An Algorithmic Solution to the Couple-Casino Game

Another 2-player game from Combinatorial Game Theory

Subhobrata Dey
Computer Science & Engineering
Jadavpur University
Kolkata,India
sbcd90@gmail.com

***Abstract*—**Casino Games are very important topics of research in the field of probability theory & combinatorial game theory[2]. Here I discuss about another highly interesting 2-player casino game known as "Couple-Casino Game" & present an algorithm for maximizing the amount of profit that can be achieved on playing this game. The solution can also be used for several other practical algorithm problems like graph theory problems involving the generation of a maximum cost spanning tree for a specified number of vertices etc.. The algorithm has also been tested with a wide variety of test cases upon implementing it in C++ programming language.

Keywords-*2-player game; Linear Programming ; Dynamic Programming; Backtracking.*

I. Introduction

The topic of this paper is to provide a polynomial-time solution to the Couple-Casino game. Here is a brief description of the rules & regulations of the game:
<i> The game is a 2-player game played by a couple in a casino & assuming that a judge is present to decide how much profit they make.
<ii> A multiset of integers will be provided to the couple by the judge using the pseudo random number generators. Now, the two persons making the couple are to split the multiset into two parts, the sizes of which are specified by the judge.
<iii> They may do so in many ways, but the goal of the game is to minimize the difference of the sums of integers present in the two partitioned multisets, i.e. if the sums of integers present in the two sets be s1 & s2 respectively, then the goal of the game is to minimize

$$| s1 - s2 |.$$

The more a couple can minimize the more money they can earn. So, naturally the algorithm described here will minimize the term & thus in turn maximizing the profit. It may seem that the problem belongs to the NP-class & will have a non-polynomial time solution using backtracking. In such a case we need to generate all possible ways of splitting the given multiset, then check all other constraints & finally optimize our goal. Thus , the above algorithm requires exponential time complexity.
However,I here propose an algorithm which solves the given problem in polynomial time complexity. For this I use the technique of dynamic programming which increases the space complexity of the algorithm but in turn reduces time complexity.
Here,I utilize the Linear Programming approach to model the problem as a LPP, & then use dynamic programming technique to find the optimum goal & then use the backtracking[1] approach to find a feasible solution to the problem. Among the common algorithms used, is the 0/1 Knapsack problem [3] which is a classic example of a subset selection dynamic programming problem.

II. Assumption

Without loss of generality,i have made an assumption that the judge always provides the size of the largest partitioned multiset,so,the solution now works.
In the next couple of sections, I present complete description of my algorithm regarding the solution of Couple-Casino game.

III. 0/1 Knapsack Problem In Brief

The problem is considered to be one of the basic problems related to subset dynamic programming problems[1] & [4]. The major idea is to select a subset from a multiset maximizing/minimizing a given criteria for profit. This algorithm is the backbone of my approach to solve the Couple-Casino game. The algorithm pseudocode[4] presented below gives a vivid description of the solution:

```
dpknapsack(inputmultiset[],sumofelements,results[][])
        //initialize results[1] to 0
        for i=2 to inputmultiset.size()
                for j=1 to sumofelements
                        if(results[i-1][j]==1)
                                results[i][j]=1
                        else if(results[i-1][j-inputmultiset[i]]==1)
                                results[i][j]=1
        for j=1 to sumofelements
                if(results[inputmultiset.size()][j]==1)
                        maxi=j
        return(maxi)
```

Pseudocode of knapsack problem

The function defined above takes three parameters,the array inputmultiset[]which consists of the respective profits for each index. sumofelements is simply a given sum as input. The other 2-dimensional array results[][] store a boolean value indicating whether a given sum is possible to achieve from the given inputmultiset[]. Finally,in the last step, we maximize the sum & return it in the variable maxi. The algorithm for the problem seems to be pretty simple. Also one can easily say , that we have optimized our profit by generating all possible subsets of the given inputmultiset. Now,one thing to realize is that we could have solved this problem just by using backt-

racking to generate all possible 2^inputmultiset.size() subsets & then optimizing our constraint. In that case,the time complexity would have been exponential. So,definitely without compromising with the optimization we can improve our time complexity by increasing the space required. This is what Knapsack allows us to do. So, using this technique the time complexity turns out to be

O(inputmultiset.size()*sumofelements)

which is polynomial time complexity.The space complexity increases to the same as time complexity.

