CS 375 Homework 3 Anchu A. Lee
October 31, 2017

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.

1. Suppose the capacity of the knapsack is 30 and the set of items $S = \{(item_1, 5, \$50), (item_2, 20, \$140), (item_3, 10, \$60), (item_4, 10, \$80)\}$ 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.

Item	$item_1$	$item_4$	$item_2$	$item_3$
Profit	50	80	140	60
Weight	5	10	20	10
Value	10	8	7	6

Greedy algorithm, take as much of higher value items as possible. Order items by value, if an item can fit entirely in the bag, take all of it. Otherwise take the fraction of the item that can fit. Add $item_1$ to the bag because it fits. Then add all of $item_4$ because it also fits. Now total weight is 15. The entirety of $item_2$ will not fit, so find the fraction: $(30 - 15)/20$ of $item_2$. Now the bag is full.
All of $item_1$, all of $item_4$, 0.75 of $item_2$.
Total profit: $50 + 80 + (0.75)140 = 235$

2. 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. Use the recursive method discussed in class to find an LCS based on the information stored in the table. Note: show both an LCS and the path that leads to the LCS.

	y	P	L	A	T	E
x	0	0	0	0	0	0
A	0	↑ 0	↑ 0	↖ 1	← 1	← 1
P	0	↖ 1	← 1	↑ 1	↑ 1	↑ 1
P	0	↖ 1	↑ 1	↑ 1	↑ 1	↑ 1
L	0	↑ 1	↖ 2	← 2	← 2	← 2
E	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3

Longest LCS is 3, and is PLE.

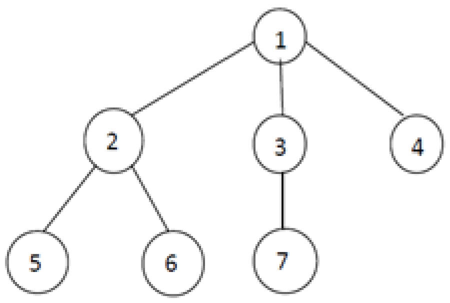
3. Briefly describe how to extend the depth first search (DFS) algorithm introduced in the class to determine whether a directed graph has a cycle (You may give a sketch of pseudo code and highlight the lines that are different from the original DFS algorithm. Comment your pseudo code to make it easy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

```
find_cycle(graph):
    # set up the graph like DFS
    for u in graph.vertices:
        u.color = White
        u.parent = None
    time = 0
    # run fine_cycle_helper on all vertices that are white
    for u in graph.vertices:
        if u.color == White:
            return find_cycle_helper(graph, u)

find_cycle_helper(graph, u):
    # update vertex information like DFS
    time = time + 1
    u.d = time
    # set color to gray - discovered
    u.color = Gray
    # for all adjacent vertices
    for v in graph.adjacent[u]:
        # if an adjacent vertex is gray, then there is a loop
        if v.color == Gray:
            return true
        # if vertex is white, search the vertex
        elif v.color == White:
            v.parent = u
            return find_cycle_helper(graph, v)
    v.color = Black
    time = time + 1
    u.f = time
    # otherwise no loop is found
    return false
```

4. 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 introduced in the class 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).

```
find_connected(graph, startnode):
    # set up graph like BFS
    for u in (graph.vertices).remove(startnode):
        u.color = White
        u.dist = -1
        u.parent = None
    startnode.color = Gray
    startnode.dist = 0
    startnode.parent = None
    # variable for counting number of connected subgraphs
    count_graphs = 0
    # goes through all vertices so it visits all vertices in the graph
    for u in graph.vertices:
        # skip vertex if it has been visited before
        if u.color = White or u == startnode:
            # add up the number of subgraphs
            count_graphs = count_graphs + find_connected_helper(graph, u)
    return count_graphs

find_connected_helper(graph, node):
    # add node to the queue
    FifoQueue.add(node)
    # variable that counts number of nodes in the subgraph
    node_count = 0
    # loop through the FifoQueue
    while FifoQueue.size not 0:
        # pop off from the queue
        u = FifoQueue.pop()
        # Typical BFS
        for v in u.adjacent:
            if v.color == White:
                v.color = Gray
                v.d = u.d + 1
                v.parent = u
                FifoQueue.add(v)
        u.color = Black
        # keep count of nodes
        node_count = node_count + 1
    # if the node is by itself, it is not connected
    if node_count >= 2:
        return 1
    else:
        return 0
```

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



BFS order: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ Done.
DFS order: $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 7 \rightarrow 4$ Done.

6. 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, $sDegree(u) = \sum_{(u,v)\in E} Degree(v)$. 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 $sDegree(u)$, and briefly analyze the time complexity of your algorithm to justify it is $O(|V| + |E|)$.

```
print_adjacent(adj):
    # array to hold degrees of nodes
    degrees = []
    # loop through all vertices in the adjacency list
    for u in adj:
        # temp variable to count number of adjacent nodes
        counter = 0
        # loop through linked list and count adjacent nodes
        while u.next not None:
            counter = counter + 1
        # save to array
        degrees.append(counter)
    # loop through all nodes again
    for i in range(0, adj.length):
        sum = 0
        # loop through linked list and add all the degrees
        while adj[i].next not None:
            sum = sum + degrees[u]
        # print
        print(sum)
```

First the function must compute the degrees of each node as they are not stored in the data structure. Once that is computed, then the function can compute the *sDegree* of each node. The function loops over each node ($O(|V|)$) and loops for every edge in order to sum all the degrees ($O(|E|)$) so then the function runs at $O(|V| + |E|)$ time.

7. Consider a modification to the activity selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. 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.

First the schedules must be ordered by earliest finished time. For a bottom up approach, the function must find the optimal solution for a_1, then for a_1 and a_2 and so on. The optimal solutions will be stored in an array $maxvalue[i]$ where i is the corresponding activity number. $maxvalue[0]$ then will act as if no activities are selected and will have

a value of 0. $maxvalue$ will also need a function called $lastfit()$ which will return the highest value activity that is compatible with the given activity start and finish times. $lastfit()$ will return 0 if no compatible activity is found. When $maxvalue[i]$ is called for any given i, the function chooses between two possible numbers: the previous $maxvalue$ entry, or $lastfit() + activities[i]$.
With the given problem, the solution will be:

0	a_1	a_2	a_3	a_4	a_5
0	20	70	70	120	130

When $i = 1$, the algorithm chooses between 0, the previous value, and 20, the last compatible activity plus the current activity. When $i = 2$, the algorithm chooses between 20 and 70. When $i = 3$ the algorithm chooses between 70 and $20 + 30$. When $i = 4$ the algorithm chooses between 70 and $70 + 50$. Lastly when $i = 5$, the algorithm chooses between 120 and $70 + 60$.
The function only needs to iterate through the list of activities once, however $lastfit$ is called for every activity. As $lastfit$ iterates through all previous activities to find a compatible activity, the time complexity of the function is $O(n^2)$ for n is the number of activities.