Data Systems: Assignment-1

P Shiridi Kumar (2021121005) Shikhar Saxena (2021121010)

Assumptions made (if any)

- One based indexing for row and column positions.
- Separate index table for each game.
- Initial position is fixed and it's already simulated.
- Since we are storing fixed rows, we are using the unspanned memory allocation strategy.

Page Layout for a particular game

Naive way of storing the game

In the naive way, we store the snapshot of the board after each step. Here, a step can have multiple updates on the boards (i.e., one step can contain many board positions to be updated). But, only the final configuration, after the step is done, is stored.

Our approach is described as follows:

Actual detailed page design with some examples on 8x8 board.

Each step in a game consists of bunch of updates on the positions on the board. Each of these updates can be stored as a fixed length record.

The record attributes are as follows:

`update_tuple(step_number, row_position, column_position, updated_value, previously_empty)`

Attribute	Description	Domain of Values
`step_number`	The associated step/move in the game (for this update)	Positive Integer Values
`row_postion`	The row position in the board where this update applies	Integer with `min value = 1` and `max value = Number of rows`
`column_postion`	The column position in the board where this update applies	Integer with `min value = 1` and `max value = Number of columns`
`updated_value`	The final value at the corresponding update position. Semantically, takes `BLACK` or `WHITE` value	Storing `WHITE` as 1 and `BLACK` as 2

Attribute	Description	Domain of Values
`previously_empty`	Boolean value representing whether the corresponding position was empty before the update or not	Storing 1 for `NOT EMPTY` and 0 for `EMPTY`

Note: We store `previously_empty` for backward replay which will be discussed further, in more detail. Intuitively, if `previously_empty` is 1 means we updated the position from black to white or vice versa and if it is 0 means we added a new coin (black or white) to that position.

For each step, thus we can save a series of such fixed length rows in the data blocks.

Also note, we do not need to store separators for consecutive steps because each record starts with the `step_number`. Using this we can determine if all records of a step have been traversed or not.

Storing a game

For each move in a game, there would be multiple updated positions in the table (i.e, blacks could turn to white and vice versa). So at each move/step **we are storing the metadata of only the updated positions in the board whose representation is discussed in the above table** .

For example: A black coin is added at $(r, c) = (5, 5)$ in the 7th move of the game and this update can be represented as $(7, 5, 5, 2, 0)$ as a record in the data block . The above insertion of a coin in the board could update some values in the board (i.e, turn some of the board positions from white to black and vice versa) and let's suppose that the coin at $(4, 5)$ has been changed from white coin to black coin. Thus, this can be represented as $(7, 4, 5, 2, 1)$ where 1 in the last index indicates that its a update (from non-empty position) operation.

Following the above scheme we store only the updated values instead of storing the entire board (for each step).

Storing multiple games

As discussed above, we can separate each game with an appropriate delimiter in the page design. This will therefore constitute our page design for storing multiple games.

Indexing

Since our updates are ordered on the non-key attribute (`step_number`); we create an index for each move.

Here the index will contain the `step_number` and the `record_pointer = block_pointer + offset` which will point to the first update record corresponding to this `step_number`.

This indexing is on one game. We assume that each game will have a separate index.

(Additionally we can also have an index table which will store the `game_number` and pointer to the corresponding game index table).

How we simulate the game

In order to simulate the game we start from the first block and first record pointer and sequentially update the values accordingly by reading the updates present in the records.

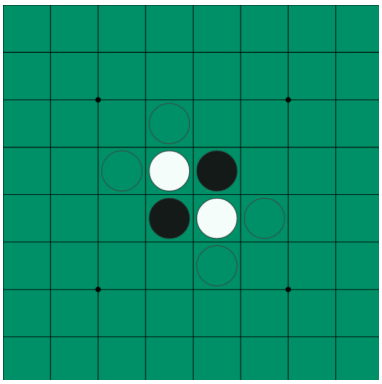

Figure 1: Initial Board Position

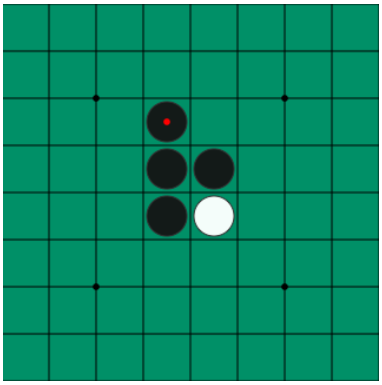

Figure 2: Black Coin inserted at $(3, 4)$

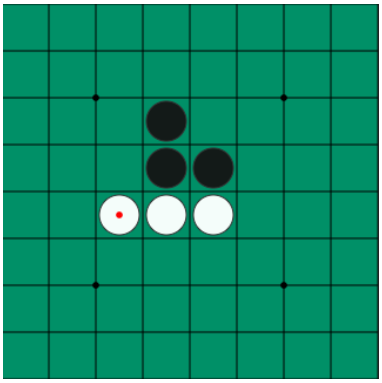

Figure 3: White Coin inserted at $(5, 3)$

Move Number	Row	Column	Updated Value	Previously Empty
1	3	4	2	0

Move Number	Row	Column	Updated Value	Previously Empty
1	4	4	2	1
2	5	3	1	1
2	5	4	1	0