In the next section, I show how to model the Couple-Casino game in the form of 0/1 Knapack problem.

IV. Description Of The Dynamic Programming Solution

The Couple-Casino game problem can be modeled at first as a Linear Programming problem[1] where we need to optimize one constraint with respect to a couple of more constraints.Here, is the mathematical formulation of this problem:

maximize

$a_1 multiset_1 + a_2 multiset_2 + ...+ a_n multiset_n$

subject to, $a_1+a_2+.....+a_n <= maxi$, where maxi is given as input.

& $a_1, a_2, ..., a_n$ are elements of set {0,1}.

& $a_1 multiset_1 + a_2 multiset_2 + ...+ a_n multiset_n <= (sum/2)$,

where sum is the sum-total of all the elements of the multiset.

From the above formulation, we get the following algorithm:

```
dpccgame(multisetsize,multiset[],maxi,sum,solution[][][])
        //initialize all elements of solution[1] to 0
        solution[1][0][0]=1
        solution[1][multiset[1]][1]=1
        for i=2 to multisetsize
                for j=0 to sum
                        for u=0 to maxi
                                solution[i][j][u]=0
                                if(solution[i-1][j][u]==1)
                                        solution[i][j][u]=1
                                if(solution[i-1][j-multiset[i]][u-1]==1)
                                        solution[i][j][u]=1
        //finding a feasible optimized solution
        solmax=-1
        for i=0 to sum
                if((solution[multisetsize][i][maxi]==1)||
           (solution[multisetsize][i][multisetsize-maxi+1]==1))
                        if(solmax<i)
                                if(solution[multisetsize][i][maxi]==1)
                                        sol1max=maxi
                                else
                                        sol1max=multisetsize-maxi+1
                                solmax=i
        print solmax
        backtracking(solmax,sol1max,multisetsize,multiset,
                                                        solution)
        return
```

Pseudocode of dynamic programming approach

The initial call to the function will be of the form:

dpccgame(multisetsize,multiset,maxi,sum/2,solution)

The function takes as input multisetsize----the size of the multiset, multiset[]----the multiset itself, maxi----the maximum number of elements that one person of the two can choose from the multiset,sum---half the sum-total of all the elements of the multiset & solution[][][]---generates all possible solutions from which the optimized one must be chosen. solution[][][] is a 3-dimensional array, where 1st dimension denotes the indices of multiset[], 2nd dimension denotes the sum to be maximized,3rd dimension denotes the positive integral values <=maxi. Initially, solution[1][][] are all initialized to 0 . Logically,if we see , there are two possible options for each element of multiset[]. When the first element from the multiset[] is chosen then,

solution[1][multiset[1]][1]=1.

Again,if the first element is not chosen then,

solution[1][0][0]=1.

Now,the next few steps of the algorithm clearly works like the 0/1 Knapsack problem. The first two of the three nested loops is similar to the two nested loops of the 0/1 Knapsack problem. The first loop provides the indices to elements of multiset[] while the second loop maximizes our result. The third loop is an additional loop which satisfies the first given constraint of the LPP.

After filling the entire solution[][][] , we look to find the optimized way which maximizes the couples' profit. Variable solmax gives us the required solution. sol1max gives the size of the found partitioned set. Now, this is not enough. From here the couple will only get the maximum profit that they can get. But, they have to make the moves so that,this profit can be achieved. Thus we need to generate at least one feasible solution for the couple. For, this reason the backtracking procedure is used.

I discuss this procedure in the following section.

V. Description Of The Backtracking Procedure

The backtracking procedure provided here gives a feasible solution to the problem. There may be many other possible solution. All these solutions can be generated by using an additional loop along with the given procedure. However, for this problem only a single solution is adequate.

Here is the algorithm for it:

```
backtracking(solmax,sol1max,multisetsize,multiset[],
                                                        solution[][][])
    if(solution[multisetsize-1][solmax][sol1max]==1)
                backtracking(solmax,sol1max,multisetsize-1,
                                                multiset,solution)
    else
        if(solution[multisetsize-1][solmax-multiset[multisetsize]]
                                                [sol1max-1]==1)
                backtracking(solmax-multiset[mutisetsize],
                        sol1max-1,multisetsize-1,multiset,solution)
                print multiset[multisetsize]
```

Pseudocode of backtracking procedure

The initial call to the function is already provided in the dpccgame function.

