Strategy File

REPORT ON STRATEGIES IMPLEMENTED & OPTIMISATIONS USED - ASSNs 1,2 & 3

AIM:

The objective is to design and implement a cost-effective algorithm to play a given coin game on a deterministic basis.

STRATEGY:

The obvious approaches to the problem are brute-force approach and the cognitive one.

The human mind tackles this problem by reducing the given configuration into simpler geometrical shapes, and use pre-conceived notions so as to develop a potential winning combination. This pattern recognition approach works for small numbers but becomes unwieldy for larger numbers.

The brute force algorithm is ruled out simply by the huge computational task involved in considering and deciding on all possible moves at every stage of the game.

Hence the idea was to streamline the algorithm so that the best possible move is selected as quickly as possible without compromising on its quality. Another major consideration was to minimise the memory usage at every step keeping in mind the limited resources available.

***The major change to be incorporated was the redesigned points system, which entailed winning on points,even at the cost of losing the game per se.

DETAILS:

Data Structure Involved-

The configuration of coins is represented as a Lower Diagonal Matrix.

```
1
2 3
4 5 6
- - - -
-
16 ...  21
```

This approach has two advantages over storage as an equilateral triangle.1) space usage is economised,and 2)the valid moves are simple,viz. vertical,horizontal and diagonal.

Depth First Search-

The current configuration acts as the root of the game tree, with the available options being the 1st level nodes forming their own subtrees.At each stage the subtrees branch out according to all the possible moves that can be played,until the last coin is picked(this forms the leaf node).Thus an exhaustive search is made of all possible outcomes starting from the current configuration.

This recursive approach had the constarint of limited memory use being feasible at each step.So instead of passing copies of the entire matrix to the subsequent levels(which entailed formation of numerous matrices),the same matrix is passed after making the requisite changes.These changes are stored in an auxiliary array,with these being reverted after its corresponding subtree has returned.This also speeds up the search immensely.

*** **Improvement**: In first round, the points system was favourable to winning the game only. So, the first best winning move was selected. In second and third round, the points system determined the move. This time the search is undertaken according to the priority of coins per move, which is decided by the given points system.The aim is to get the optimum points picking up the optimum number of coins.Priority is 1,3,5,2,4 coins per move. 6 coins move is avoided altogether.

Decision Making-

Minimax approach- The entire is built on the precept of both players are equally intelligent,with the first player having the initiative to select the winning move,and the second player relying on the former slipping up for the latter to win.Thus at each stage, both players make the best available option.

Cut-offs Applied- Given a configuration,the probe stops once a winning move is found and its sibling trees are not analysed.This economises the time required for the decision to be made. This remains the same for both players with a sure-win situation for one being a sure-loss situation for the other.

When faced with a sure-loss situation,a single coin is picked.This is done in order to make the opposing algo think the hardest giving us a chance of timing it out.This also stretches the game giving the opposition maximum chance of slipping up.

Special Optimizations-

I)

We observed that a large number of similar configurations were analysed repeatedly when the number of coins was in the sub-10 range. This unnecessarily slowed down the execution of the algorithm where recursion was concerned. So, taking a cue from Tabular Dynamic Programming techniques, we decided to store the winning configurations of the game up to 8 coins.

But this implies storage of millions of possible matrices, with the extra time taken to search for a specific configuration and comparisons. So, this was not feasible unless some better way of matrix storage was applied.

The total number of possible configuration is (2^21)-1, which is well within the "long int" range. So, if a matrix can be encoded and decoded into an integer, the storage space can be drastically reduced. The matrix is first encoded into its binary equivalent, which is further converted into its integral form.

Thus a program was made which found all possible winning configurations when the number of coins was limited to <=8. These configurations are later stored in the main program. Hence, whenever the number of coins falls in the sub-nine range, we encode the matrix to its corresponding integral form and use Binary Search to find if it is a winning configuration. Thus the probe terminates immediately after no. of coins falls to less than 9. This greatly speeds up the algorithm.