The initial configuration is shown in Figure 1. The first update value in the move consists of $(1, 3, 4, 2, 0)$ as shown in the above table , which indicates black coin(2) has been inserted(since last index is 0) at $(3, 4)$ and then followed by a update in the next tuple which is $(1, 4, 4, 2, 1)$ which indicates the coin at $(4, 4)$ is updated to black and the resulting image is shown in Figure 2. And similarily we could follow the same manner for updating the positions in the next move and eventually we could arrive to the board state which is represented in Figure 3.

How we will get to a random move

In order to get to a random board state for an arbirtary move no k , we need to sequentially traversefrom the first moves since all the updates are stored in a sequential manner. **And in order to overcome this overhead we are extending our current page design By storing the snapshots of the block states at some time steps which would be discussed in detail as follows:**

- Instead of not at all storing the table states , We now store the table state for some fixed number of blocks (For a board of 1000x1000 we could store the board states for every 1000 moves and for a board of size n*n we could store the board state for every n moves)
- We will store the board state in a naive way of n*n matrix where an entry of 0 represents empty space and 1 represents white and 2 represents black.
- How to store board state in block?
  - Now for every index entry in the index table we could get the block ptr for every nth move and we can store a pointer to the board state at the start/footer of the block and if the index entry is not divisible by n the pointer in the start/footer would be null.
- How to get to the board state for a move k ?
  - In order to get to a move k we esentially need to find the nearest less than or equal to move number which is divisible by n
  - Which could be computed by $\lfloor \frac{k}{n} \rfloor \times n$
  - Eg : Lets suppose the board size to be 120x120 and now we store the board state in the blocks where the every 120th move is started or every 119th move is completed. Now if we need to get to a random move we need to find the board state for a move 1227 then we can find the nearest less than or equal to move which is divisible by 120 and that turns out to be 1200 now since we no that there would be a block pointer in the block where the starting record of the move 1200 is present which points to the board state which contains all the updates till that block start). Now since we know the nearest row which is 1200 in our case we could index on that and get to the start of that corresponding block and start reading the updates from there on .
  - Now since we are esentially reading only updates of 27 moves instead of reading from the beggining which is a huge change.

How we will replay in reverse

Since we are having all the updates done in each time step we could ideally scan all the updates till the end to get to the final board position. And now since we reached the final board position we can just read

the updates from the last block and last record and just revoke the updates (i.e, if the tupple is (5,2,2,1,0) then this implies there is a white coin inserted at (2,2) so we revoke it by removing the white coin and if the record tuple is (5,2,2,2,1) this implies that a coin is turned from white to black (since the last index is 1 which implies that it is a update operation and not insertion of coin) we now just revoke the update by turning it to white) we continues the process till we reach the desired timestep/move.

To store one completed game, how much storage is needed.

Given the board size is 1000x1000 and the maximum number of moves is 10^6.

In the Worst Case

For a move in the worst case, we put a coin and other coins flip in all directions.

Their are four lines passing from each point (except the border and edge points). So, in the worst case about 4×10^3 updates will take place (where 10^3 is row dimension).

Therefore, the total storage needed for a move $= 5 \times 4 \times 10^3 = 2 \times 10^4$.

So, for 10^6 moves i.e., the storage needed for one completed game

$$= 10^6 \times 2 \times 10^4 = 2 \times 10^{10} \text{ units}$$

Where each unit is equivalent to appopriate memory needed to store an integer.

Whereas for the naive approach the memory needed will be $= 10^6 \times 10^3 \times 10^3 = 10^{12}$ units.

In the Average Case

Although the worst case for our approach is quite high **ideally that is going to happen in very rare situations** The reason is the number of updates for a move are assumed to be 4000 in the above calculation which is not true in general its quite rare to happen that a coin is placed in the middle and all the rows and columns and diagnols around it are updates . So on an average lets assume the number of updates for a move to be half of the number of columns or rows which is equal to 500 in above case . then the space required would be

$$= 10^6 \times 5 \times 500 = 25 \times 10^8 \text{ units}$$

and the avrage case for a naive approach would be still the same since we need to store the board state for all the moves and is equal to

$$= 10^6 \times 10^3 \times 10^3 = 10^{12} \text{ units}$$

As we can see that approach is far superior than the naive way .

Why you think your page design is the best?

Firstly, our page design is better than the naive approach because we save up on memory by not storing the entire board state for every move.

Since, the approach we initially proposed requires sequential scanning for accessing a random move; we also proposed an extension to the original approach by storing board states snapshots after a fixed number of moves which is proportional to n where (n is the dimensions of the board).

Since, we are optimizing the space and the time (for a random move simulation); we think our approach is the best approach for this problem statement.