The required parameters of the procedure are: solmax---the optimized result, sol1max---the optimized sub-result, & multiset[], solution[][][],multisetsize all of which are defined earlier. Now , before we discuss the backtracking procedure, lets first recheck what solution[][][] contains.

solution[i][j][u] gives using upto i^{th} element of multiset & taking atmost u elements of them,the maximum sum satisfying the constraints is j. Thus, we see that we must start from the

solution[multisetsize][solmax][sol1max]

& then move upward until no moves become feasible. Thus, we will get one of the two disjoint sets to be generated. Now, since the union of these two sets must be the entire multiset, so,we can very easily generate the other set as well. This give us the moves that the couple is going to use to get the maximum profit.Now,here is the description of the backtracking procedure in detail.

The procedure is almost just the reverse of the previous dynamic programming procedure. For each element of multiset[], we have two possible options—Either to take it or to leave it. When we are leaving that element,we are not including that element in our current set.

Thus, there is no need to print that element. Again,if we take it,then automatically (as well as logically) the parameters for the backtracking procedure changes & that element is printed. Since, this is a pseudocode so the terminating condition for the procedure is not specified explicitly. Whenever , one of the indices of solution[][][] becomes less than 0, the procedure must terminate.

Before,i finish discussing the algorithm,here is a list of a few less important points.

VI. Few Less Important Points

i> The ordering of the moves doesn't matter in the problem.Any order of moves which gives maximum profit for the couple is valid. Only each of the persons can take elements from a single set.

ii>The backtracking procedure can be developed such that the 2- player game can be finished in the minimal number of recursions. This will further optimize the time complexity of solution.The next section discusses issues with time & space complexity.

VII. Time Complexity Analysis

The algorithm presented above is a polynomial time algorithm & its simple to show that from the above pseudocodes.

Analysis of the dpccgame function shows that, there are three nested loops in the function at first. This gives the time complexity of the function as,

O(multisetsize*sum*maxi) , where all the variables have been described earlier.

The backtracking procedure has a linear time complexity of

O(multisetsize)

Now, we find that if the elements of the multiset[] are too large,then sum will also be large which increases the time complexity. So, it is necessary to assign low values of elements of multiset[] in order to utilize the polynomial time complexity of the algorithm.

VIII. Space Complexity Analysis

As discussed previously, we compromise with the space complexity in order to optimize the time complexity of the algorithm. Thus,space complexity which would have been linear, had the time complexity been exponential,now turns cubic in nature.

The space complexity of this algorithm would now become,like this:

O(multisetsize*sum*maxi),where all variables have been described earlier.

The problem that we saw with elements of multiset[] in time complexity becomes more profound here.This is one of the major difficulties of this algorithm. The solution however is same as that discussed in the time complexity analysis section.

IX. Conclusion

In the previous sections, I have discussed an algorithmic solution to yet another combinatorial 2-player game called Couple-Casino game. As I complete preparing this paper, I realize a few defects of this algorithm which I have pointed out in section VI & VII,VIII respectively. I have also not proved the lower bound time & space complexity for this problem. So, there may be also scope for improvement regarding this optimization problem.

X. Acknowledgment

I have not taken any idea regarding this problem & its solution from anyone personally. However , I have used a lot of web resources to learn the concepts of combinatorial games & their solutions. I have also studied a lot of algorithms for solving out this problem. Thus, I have no one to thank personally. However, its probably my love & interest in solving mathematical puzzles & combinatorial games like chess,sudoku etc. allowed me to think of writing a paper on combinatorial games.

XI. References

[1] T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein, Introduction To Algorithms, 3rd ed.,Phi Learning pvt. Ltd., pp-359-413 & pp-843-860.

[2] Lim Chu Wee, Introductory Combinatorial Game Theory, Web Tutorials, National University of Singapore.

[3] Lecture notes from Eastern Washington University on 0/1 Knapsack problem.

[4] Wikipedia notes on variation of Knapsack Problems.