II)

*** When the number of coins is large, there is still the problem of extended
recursion.The solution is picking 1 coin until 16 coins limit is reached,thus
giving us maximum points: coin ratio and leaving the maximum number of coins
for the opposition to slip up.

Compromises-

In order to limit the depth of the game tree, when playing as player 2, we pick a singlet coin until the number of coins is reduced to 16.

Scope of improvement-

We select the first winning move without regard to the points it will fetch us.This can easily be incorporated in the code by storing the win/loss points ratio alongwith the winning combinations.This could then be made use of to select the winning move with the most favourable points ratio.Since this would have involved more storage space and would have bloated the code, it was avoided for a game of this magnitude.

The same strategy could also be implemented in a sure-loss situation,thus giving us the chance to lose in the best possible manner.

**** **Important**: The strategy of deciding best moves on the basis of best 'net points' was tried out and discarded on account of the time limit being exceeded.

Summary-

In assignments 1 & 2, the thrust was on winning the game, since the points system demanded it. This was implemented successfully by the algorithm. This algorithm was first in assignment 1. But in assn.3, the points system required opting for the greatest points move even at the expense of winning the match. Hence a change was made in the intervening rounds of assn 3, making the algo more greedy in nature. But the strain of searching the entire decision tree looking for the optimal move (one

which guaranteed a win as well as greatest points) proved too much for the algo.

Hence,the approach for assns.4 & 5 began with devising an improved searching algo which minimised the repeated spanning of the decision tree, moved on to devising an efficient memoization technique and using the memoized results to improve performance on the fly.

Strategy for assignment 4 & 5

General strategy for assignments 4 and 5 was same, and will be described below. However, assignment 5 incorporated dynamic improvement and memoization, unlike assignment 4. Both these features are explained below.

Changes in the problem domain

- A Triangular configuration is replaced by a 6-row Square Matrix. This didn't involve many changes in our previous encoding, with the Lower Diagonal Matrix employed before being upgraded to a Square Matrix.

- The points system involves the number of coins picked.

Need for a New Approach

- Recursive calls to NextMove (amounting to a few million) while the current move is being considered are the bottleneck in the decision making process. This is due to the jumps involved everytime as new function is called.

- To open a sub-branch of the decision tree for all possible moves, even with cutoffs, is a wastage of time. The idea here is to replace recursive calls by iterative calculations at each call.

New Algorithm

We make a break from the standard recursive approach allied with minimax approach with alpha-beta pruning. A totally new algorithm is implemented which optimises the decision making process at each call to NextMove, taking into account the problem domain.

Conventions Used

NextMove:- Checks the matrix configuration, and returns whether it is winning or not? In case it is, it determines whether it can combine parent's move with the winning move at its level, and thus help the parent to win. This function calls itself recursively. Two different types of calls are made: for neighbouring coins, and for non-neighbouring coins. This is because non-neighbouring coins can never be combined with parent's move and this can be used to speed up the function.

