CS310: Advanced data structures and algorithmns

Fall 2013 Final exam – December 19, 2013

Instructor – Nurit Haspel

General Instructions

1. You may use any printed/written material. Cell phones must be silent and used only to show the time.
2. The work is to be your own and you are expected to adhere to the UMass Boston honor system.
3. The exam contains 4 questions. **Read each question carefully before you answer**.
4. Write your answers in the available spaces, using the back of the page if needed. Write clearly and concisely and try to avoid cursive.
5. Please explain your answers if needed **but do it briefly**.

Good Luck!

1. (20%) **Data structures and applications:** Parts (a) and (b) refer to the following scenario: You are working on a system security analysis program. Each user (identified by a unique String username) can login to a few systems (each with a unique string system name), and each system has many users. **Each of the users has the same username and password on all systems he/she can log into**. Note that any time a user can login to two systems, that condition can be considered a link between the systems from a security standpoint, since an invader can break into a user account on one system and then use that users password on the other system. You have been given the task of finding the sets of systems related by such links. For example, if system X is linked to system Y by user s, and system Y is linked to system Z by user t, then the set of systems X,Y,Z is considered as a linked-up set in this sense.

   a. (5%) What kind of Graph should we use to hold all such links between systems? directed or undirected? We do not have to store which users are involved in the links. Show how to allocate one such empty data structure using the JGraphT.

      This is an undirected graph since login is symmetric in this case.
      Graph<String,DefaultEdge> g = new SimpleGraph<String,DefaultEdge>();

   b. (6%) How can we represent the sets of systems related by the links described above, in a Java data structure? In particular, define a Java variable whose value is the full data structure you are proposing, but empty of data. Note that this is not itself a graph, just a group of sets. In fact the sets are disoint. Write a line of code to allocate such a data structure.

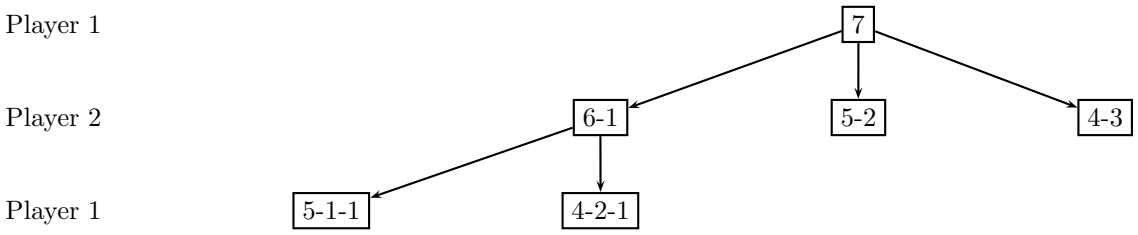
      Set<Set<String>> = new HashSet<Set<String>>();

   c. (6%) For a movie rental store, you are asked to simulate borrowing habits under various late-fee arrangements. For a given late-fee policy, customers keep movies for various lengths of time in a certain distribution, like the connection times in the modem simulator we studied. What data structure should you use to hold objects (named Rental) representing movies that are currently rented-out (and coming back at a certain future time in field returnTime) in your simulation?

      PriorityQueue<Rental>

   d. (3%) Additional requirement on class Rental for this to work:
      Implement comparable to be able to be a priority queue type.

2. (20%) **Games:** NIM-7 is a version of NIM with the following rules: We start out with a pile of 7 coins. Every player, at his/her turn, should split one pile into two piles **of different sizes**. The game ends when no pile can be further split. That is – all the piles are of size 1 or 2. The loser is the player who can no longer split a pile. There is no draw. The tree below shows the first two levels of the game tree, plus the first level below 6-1. Notice that 3-3-1 is not a valid option because it divides the pile of 6 into two piles of equal sizes.



a. (14%) Complete the game tree and assign values for all the nodes using minimax. Since there is no draw, assign a value of +1 for player 1 winning, and -1 for player 2 winning.

b. (6%) Would this game be sped up by dynamic programming? If so, show a part of the game tree and the actual position (mark the game positions on the part of the game tree) involved in a saved and later reused computation – no need to draw more than one example even if you can think of many.

Yes. For example – the state 1,2,4 appears in multiple branches and you can save it and reuse it.

3. (15%) Character coding

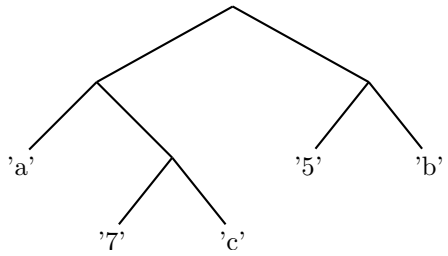
a. (5%) Which of the following sets of codes are prefix codes for a file containing only the characters, a, b, c and d?

   a. a = 1, b = 01, c = 001, d = 000
   b. a = 1, b = 01, c = 00, d = 11
   c. a = 1, b = 01, c = 101, d = 1001
   d. All of the above are valid
   e. None of the above is valid

a) is correct. In the other options 1 is a prefix of at least one other code.

b. (7%) Draw the tree that results from the following character distribution: No need to show the stages of the algorithm.

