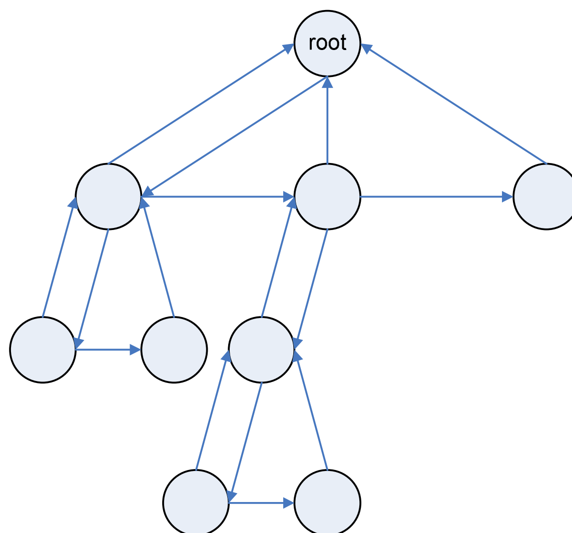
Jade Yu Cheng
ICS 311
Homework 9
Oct 2, 2008

Question for lecture 12

Problem 15-4 on p. 365

Planning a company party
Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4, as figure below. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.



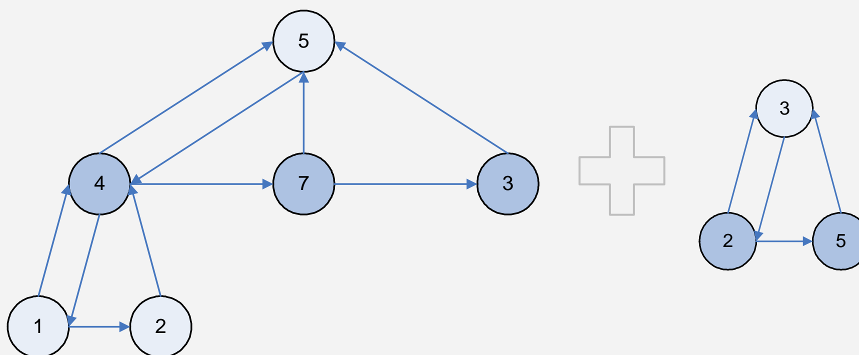
Answer: Consider the problem is to find the best combination of the nodes that give me the largest summation of conviviality value. The constraint is that the selected nodes don't have direct parent-child relationships with any other nodes in the selected collection.

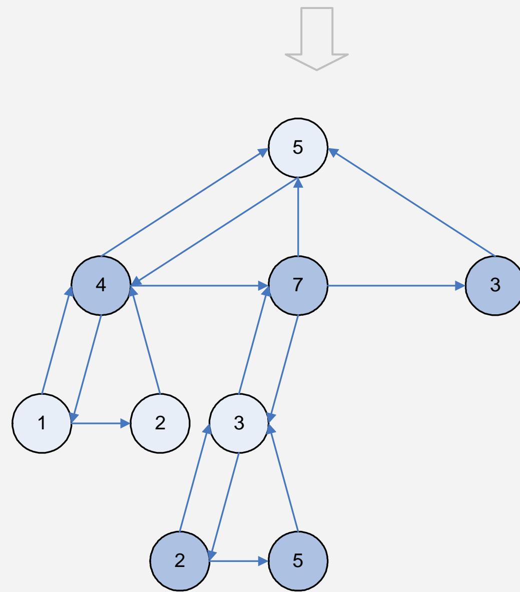
When $n = 1$, the tree structure has only root node. The problem in this case can be considered as the base case. We would always select it and the total conviviality value. $C = c_1$.

When n = 2, or n > 2 while the tree structure has a height of one. Or we can just say when the tree structure has a height of one. They are the parent node and the child node (nodes). The problem in this case is to find the larger conviviality value between the parent node and the child node (nodes). $C = \max\{c_1, c_2\}$, or $C = \max\{c_1, (c_2 + c_3)\}$.

