Report for Programming Problem 1 - 2048

Team: 2018283497_2015235572

2015235572 - José Miguel Saraiva Monteiro
2018283497 - Daniel Roque Pereira

1. Algorithm description

The very basic of the approach used, is to try every possible move (up, down, left, right), on the board and trying to solve the board that way, relying on recursion. The base case is when the board has no remaining moves. Each recursive step the first thing done is the specified move after which the board is scanned to see if it changed to a solved state. After that there are additional checks being made to confirm the maximum moves haven't been exceeded, the new board generated is different from the starting one and if there hasn't been a better solution found already. If any of these conditions have been found, the function *executeStep* is called another three to four times to perform another set of possible moves, depending if after the last move there is the possibility to execute the same move again, and produce a different result.

When talking about speed up tricks, one of them was to save the data in a one dimension vector, which is more efficient than a two dimensions one. The advantages of this approach is that a single dimension vector consumes less memory than the multi dimension counterpart. The access of the data is also faster, considering that the data is in a single memory block, instead of smaller, more spread blocks. In the beginning, at the time of reading the input, every element that's not zero is added in a temporary variable and after the board is fully read it's checked if the sum is a power of two which determines if the board is solvable (e.g. a 2x2 board with the numbers [2,0,4,0] is impossible to solve). After the first solution is found Backtracking is used and the number of moves used is saved in a global variable. For each recursive call if the difference of the *Max Moves* and the *Current Moves* is equal or greater than the moves used in the current best solution, the program aborts the current function call and begins to search for another solution where the number of used moves is lower than the best solution found. Following each board movement, the new board is compared to the old one, and if they are identical, the program concludes that the board hasn't changed, and there is no point in continuing down that path because a move was already wasted.
After extracting the values of the row/column the approach used to sum the equal values consists in storing all the non zero values in an array and if any adjacent numbers are equal the second appearance of the value is deleted from the array and their sum is stored in a new array.After all the sums are done we add 0s to the new array until it matches the size of a row/column of the gameboard.

The last trick used to speed up the program is to check when adding each row or column, to verify if moving the board in the same way again will result in any piece movement, or if doing it a movement would be wasted.

2. Data Structures

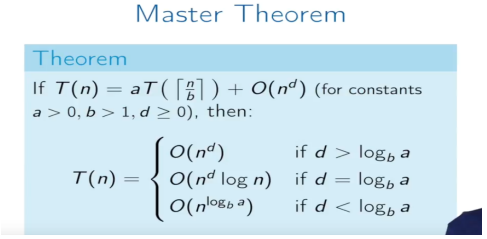
The data structure used in this approach is an array with a dynamic size. The array size is determined by the total amount of possible squares in the board (boardSize²).

3. Correctness

The solution is thought of in a way that it can satisfy all the possible test cases taking in account that all the inputs from the board are powers of 2. It's known that a case where the sum of all numbers is not a power of 2 is instantaneously considered an invalid solution.
Following these 2 verifications, the program can now move to the board solving itself. Considering the algorithms applied, mooshak's test cases and tests designed by the group, the board is always solved correctly, since it always iterates through all the possible movements until a solution is found, or not, in which case the board is unsolvable.
When excluding recursive paths, at the time of executing moves in a certain direction, it's verified if moving again in the same direction will produce a different board than the one that's being currently generated. If that wouldn't be the case the repeated movement is not executed, considering it'd be a redundant move. Another recursive path exclusion being done is after performing every board movement, it's checked if the new board is exactly the same as the old one, which means that the move executed is redundant and going down that path would only waste resources. At last, after the first solution has been found, every path that reaches the same moves used as in the best solution, is excluded. This last exclusion doesn't affect the board solvability, considering that a *best solution* has already been found, and the program is just interested in trying to find a better solution, therefore using less moves than the current best solution.

4. Algorithm Analysis

After an analysis of the algorithm using the master theorem supposing the following:
a-> number of subproblems
b-> size of the new problems
d-> complexity of the algorithm
The expression T(n) = 4T(n/2) + O(n) was reached. However,this analysis was incorrect since we could suppose that in a worst case one of our subproblems does nothing and our b would end up being 1.



We would then end up in a situation where comparing d and $\log_b a$ would be impossible since 1 can never be reached because $\log_1 4$ is infinity which means the master theorem fails to solve the problem of our recursion algorithm's temporal analysis.
In the worst case our algorithm's temporal complexity will be 4^n since the cuts made won't affect the algorithm and it will end up being 4 recursive calls to the power of the number of moves that a player is allowed to.
In the average case,it ends up in a situation where when a solution is found it will stop making the recursive calls related to the parent of that tree node since the best solution related to it has already been found.That means we will end up cutting to a point where the solution will be found in 4^{n-t} where t is the number of branches that have already found a solution.Some of the branches can also end up being reduce 3 recursive calls(explained in best case).
For the best case,if we join both the fact that all our movements do not need you to repeat the last move you executed we are able to exclude 1 recursive call and end up with only 3^n cases that end up being reduced by the same cut used on the average case to a point where it reaches 3^{n-t} being our time complexity.

Spatial complexity-wise we can affirm that the complexity is 4^n due to the fact that a 1d vector is created for each recursive call.

5. References

- https://en.cppreference.com
- Class Powerpoints
- https://stackoverflow.com/questions/17259877/1d-or-2d-array-whats-faster