Game Playing Algorithms

Group:cs20403

Game 1:

This program was based on a recursive implementation of analyzing the best possible move at any given stage in the course of the game.

We calculated the probability of any choice of a move at any given stage and proceeded on the basis of highest probability achieved from all the moves.

We had plans of including pattern recognition at a later stage after analyzing the performance of our game in the first round. (This was later achieved as mentioned below)

Game 2:

This program was restructured after the first round of the games. This program dealt extensively with the various patterns emerging during the course of the game.

The strategy was based on “Pattern Recognition” of familiar and winning moves and the program also tried to create favorable circumstances.

The pattern recognition is done utilizing the various combinations of coins in straight lines.

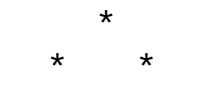
For example:

```
  *          *
* *    :--> One 3 {  * } , two 2s.
  *               *
```

Numerous patterns were identified making the task of identification of winning and losing patterns very easy. As these patterns occurred at every step in the recursive analysis, the task was just reduced to the identification of the already recognized patterns and then building favorable patterns on top of them. This bottom – up approach proved very efficient as was evident from the scores.

For example:-

is a combination of 3 coin triplets and 3 duplets. Such a combination comes unique and can be built upon as



a combination of 2 triplets, 1 quartet and 3 duplets of coins.

The patterns emerging from this strategy are unique under the constraints mentioned above.
The whole program has dealt with recurring emergences of these simple patterns, on the basis of which the strategy is molded into a favorable case as far as possible.

Game 3:

This program was an up gradation of the first strategy of recursion with "Mini-max" implementation .Also many redundancies of the recursive algorithm were removed and optimization was done to improve upon the time taken.

A distinguishing factor in this program was the detection of a "Sure win" condition apart from the probability considerations. I f a sure win was detected then the program followed its course to the winning move. This involved a "Greedy Approach". A little detail:
Sure win condition was determined when at a given stage in the game, for each possible move of the opponent we had at least one

move which could guarantee a sure win. Subsequently a flag was set
and the path of probability was abandoned immediately.

Game 4:

Involed: Recursion, bounding functions, dynamic programming memoization.

This was a program which involved heavy use of "Dynamic Programming" techniques .It was modified from playing a triangle to a square. As the sample size of the problem grew drastically, many measures were taken to reduce the time taken.

A new updating technique was developed which consisted of simply setting flags on the moves already made (and inside recursion too) to isolate them from the left valid moves. The idea of memoization was attempted through extensive use of files (this had to be abandoned at the last moment due to lack of time and some bugs).

Considering the symmetry of the sample space, a bounding function was implemented which picked one of every four consecutive moves initially in the game tree, which was then reduced to every alternate move being analyzed and finally when the number of coins reduced below a specific number every move was considered.

Game 5:

Involved:

Memoization, Dynamic Programming , Improved File implementation, Greedy approach.

This version of the game improved over the bugs of the last game and log files were used extensively to store(memoize) the already discovered moves.

Memoization :

Because of the large sample space storing the coin configurations at all stages of the game, would have lead to very bulky log files and traversal among them would have itself lead to increased time consumption. In order to overcome this bottleneck, we devised a methodology to code the board configurations using six integers (0-64) signifying each row. Each number was generated by considering the binary representation of the row (considering the initial row configuration to be 64 and empty to be zero with a transformation from 1 -> 0 upon picking)
For example.
1 1 0 1 1 1 == 60 (leftmost is the least significant bit)
0 1 1 0 0 0 == 6
0 0 0 0 0 0 == 0
1 1 1 1 1 1 == 63
1 0 0 0 0 0 == 1
0 0 0 0 0 1 == 32

So the array containing the board configuration will be
[60 6 0 63 1 32]

Along with this array a count of the number of wins and number of losses in the sub tree below this configuration was also embedded into the structure which was written into the file.

Another useful feature which made this information stored in the log file to be stored was the knowledge if the current board configuration provides a sure win for us or the player (if so).

Any coin could easily be mapped to its row and column as follows:
Row = coin no. /6
Col = (coin no-1) % 6
For example :-
For the board configuration being identified as
1 2 3 4 5 6
7 12
. ... 16 18

. . . . . .
36

If we are to identify the row and column of coin number 16, then the new configuration of the row could be updated as:

Row = 16/6
= 2 (after integer division)

Column = (16-1) % 6
= 15 % 6
= 3

So the row and column of coin number 16 is [2,3]

Board [Row] -= (short) pow(2 , col)

To up date the row number 2 , if suppose the initial configuration was 1 1 0 1 0 0 == 11

New configuration(in binary representation technique as developed by us

Board [2] -- = 2 ^ 3

== 11 -8
== 3

To improve the search time separate log files were created according to the number of coins left. Also separate files were created for the same configuration being put to the opponent and self. Along with the configuration no. of wins and no. of losses and flag for sure win were stored as a structure. This helped in reducing the search time in the game tree drastically.

Considering the colors applied to the coins the moves were so arranged that the moves generating negative points would only be made when no other move of more points was possible unless we had a sure win in sight.

```
        if player =1
                put a predetermined highest score move
                goto continue
        if player =2
        continue: update opponent's move
                  enter recursion
                   {   first check if the configuration of borad presented is
in file or not
                                if yes, use the data and go up the tree
                                if no, continue recursion and store the result
in specific file
                    }
```

The algorithm can be simplified as

```
        If (found new configuration )
        {
                Recursion
                Update in log files
                If(timeout)
                        Goto emergency
        }
        else
        {
                Find the value from log file and give the
                best possible move.
                        If(timeout)
                                Goto emergency

        }

Emergency(){
        If(surewin)
                {
                        Give sure win move
```

```
        }
    Else
        {
            Find out the best possible move in terms of
probability of winning
        }

    }

    Surewin()
        {
            If(player ==1 )
            {
                // we are deciding for our self in the
                //recursion analysis
            If ( for every possible move of the opponent there
    exists a move to a sure victory )

                Return sure win flag == 1
            }
            Else
            {
                If( player == opponent )
                {
                    if(there exists atleast one path which
                gives a sure win )
                    return sure win flag =1
                }
            }
```

Limitations:

The program suffered from excess time consumption due to repeated call of fread() and fwrite() function calls .

In order to improve time performance of the algorithm , the log file data which amount to only a few kilobytes of data, could be loaded into the memory (RAM) of the computer . This will drastically cut down the time consumption.

Individual contributions:

The development and implementation of the programming algorithm and gaming techniques was done collectively by both the group members , so the role of any one cannot be placed above the other. Thus the contribution can be considered to be equivalent .

Group Members:

Sharma Ashish 01010124
Anurag Rajat 01010102