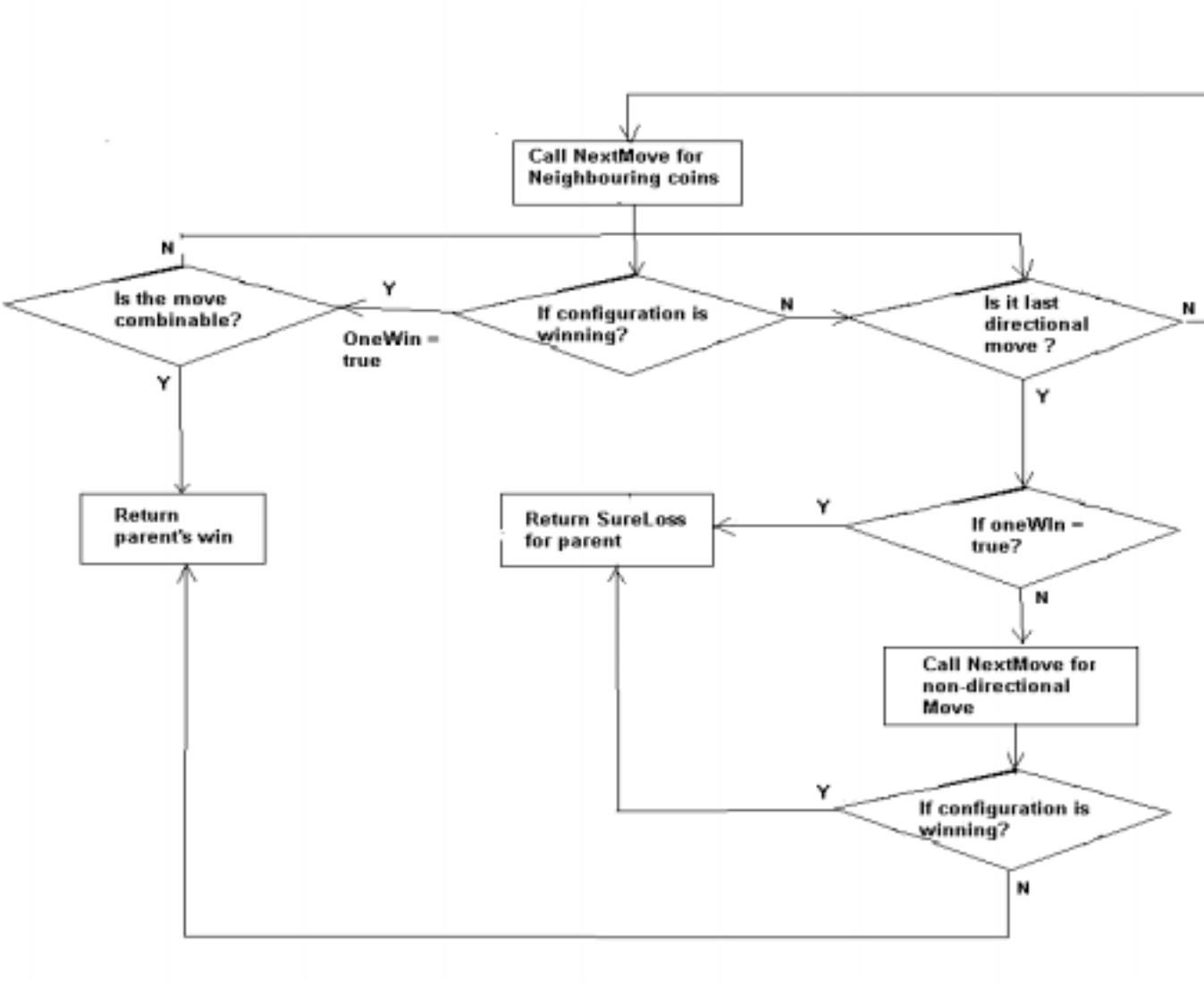
One Win:- This variable stores whether atleast one winning move is present in the current configuration by picking up neighbouring coins.

Neighbouring coins:-

```
1  2  3  4...
7  8  9  10 ...
13 14 15 16 ...
```

In this configuration, the neighbouring coins of 8 are : 9,13,14,15. It doesn't include 2,3,4,7 etc. because in the ordered progression in which each coin is considered, the combinations of these coins have already been analysed.The move involving neighbouring coins alongwith the current coin is called a **Directional Move .**
Moves involving non-neighbouring coins are called **Non-directional moves.**

ALGORITHM CONTROL FLOWCHART



Explanation:-

In this algorithm, we use the domain knowledge of the game and an advanced use of the minimax approach to make decisions at each level. At each stage, the possible moves of the opponent are considered and the decision is made with a view to preempt the possible winning situation for the opponent. Thus the decision at every stage involves recursively two levels of the decision tree. It involves implementing pruning techniques to arrive at the optimal solution at the earliest possible stage for all possible situations. The aim is to render redundant as many moves as possible thus minimising the breadth of the decision tree spanned. This approach improves the actual performance by a factor.