When the tree structure has a height larger than one, we can divide the problem into parts. If we don't consider the sub-trees, the problem would belong to one of the two categories above. We would easily find out the selection solution for the root part of the tree. We assign it as $C = c_{pg}$. At the same time there exist the rest of the nodes that sum up to c_{pb}. Then, we consider sub-trees. There are two cases that are independent from each other.

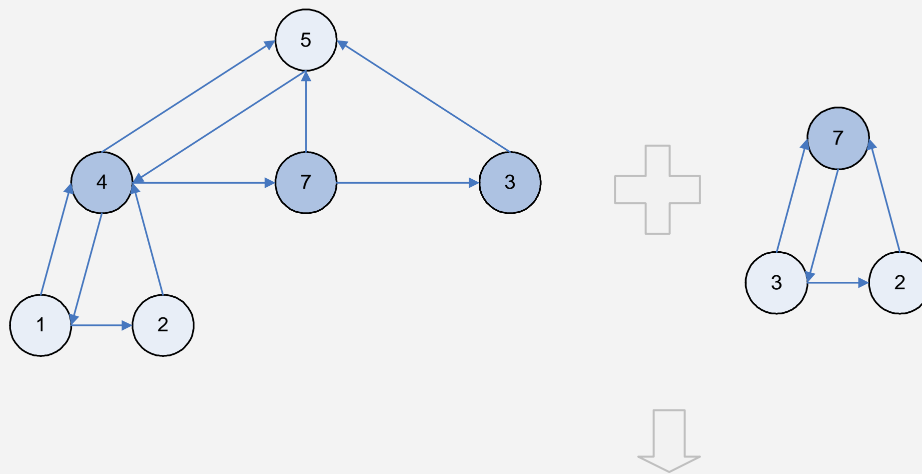
Case 1. If the selecting solution for the sub-tree that we are going to add does not conflict with the selecting solution for the root part, we simply add the sub-tree, and sum up the conviviality values. $C = c_{pg} + c_{cg}$, where the c_{cg} represents the best conviviality value summation of the particular child sub-tree. We can put it as an example below:

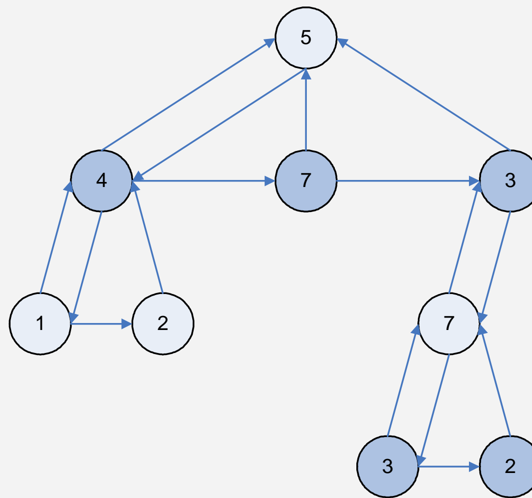




Case 2. If the selecting solution for the sub-tree that we are going to add does conflict with the selecting solution for the root part, we can't simply add the sub-tree, and sum up the conviviality values. Instead we need to examine the payoff of different combinations.

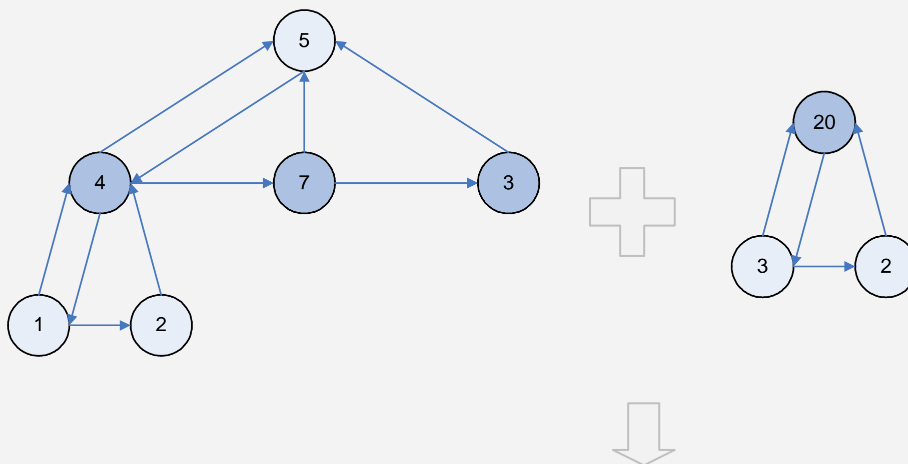
We really have two ways to handle it. One option is the combination of $c_{pg} + c_{cb}$, where c_{cb} represents the conviviality value of nodes that are not contained in the good selecting solution for the child tree. We can put it as an example below:

