SUBMITTED BY Aquino, Kurt Neil

Choo, Beatris Mariell

Choy, Seaver Matthew

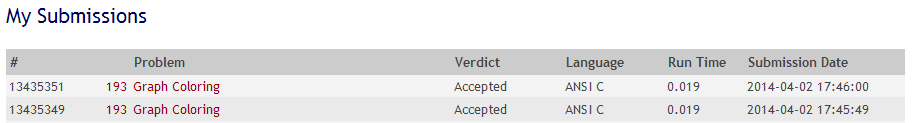
SUBMITTED TO Sir Juan Lorenzo L. Hagad

Graph Coloring

UVA Online Judge

DASALGO Problem Code: G3 Status: Accepted

UVA Problem 193 – Graph Coloring Runtime: 0.019 seconds



(Source code with comments at the last page)

Discussion

The problem was to write a program that tries to find an optimal coloring for a given graph. Colors are applied to the nodes of the graph and the only available colors are black and white. The coloring of the graph is called optimal if a maximum of nodes is black. The coloring is restricted by the rule that no two connected nodes may be black.

The specifications imply the use of nodes or linked lists in order to make the graph. For the sake of simplification, I chose to use a two dimensional array instead so that pairing together two nodes would become as simple as setting their coordinates.

For the input, the first line should include the number of graphs *m*, with this we can determine how many times we must repeat the program in order to color (in this case mark) all of the nodes. The second line should include the number of nodes *n,* and the number of edges *k.* The connection of the nodes is given by user input as well. With the parts of the graph(s) complete, all we need to determine is the maximum number of black colored nodes (considering the given rule), as well as their indices.

Here is the step by step algorithm:

1. Determine number of graphs
2. Determine number of nodes and edges
3. Create an array storing the nodes’ indices and a two dimensional array for their pairing/connection
4. Create a function that would set the colors of each node by:

* Setting the first node to color black, while keeping count of the number of black colored nodes.
* Checking if it’s next pair (numerical order) is already ordered

1. if not, set it to white
2. otherwise set it as unmarked

* Check the rest of the unmarked nodes.
* Repeat until all of the nodes have been colored.
* Finally, copy the indices of the black colored nodes to a final array to pass as a result

The confusing part of the code is how to determine the colors of the different branches the current node is connected to. Before proceeding with marking any of them, we have to first determine if they are connected to any black colored nodes. But the answer to the problem was simple, since we already made an array of the nodes, keeping track of their markings; all we have to do is to refer to that list (their indices) when checking if a node is directly connected to any black colored node, thus avoiding the violation of the rule that no two black colored nodes should be connected.

The only thing we did in order to modify the program was to give detailed variables, proper indention, optimization of some useless bits if code, and overall the translation (since the source was in Spanish, which made things a little difficult to understand). As for the compilation, the original code missed a few libraries for some of the functions used in the code, therefore slowing down the compile time.

References

Cespedes, J.(n.d.) *Graph Coloring*. cespedes.org. Retrieved April 02, 2014 from <http://www.cespedes.org/uva-acm/1/193>”

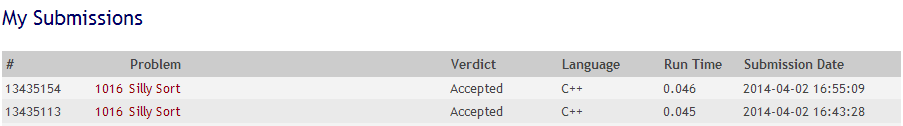
"UVa Online Judge." UVa Online Judge. N.p., n.d. Web. 2 Apr. 2014. <http://uva.onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page= show\_problem&problem=129>

Silly Sort

UVA Online Judge

DASALGO Problem Code: SRT3 Status: Accepted

UVA Problem 1016 – Silly Sort Runtime: 0.045 seconds



(Source code with comments at the last page)

Discussion

The problem is, given an array with unsorted elements, given that each swap would cost the sum of the 2 elements’ values; find the minimal cost to sort the sequence of the array.

First, get the lowest possible sum for sorting the array, we must switch the lowest with the element that was supposed to be in the lowest index. If the lowest is already at the right index, it will find the next lowest switch. However, there are cases in which the lowest may not be used, but when used, the sum is lower than that of the next lowest. Therefore, we would need to replace the lowest with the next lowest to see whether the values that would come out is higher or lower.

Another way to explain it is this. Since only two elements can be swapped at a time, all of the elements that are not in the right array index should be swapped. Therefore, all of them should be added up. The next question is which one is our constant swapper? The answer is the lowest value in the array that is not in the right position.

In the end, the sum would be the all elements of the array that are not in the right index and the number of times the lowest value is used.

However, there are special cases in which the lowest value is not used, but when used produces a sum lower than that of the lowest wrong-indexed value. Therefore, we would have to "borrow" it so that it can be used as a replacement for the swapping.

We faced minor problems during the submission of the code, we always exceeded the time limit and some of our java programs would not even compile nor run when submitted to the online judge. We solved this by looking of a c++ version of our code, edited out some redundant variables, applied some of our code into it, and added some comments for the explanations of each block of code.