To be more specific, at each recursion of NextMove, **only one coin is picked up,** and the control passes on to subsequent stages. The children, ie, the opponent stores its bestmove, and passes it up to the

parent. The parent analyses the move, and if the parent can combine the coin it picked up with the opponent's BestMove, then effectively the parent has found out a winning move for itself. This process is applied at each level of recursion tree. Thus, at each call to NextMove, only moves involving a single coins are analysed recursively. All other are analysed by this procedure.

As an example, for configuration having only 4 coins: 1 2 3 4 0 0 ..
0 0 ...
...

1. Player 1 picks up coin 1.
2. Player 2 picks up coin 2.
3. Player 1 picks up coin 3.
4. Player 2 picks up coin 4, and loses, and returns to parent (P 1).
5. Since P 1 is already winning, it passes on its winning move (coin 3) to its parent P2.
6. Player 2 was earlier losing, so now it attempts to combine its move (coin 2) with its child's move (coin 3). It is combinable, so it becomes the winning player with its BestMove being 2,3, and returns to Parent P1.
7. Now, P1 is losing. So now it attempts to combine its move (coin 1) with its child's move (coin 2,3). It is combinable, so it becomes the winning player with its BestMove being 1,2,3.

So, finally P1 wins with move 1,2,3 even though only single coin moves were taken up in each recursion.

Advantage:- Earlier implementations would have considered the same problem in a different manner. At level 1, the player 1 will consider all the possible moves like 1; 2; 3; 4; 1,2; 2,3; 3,4; 1,2,3; and so on. This very same process will be involved at each level of recursion. Thus we are making the no. of moves at each level from **exponential** to **linear** order. The no. of levels, however, still remains exponential by the very defintion of the problem. But, the drastic improvement in the decision making reduces the problem magnitude from super-exponential to exponential order, which can be handled in given time.

Besides, as only one-coin moves are considered, the redundancy in matrix analysis is eliminated.

Dynamic Memoization and Log File Storage

An ingenious algorithm was devised for storage of matrices, which was new to this assignment. Each matrix was encoded in 1 **bit** only, with **O(1) storage and retrieval**.

The 45 second period with more than 18 coins present in the matrix is utilised for generating less than 8-coin matrices, analysing for win-lose, and storing in log files. This ensures maximal utilisation of the total time available.

A **long** in C is 4 byte (32 bits). We store the results of 31 matrix configurations in an integer. All matrices of different configurations are mapped to a **contiguous set of integers** in the following manner:

For example, the number of matrices with 5 coins possible are 36 C 5.

Convert the 2-D Matrix into a linear 36-bit array. Treat this array as a binary 36 bit integer. Now each matrix with 5 coins is to be mapped to a particular number with the first matrix (0 0 0 ......0 0 0 1 1 1 1 1) being mapped to 1. Observe that the arrays are ordered according to increasing value of the decimal corresponding to the binary number. Now all possible matrices with 5 coins are generated, and mapped to contiguous integers using combinatorial arithmetic. Similarly for matrices with other no. of coins, and the results are stored in different log files.

Storage and Retrieval:- After 31 matrices have been generated sequentially, and the Win-Lose results analysed, the binary string of 31 ones and zeros is converted to its decimal equivalent, and written into the log file corresponding to that no. of coins (e.g. cs20401.log5 for above). The information is written into the log file at the position corresponding to the 32 bit **long integer** generated. This ensures error-free, space-saving and highly efficient (order(1), since no searching at all is required)storage and retrieval.

For retrieval, the matrix (36 ones and zeros) to be retrieved is converted to its decimal equivalent. The integer at that position is retrieved.

Suppose the decimal equivalent of the matrix is N. Now N/31 gives the position of the integer containing the results of a group of 31 matrices, encoded as a **long integer** in the log file. To retrieve the required result from within this set, the bit at ((N-1)%31 +1) position is checked for one or zero. This allows all searching to be within O(1) time. Almost all our log files are of very small size. So, this highly optimised storage scheme ensures that program can run successfully on a system with very low resources.

