest ending time, aj; or (ii) all activities that are compatible with aj (the one with the latest ending time). How do you quickly find out the activity with the next latest ending time but is still compatible with aj?

Your solution should contain the following parts (no need to show the pseudo code of your algorithm).

* 1. [10%] Clearly define the recursive function c[j] which represents the optimal value for the subproblem including activities ending when *aj* finishes. c[n] represents the optimal value for the original problem.

0 if j == 0

C[j] =

max (c[j-1], vj + c[qj]) if j >0

where qj is the largest index compatible with j; i.e. the ending time of qj is less than the starting time of j.

* 1. [10%] Demonstrate your solution on the following problem instance. Provide your solution steps in a table that contains optimal values for all subproblems.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a1 | a2 | a3 | a4 | a5 |
| Start Time | 0 | 1 | 3 | 5 | 7 |
| Finish Time | 2 | 4 | 5 | 8 | 11 |
| Value | 20 | 70 | 30 | 50 | 60 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| j | Qj | Vj | C[j-1] | C[qj] | C[qj] + vj | C[j] |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | $20 | 0 | 0 | $20 | $20 |
| 2 | 0 | $70 | $20 | 0 | $70 | $70 |
| 3 | 1 | $30 | $70 | $20 | $50 | $70 |
| 4 | 3 | $50 | $70 | $70 | $120 | $120 |
| 5 | 3 | $60 | $120 | $70 | $130 | $130 |

* 1. [10%] Briefly analyze and justify the time complexity of your solution.

T(Cj-1) + O(compare 2 number) + T (find qi and get Cqi) = T(Cj-1) + O(N)

T(n) = T(n-1) + O(N) for n >1

O (Cj) = O(N^2)