References

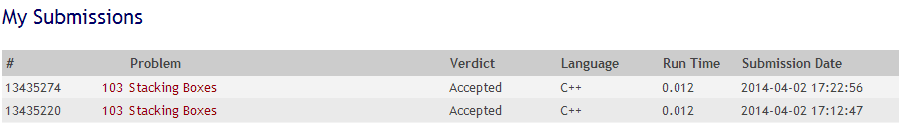
"I solved a problem." : Silly Sort (Spoj. N.p., n.d. Web. 2 Apr. 2014. <http://isolvedaproblem.blogspot.com/2012/02/sillysort.html>.

Stacking Boxes

UVA Online Judge

DASALGO Problem Code: S1 Status: Accepted

UVA Problem 103 – Stacking Boxes Runtime: 0.012 seconds



(Source code with comments at the last page)

Discussion

Box 1 can be stacked to Box 2 if all of the dimensions of Box 1 are lower than Box 2. This is the condition for boxes to be stackable.

First, two numbers are asked for input, these numbers corresponds to the number of boxes and the dimensions it has respectively. Immediately after the dimensions of the box are inputted, it is sorted. After which, all boxes are then compared, to see which one is higher than the other, this sorts the boxes in terms of highest to lowest dimensions. Box 1 can be stacked to Box 2 if all of the dimensions of Box 1 are lower than Box 2. To get the maximum number of boxes that can be stacked from the given input, two arrays are created to store the information about the boxes. First array contains the stackable boxes. Second array contains the indexes of the stackable boxes. With this information, we can get the most number of stackable boxes and their indexes. The first array gets its values changed when the box can be stacked on by other boxes, and the boxes before it have lower ability to stack. The second array gets its values by getting the indexes of the boxes that can be stacked. After the values are changed, it is time to put the stackable boxes in order. To do this, we get the index of the highest value in the first array, meaning it is the highest possible stacks we can have. After getting the index of the highest, we use this index in the second array to get the index of the box that can have the highest stacks. By replacing the recent value of the second array to the index of the next box, we can recursively find the boxes to be placed in order until a value is repeated. We then use the gotten indexes to find the values in the original array. The values in the original array are then pushed into a stack. The output is the maximum number of boxes that can be stacked, and the corresponding order by getting the values from the stack.

References

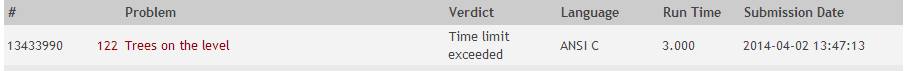
"programming contest problems and solutions." : ACM. N.p., n.d. Web. 2 Apr. 2014. <http://acm-solution.blogspot.com/2010/11/acm-uva-103-stacking-boxes.html>.

Trees on the Level

UVA Online Judge

DASALGO Problem Code: T1 Status: Time Limit Exceeded

UVA Problem 122 – Trees on the Level Runtime: 3 seconds



(Source code with comments at the last page)

Discussion

Table 1: Test Cases

|  |  |  |
| --- | --- | --- |
| (3,)() | Tree only contains root node | 3 |
| (5,RRL) (6,R) (7,L) () | Tree is not completely specified | Not complete |
| (7,L) (5,) (4,LL) (5,RR) (10,RRR) (3,R) (8,RL) (9,LR) () | Tree is completely specified but is not full | 5 7 3 4 9 8 5 10 |
| (8,L) (5,R) (9, LL) (3,) (4,LR) (1,RL)(2,RR)() | Tree is full and completely specified | 3 8 5 9 4 1 2 |
| (1,)(2,L)(3,LL)(4,LLL)(5,LLLL)() | Tree is one sided and looks linear | 1 2 3 4 5 |

A binary tree can easily be represented as an array. A full binary tree with at most 255 nodes has at most 8 levels.

The code starts by going through the characters in the input and reading them accordingly. The character ‘(‘ specifies the start of the input for a single node. When the program reads this character, it checks if this character is followed by an integer. If not, then, assuming all input is correct, then the next character is a ‘)’, which indicates the end of input for a single tree. Otherwise, it stores this input in a temporary integer variable. It then continues to read characters until it detects the ‘,’ character, at which point the program will start calculating for the index of the node. The next input, again, assuming that all input is correct, is a string of ‘R’’s and ‘L’’s that describe the placement of the node in the tree. The index calculation starts with a value of 0. For every R that the program reads, it increases the index by (2v+2) where v is the index. Every L that it reads increases it by (2v+1). When the program reads the ‘)’ character after this string, it stores the value of the node into its calculated index and marks any possible children nodes (at indexes 2i+2 and 2i+1, where i is the index of the node) as ‘has parent’ (where 0 means no parent, and 1 means has parent). It then increments the number of times the node at this placement has been given a value.

To check whether the tree is complete or not, the program simply iterates over the nodes and checks the number of times it has been given a value and the Boolean stating whether or not it has a parent. If the node being checked has been given a value and has no parent, then the tree is automatically tagged as incomplete and the words “not complete” are printed onto the output.

If the tree is complete, the program iterates over the nodes again, starting at index 0. Any node that has been given a value is then printed onto the output.

UVA did not accept this code, stating that it exceeded the given time limit.

References

"UVa Online Judge." UVa Online Judge. N.p., n.d. Web. 2 Apr. 2014. <http://uva.onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page= show\_problem&problem=58>