Optimization of Resources:-

- Algorithm for recursive search is such that it greatly reduces NextMove function calls, and tries to decide a move on the basis of moves decided by its children. For instance, a winning move for a child node, if combinable with itself is viewed as a winning move for the current node. The stack saving procedures and jumps involved in a function call are thus minimised and replaced by simpler computations, leading to lesser strain on resources.

- The result of a particular matrix configuration is stored in effectively **1 bit**.The number of 6 coin matrices is approx. 19 lacs. This grows to around 83 lacs for 7 coin matrices, and 30 millions for 8 coin matrices. Storing the results as 'ints' would have required 4 bytes(32 bits) for every matrix. The space requirement of the program grows rapidly with increasing coins. The approach here is to store in 1 **int** the results of 31 matrices. The encoding and decoding of results info has already been explained above. So the technique results in **31 times** improvement in the storage space.

- The retieval of info stored in files takes place as in arrays, with the file pointer moving directly to the location where the result is stored. Thus, the retrieval takes place in O(1) time.

Greedy Choice Property

For the range where our algorithm is not able to analyse the move (no. of coins >=18), or when we are given a losing configuration, our algorithm analyses the configuration, and picks the move with the maximum no. of points. This ensures we don't lose with less points.

Runtime Analysis

- In the period where there are more than 18 coins in the game, it picks the best option on points but devotes the initial 40 second period to generating and analysing smaller matrices, and memoizing the results. Hence the time taken would be greater than 40 seconds.

- Below 18 coins, the actual algorithm works to analyse the available matrix and come up with the winning move.

- Initially, with 15-18 coins left, the time taken is maximum 40 seconds, and the minimum time with the most favourable configuration might be as low as 10 seconds. This time decreases with more matches being played. This is due to the memoization during the initial matches. The maximum limit reduces to around 30 seconds then.

- With 8-14 coins, the maximum time sans memoization is around 30 seconds. The minimum time is almost negligible for favourable configurations.

- For less than 8 coins, the results are read from the files. This memoization can be raised further to 9 or 10 coins easily, but to calculate all possible combinations and their results, large number of matches required. Uploading previously calculated log files would result in vastly improved performance, but detract from actual dynamic improvement, when the log files are created dynamically, on the fly.

Dynamic Improvement-

The improvement in performance of the algorithm is amply demonstrated by the fact that in the second round, it defeats teams to which it had lost to in the first round. This reflects clearly in the difference between the first and the second round scores. For example, this algorithm lost to groups cs20403 and cs20404 in the first round, but defeated them convincingly in the second and third rounds.
The minor decrease in the 3rd round score may be ascribed to improved performance by other teams leading to wins by lesser margins.

Further Improvements

- Currently, information from memoized files involves numerous calls to **fread** and **fwrite** functions. Compared to reading from the RAM, accessing information from files involves much greater time. If, at the start of each game, all the memoised files are loaded onto the RAM (in arrays), the improvement in running time is drastic. Moreover, since the size of log files has been minimised, the load on the system resources is minimal, thus

ensuring improved performance.

- Without any memoization (as in assignment 4), the the improved recursion technique starts analyzing moves from 16 coins onwards. Memoizing results of matrices upto 8 coins should have increased this statistic to 22-24 coins (something which has not been achieved by any group). However, this was actually limited to 17-18 coins because of the speed difference in reading from hard-disk and reading from RAM. If the above mentioned improvement is implemented, the performance will improve drastically.

INDIVIDUAL CONTRIBUTION OF TEAM MEMBERS:

All the team members contributed in a unified manner and were equally actively involved in every stage of development of this algorithm. Whenever one put forward an idea, the others analysed, debated, developed, implemented and tested the resulting algorithm. So equal credit goes to all the team members.

THE TEAM (CS20401):

Akshat Kumar : 01010101
Ashish Awasthi : 01010104
Mohit Tiwari : 01010114