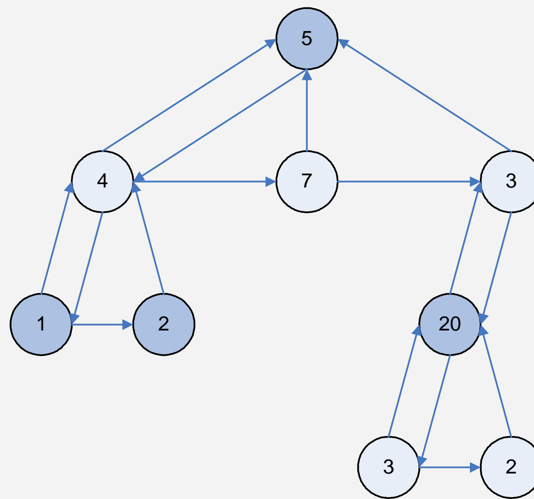




We put the combined solution for this particular parent tree and child tree as $c_{pg} + c_{cb}$ because $c_{pg} + c_{cb} = 4 + 7 + 3 + 2 + 3 = 19$, while $c_{pb} + c_{cg} = 5 + 1 + 2 + 7 = 15$. Therefore we sacrifice the higher term in the child sub-tree, while keeping the parent tree as it is.

The other option is the combination of $c_{pb} + c_{cg}$, where c_{cg} represents the conviviality value of nodes that are contained in the good selecting solution for the child tree. We can put it as an example below:





We put the combined solution for this particular parent tree and child tree as $c_{pb} + c_{cg}$ because $c_{cb} + c_{cg} = 5 + 1 + 2 + 20 = 28$, while $c_{pg} + c_{cb} = 4 + 7 + 3 + 3 + 2 = 19$. Therefore we sacrifice the higher term in the parent tree, while keeping the child sub-tree as it is.

Apparently, the overall best solution contains the best solutions of sub-problems. In order to compare and choose the best solution of the tree structure as in the last figure above, I need to know the best solution of the child sub-tree. The first Hallmark of Dynamic-programming is the optimal substructure. An optimal solution to a problem (instance) contains optimal solutions to sub-problems.

Also, to compute all the combinations for tree structures that have different heights, I need to repeat some of the sub-problem works. For example, in order to find the best solution of the tree structure as in the last figure above, I need to know maybe first compute the left sub-tree that contains value 1, 2, and 4. The when we consider the problem of how to incorporate the right child sub-tree, we already assume that the left sub-tree is a part of the parent tree. At this moment, we don't need to compute the left sub-tree more than once. Instead we would store the result of the solution of the left sub-tree somewhere. When the solution for this sub-tree is needed later on, we can just use it. This feature matches with the second Hallmark of Dynamic-programming—Overlapping sub-problems. A recursive solution contains a "small" number of distinct sub-problems repeated many times.

Therefore, the planning a company party problem is a good candidate of using dynamic programming. Based on this thought, I can implement the Memoization algorithm. I can try keeping a list to store the sub-problems that will be gradually computed and will be used more than once while computing the larger sets of inputs. If we consider the operation numbers of all the elements in the list I

would get a collection of h elements, where h is the height of the tree structure. The runtime of each element is either $O(1)$ or $O(2)$ because we need to do the comparison only two times for the most, comparing the summation of $c_{pg} + c_{cb}$ and $c_{pb} + c_{cg}$. We don't have control of the tree structure. If it looks more like a linked list, h would be more close to the value of n, $h = O(n)$. If it's the opposite, then h would be a lot smaller than n, $h < O(n)$. Therefore the worst case runtime of the problem as a whole would be $O(n)$.