**Description**

As the task described, for a certain unit time node, we need to get the maximum value for these 6-advertisement boards display. For example, in the third second, at this moment, we have six boards are displaying 6 different advertisements contents, respectively.  We should choose one advertisement for every selected board from their scheduled requests. Every board has maximum three candidate schedule requests. The maximum weight can be calculated by multiplying the value of the advertisement content and the value of the board itself. And we have one constraints is that no two board shows the same advertisement.

**DFS**

Cuz for the different time node, the displaying locations are different, so we need to use  hashmap data structure to record the relationship between the different time node and the different advertisement location.

Based on the ScheduleRequest rules and features, each board has maximum three candidate advertisements for choosing and scheduling. So we use another map to record the map relationship between the advertisement location and the ScheduleRequest

So the code is shown

HashMap<Integer, HashMap<String, ArrayList<ScheduleRequest>>> map = new HashMap<Integer, HashMap<String, ArrayList<ScheduleRequest>>>();

At the beginning, the map is empty, we put all the input into a HashMap, traverse each input ScheduleRequest from starttime to endtime. If the new timenode didn’t exist in this map, we put the the timenode as the key and a new hashmap as a value into the map. And then, we add a schedulerequest in this inner hashmap.

Then we need to figure our corner case. For a certain selection choice, the time node, if we check the map don’t have the time node, we can know that there is no ScheduleRequest at this time can be provided for us to make the decision.

Then we use depth first search  to traverse these boards.  Depth-First search is a specific form of backtracking related to searching tree structures. Cuz each board has three candidates ScheduleRequest and we need to avoid the contentID conflict.  So we need a hashset to store the visited advertisement contentID.

A good way to illustrate the structure , consider the following structure.

图片

Traversing the tree then becomes a natural way to think about the recursive nature of the algorithm

Let's say we have 4 boards , at the first level , first board we have three candidate SR , and for the first SR as the start, we go down to the next level, next boarad, 图片讲解

So basically the idea would be to sequence through an  array using an index , that for each stage of our recursion(which in this case would be the "levels" of the tree). For the dfs procedure we use index to record which board we are visiting and the dfs function start from index = 0 ;   if the index is larger than the size of the boards or areas return.  Then we use dfs within index + 1 , which means we go to the next level , the next board to continue the dfs function. So note that we only need to recurse N times. For every level level, we need to maintain Therefore you would have some recursive function like the following:

When we reach the last location, we compare the maxvalue to current value. If current value if larger than the current value, we replace the max value as the current value. After every DFS level finished, we need to remove the previous location id to maintain the current res and visited set correct.

**Alphabet pruning algorithm**

During the dfs process, I found an interesting problem. I begin to think about if this method can get the optimal solution. The answer is no, it still has possiblity to optimize .  The dfs seems continually search and traverse by level and from the first board to the last board in order to get the maximum weight, however it involves many useless traverse procedures.

In here, I plan to use alphabet pruning algorithm to reduce the computation times and make the dfs function more effectively.  Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. At the beginning we have an array to store the locations.size + 1 elements, with length locations.size + 1. We firstly decide to store the max value of each board. For these elements are all the max value of each board without considering the contentID conflict. We can consider its sum value as the global max value. The stored elements from first to the end are the sum of the all values of all boards and the sum of values of last three values, last two values, last one, respectively, and the last element of array is zero.  We can see the figure, I can easily display.

In the first dfs round , we can get a current maximum value , and this result is based on the contentID conflict considered and  when we calculate the first board and the second board we can get the maximum of these two boards, at this time we make temp maximum value  + the corresponding value stored in array we have , if the value is still smaller than the current max value , then we can know the temp maximum value is useless ,cuz even if we choose the following boards max value without considering the content conflict ,the value is still smaller than the maximum value we get in the first dfs round. So we can cut off this brunch and it definitely save the process time and improve the efficiency.

In each board, we need to find the max score in these three scores for each time node. So we need another hashmap to maintain the optimal solution for each unit time node.

HashMap<Integer, HashMap<String, Integer>> contentmax = new HashMap<Integer, HashMap<String, Integer>>();

And to make sure for the contentmax the sr.score is always the max value. If we traverse the score is larger, we update the score and push the new score into the hashmap for contentmax.