char	'5'	'7'	'a'	'b'	'c'
freq	10	7	14	8	4



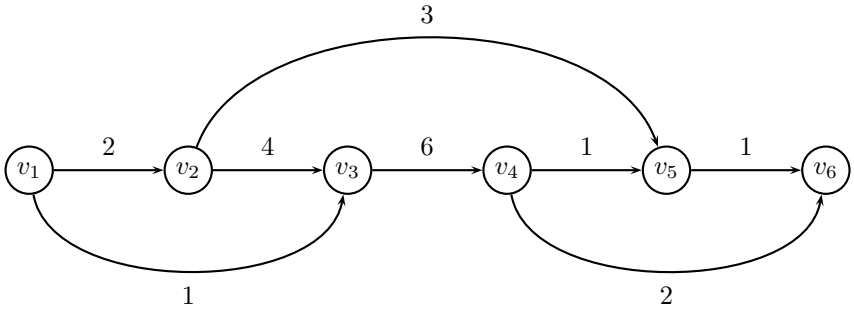
c. (3%) Fill the code for each character in the following table:

char	binary code
'5'	10
'7'	010
'a'	00
'b'	11
'c'	011

I accepted any solution that produced a correct Huffman tree, even if not the same as the one above.

4. (20%) **Dynamic Programming:** The weighted single source shortest path on a DAG (directed acyclic graph) can be solved using dynamic programming rather than Dijkstra's algorithm as follows:

- Topologically sort the graph and mark the distance of the source node as 0, the rest as ∞.
- For every node v in the topological order, set $d(v) = \min_{u,v \ s.t. \ (u,v)\in E}(d(u) + c_{u,v})$ where $c_{u,v}$ is the cost of the edge (u, v).



a. (10%) Find the shortest path from v_1 to all the other vertices in the graph above using dynamic programming. Fill the values in the table below.

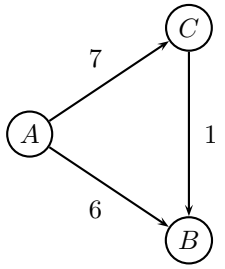
node	path length	predecessor
v_1	0	null
v_2	2	v_1
v_3	1	v_1
v_4	7	v_3
v_5	5	v_2
v_6	6	v_5

b. (10%) What is the run time of the algorithm as a function of $|V|$, the number of vertices and/or $|E|$, the number of edges. Explain briefly.

The run time is $O(|E|)$, where E is the number of edges. The topological sorting stage is $O(|E|)$ and the assignment of weights according to the topological order looks at every edge exactly once, and performs a constant number of operations per edge: Looking at the incoming vertex and comparing its distance + edge weight to the running maximum. Hence, this stage is also $O(|E|)$.

5. (25%) **Graphs:**

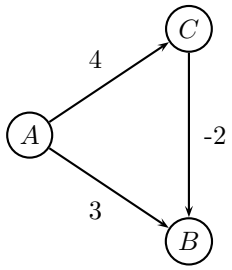
a. (7%) 4. In Dijkstra's algorithm, sometimes the next eyeball node is not adjacent to the last eyeball node. Explain how this can happen and give a concrete example (no more than 3-4 vertices).



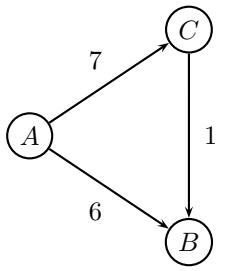
In this case, if we start from A, the eyeball order is A, then B and then C, even though there is no edge from B to C.

b. (6%) We discussed in class that Dijkstra's algorithm does not work if the graph has negative edges. However, a student suggested the following workaround: Add a positive weight X to all the edges such that they are now all positive and run Dijkstra's algorithm on the modified graph. Return the results as the shortest path for the original graph. Show why this does not work by assigning weights to the edges of the graph below and explain why adding a positive weight will give a different result than the original graph when calculating the shortest paths from A.

Here is an example, with weights.

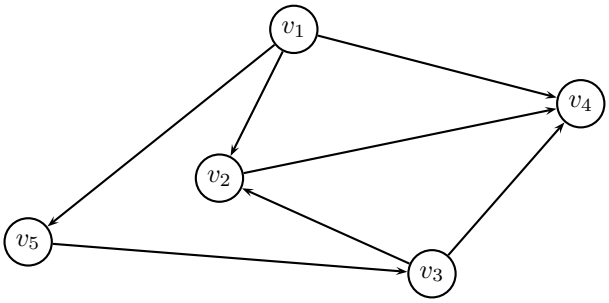


The shortest path from A to C is $A \to B \to C$ with a total weight of 2. If we add 3 to every edge to make all the weights positive, the shortest paths is now $A \to C$ with a total weight of 6.



Explanation: The path $A \to B \to C$ contains two edges, so if we add a constant weight to each edge, we increase its total weight by 6, where as the path $A \to C$ has only one edge, so its total weight increases only by 3. Generally, the more edges a path has, the more weight adds to it, so the results will not reflect the shortest path in the original graph.

c. (12%) Trace Depth-First search (DFS) on the following graph, starting from v_1. Use the same notation as in HW5. To make the search fully specified, if two or more options exist, select the one with the smaller index. Mark by * the edges that participate in the DFS tree.



Visit v_1 before edges
Visit edge $v_1 - v_2$ (tree edge)
Visit v_2 before edges
Visit edge $v_2 - v_4$ (tree edge)
Visit v_4 before edges
Visit v_4 after edges
Visit v_2 after edges
Visit edge $v_1 - v_4$
Visit edge $v_1 - v_5$ (tree edge)
Visit v_5 before edges
Visit edge $v_5 - v_3$ (tree edge)
Visit v_3 before edges
Visit edge $v_3 - v_2$
Visit edge $v_3 - v_4$
Visit v_3 after edges
Visit v_5 after edges
Visit v_1 after edges

The tree looks like this: