Page Design for storing Played Games for Reversi/Othello with efficient navigation queries

- Anmol Agarwal (2019101068), Sanchit Arora (2019101047)

Game rules: https://en.wikipedia.org/wiki/Reversi

Some rules & board invariants worth highlighting for the purpose of our design:

- **A1:** Different sources mention different criteria for ending of the game when a player cannot move. Some sources say that the game ends when either player cannot move. Others say that the game ends only when BOTH players are not able to move.
    - We assume: GAME ENDS when even one player does not have a move left.
- **A2:** The disks on the board make **ONE SINGLE connected component** (where valid moves are horizontal, vertical and diagonal). This is easy to see as a disk can be added only when at least one of the opponent's disks swaps its color.
- **A3:** In each move, exactly one disk is added to the board i.e. the number of disks on the board is the same as the number of moves done on the board so far.

Some terminology/notation:

- **N** = side of the board => N x N = number of cells on the board | N<=1e3
- For ease of reference, without loss of generality, we consider the initial configuration of the board played i.e. having 4 disks as the first four moves. Hence, MAX MOVES in our case: N x N (since we are considering the initial configuration of the game to consume 4 moves)
- **G**: game ID
- **TotMoves:** total number of moves played in the game.
- $\mathbf{M_i}$: ith move of the game where 1<=i<=TotMoves. (NOTE: M1, M2, M3 and M4 are fixed since they correspond to the initial configuration of the game)
- $\mathbf{S_i}$ = state of the board after Move "i"

Queries we are prioritizing:

There were many page designs in our minds which could have been optimized for different queries for the game. While presenting our below design, we consider the below to be our topmost priority:

- Q1: Size needed to store each game
- Q2: Time needed to move from state **"i"** of the game to state **"i+1"** ie replay the game
- Q3: Time needed to quickly switch among far away states of the game.
    - Eg: instead of starting from state 1, the user can directly ask to view the state of the game after 4500 moves. After that, he/she can ask to **directly** move to the state of the game after move 11005 without transitioning through all the states from **move 4500 to 11004**.

How a state change might be implemented in a OTHELLO GAME:

- LET US ASSUME ONLY for an instant (and NOT for the remaining part of this document) that we are able to load the entire game into memory into a matrix ARR where ARR[i][j] is:
    - 2 if cell is empty
    - 0 if cell is filled by BLACK DISK
    - 1 if cell is filled by WHITE DISK
- Let's say a move has been made which places a white disk at cell (P,Q) of the board. This move can evoke a series of cascading color swaps in all 6 directions:
    - Up
    - Down
    - Left
    - Right
    - Diagonal
    - Reverse Diagonal
- As a result, due to this move, we will have to make changes in the matrix.
- Since we have 6 directions to search for, in a naive implementation, we might have to take ATMAX "N" steps in each direction to swap the disk colors. So, the time complexity of making updates due to a move in a naive way is 6N operations approx which is **O(N)**

- Such an **O(N) update per move** may be too inefficient for queries of type Q3 where the USER jumps between states of the game. Eg: if board size N = 1000 and the user wants to jump from move 1 to move 1e5, then the number of operations will be ~ 1e5 x N ~ **1e8 operations** which is very large. To make answering such queries more efficient, we propose storing multiple snapshots of the game in the manner proposed later.

Page Designs:

While brainstorming, in our shortlisted page designs, we often found a tradeoff between storage space needed and time needed to update the state of the board when a move is made. As a result, in our pipeline, we try to incorporate the best of both worlds:

- **Page Design 1 (Storage based Page):** This is the design we follow to store a game. Once written, these pages are READ-ONLY in nature.
- **Page Design 2 (Update based page):** This page design is used to store the current snapshot of the game. Once the user has finished investigating the game, the space occupied by these pages can be relinquished. These pages are regularly updated as the user navigates through the game.
    - If the user is currently at state “i”, these pages are used to store state “i” of the game.
    - When the user moves to state “i+1” from state “i”, then updates are made within these pages themselves.
    -

Making navigation queries efficient by using SQUARE-root decomposition type logicProposal:

Let us assume that game “G” has “TotMoves” total moves. Then, we store the snapshot (checkpoint) of the board (using page design 1) for every Kth state where K = sqrt(TotMoves).

For eg:

Consider the initial empty board to be S_0.

After move 1, the state of board transitions from S_0 to S_1.

After move 2, the state of board transitions from S_1 to S_2.

.

.

.

After “TotMoves”, the state of the board transitions from $S_{TotMoves-1}$ to $S_{TotMoves}$.

Then we will be taking a snapshot after every K moves where K = sqrt(TotMoves).

Number of snapshots we will be taking will be: TotMoves/K = TotMoves/sqrt(TotMoves) = sqrt(TotMoves).

How does this help in answering queries efficiently ?

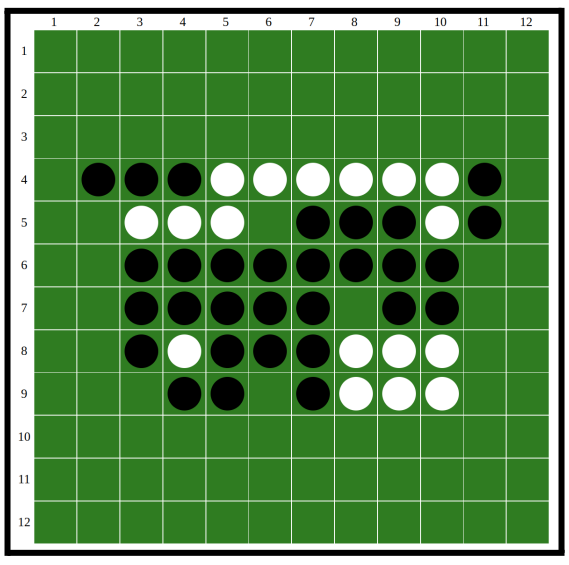
- Let us consider a game with TotMoves = 1e6.
- Let’s say that a user is currently viewing state P of the game. Next, the user suddenly wants to move to state “Q” of the game.
- **In case we hadn’t stored checkpoints**, we would have to move “diff = Q-P” moves ahead with O(N) operations per move.
    - As a result, we would be making approx O(diff x N) operations. In the worst case, diff can be nearly N^2 and hence, we might end up making **N^3 operations**.
- **Since we are storing checkpoints**, we can do the following to reach state “Q”:
    - Find the nearest previous checkpoint to state “Q” which has been stored. Let’s call this checkpoint “X”.
    - Then, we already know the configuration of the board after move “X” (since we stored it) and hence, we now need to make only “diff = Q-X” moves.
    - Here, since we store a checkpoint every sqrt(TotMoves), diff can be atmax sqrt(TotMoves).
    - In the worst case, diff can be nearly N^2 and hence, we might end up making (N x sqrt(TotMoves)) operations where TotMoves can be atmax N^2 and so, in worst case, we would end up making **N^2 operations** which is an improvement by a factor of “N” (ie an improvement by a factor of 1000 potentially for large board sized games).

Page Design 1 (Storage based Page)

We propose to use this page design for storing the checkpoints of the board state every sqrt(TotMoves) moves as described above. These pages will only be READ-ONLY in nature.

We will be covering this section with the help of an example:

Consider that we need to store the following state of the 12 x 12 board:



Feature 1: Truncating the board to store only the smallest rectangle which covers the entire connected component

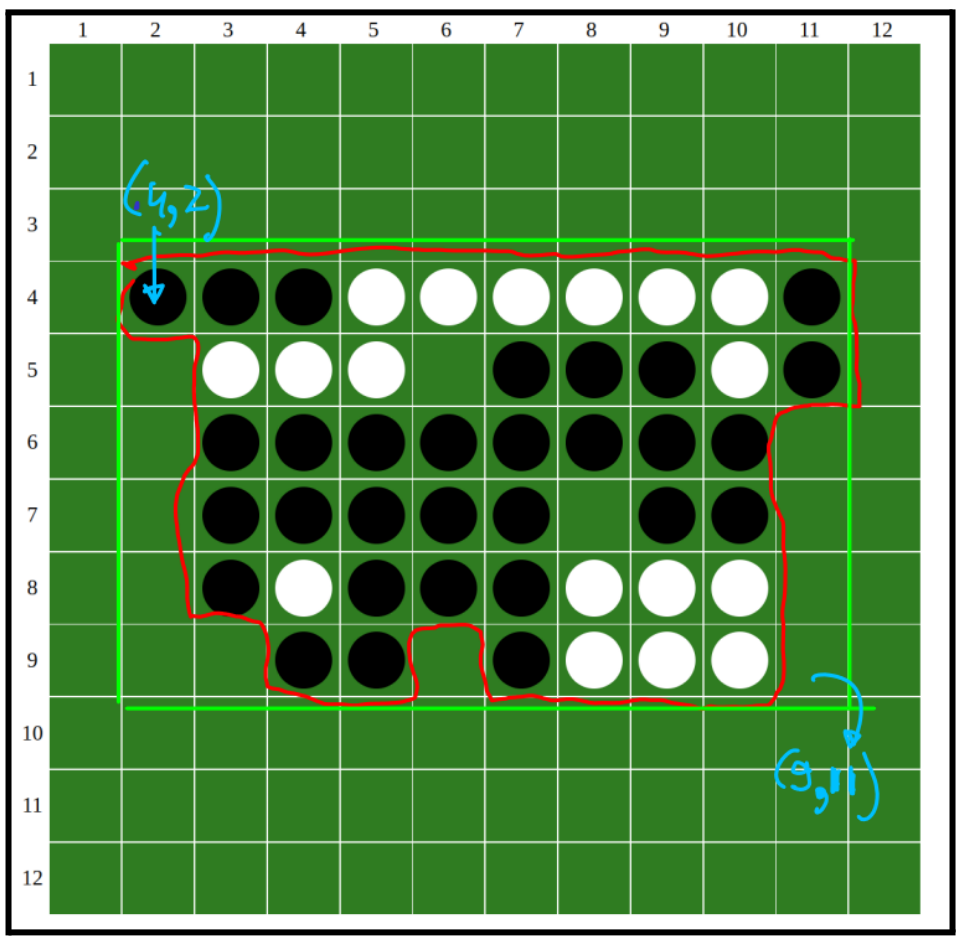
First we make the following claims:

Claim ID	Claim:	Reason
1	The disks on the board make ONE SINGLE connected component (where valid moves are horizontal, vertical and diagonal)	This is easy to see as a disk can be added only when at least one of the opponent's disks swaps its color.
2	The number of disks on the board is the same as the number of moves done on the board so far.	In each move, exactly one disk is added to the board

As a result, let's say that **M** moves have been done. This means that **M** disks have been added to the board. Let the current connected component of disks be "**C**". Let "**R**" be the smallest rectangle which covers the entire connected component. Then, we claim:

- **Width W of rectangle + Width H of rectangle <= M** (since in each move, we can either increase the height of rectangle enclosing connected component by 1 or we increase width of connected component by 1 or both height and width remain the same)

In the above example,



In the above figure, we see the connected component "C" is enclosed within the **RED** polygon and the smallest rectangle which covers the connected component "C" is enclosed in **GREEN**.

The top-left corner of the rectangle is at cell (4,2).

The Bottom right corner of the rectangle is at cell (9,11).

In order to represent this configuration,
we would be storing H x W cells (more details on the way these cells are stored mentioned later in the doc). Since **W+H<=M**, by AM-GM inequality, we can say that **H x W <= $M^2/4$**

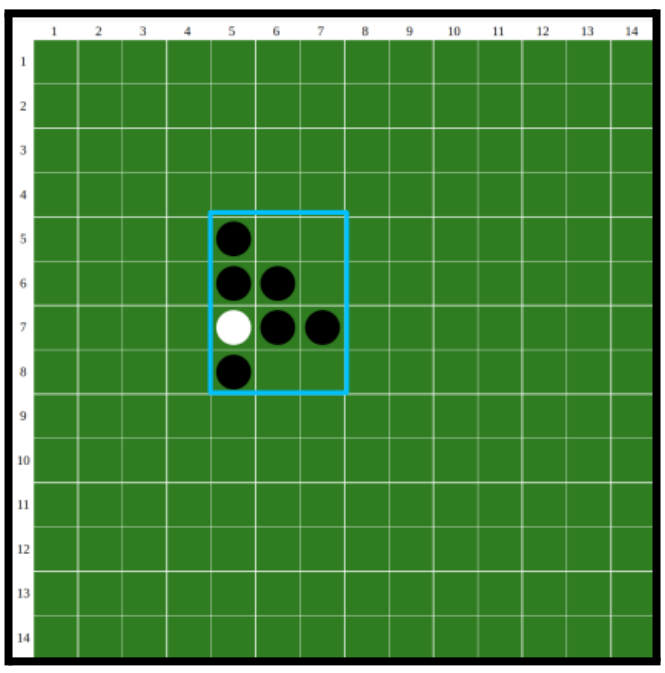
Proposal:
Instead of trying to capture the entire board, we can try to capture only the smallest enclosing rectangle. For storing a particular state of the game, we can **store the coordinates of the top-left and bottom-right corner of the rectangle** along with the state of the rectangle (which is empirically MUCH SMALLER as compared to the entire board in the INITIAL AND MIDDLE PART OF THE GAME)

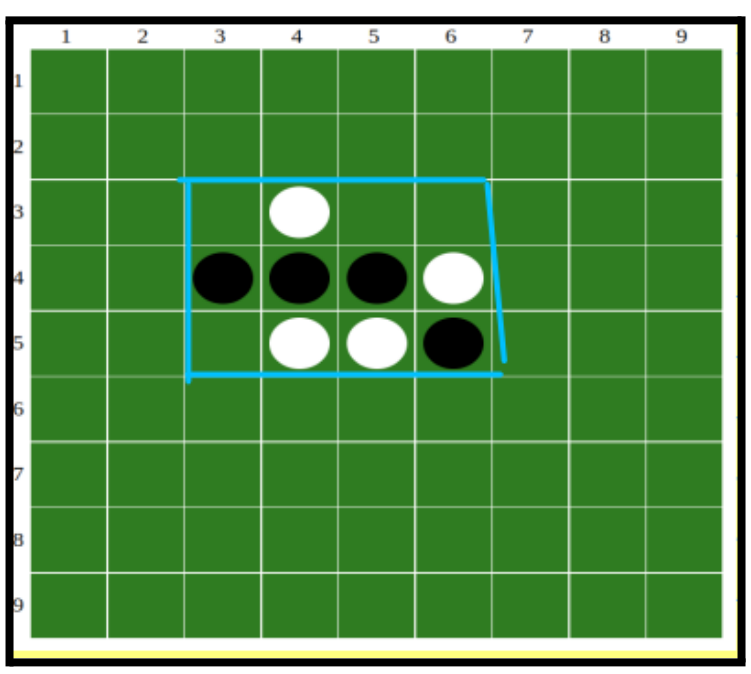
Why would this save a lot of space ?

For perspective, for a **1000 x 1000 board** with 1,000,000 cells,

Moves Made	**Maximum number of cells enclosed by the smallest rectangle enclosing the connected component**	**Fraction of total cells in the board**
1	1	1/1e6 = 0.000001
5	7	7/1e6 = 0.000007
15	57 (which is very less as compared to 1e6 ie total number of cells in the board)	57/1e6 = 0.000057
100	2500	2500/1e6 = 0.0025

Some more examples of the smallest enclosing rectangle:



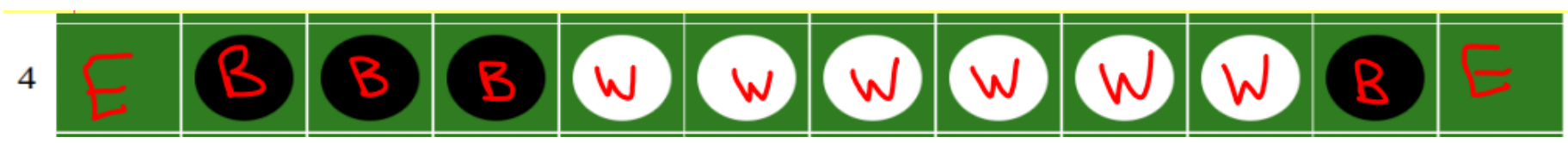


Feature 2: Encoding rows

Now, let us store the board row-wise ie calculate an encoding for each row.

Suggested encoding 1 for rows:

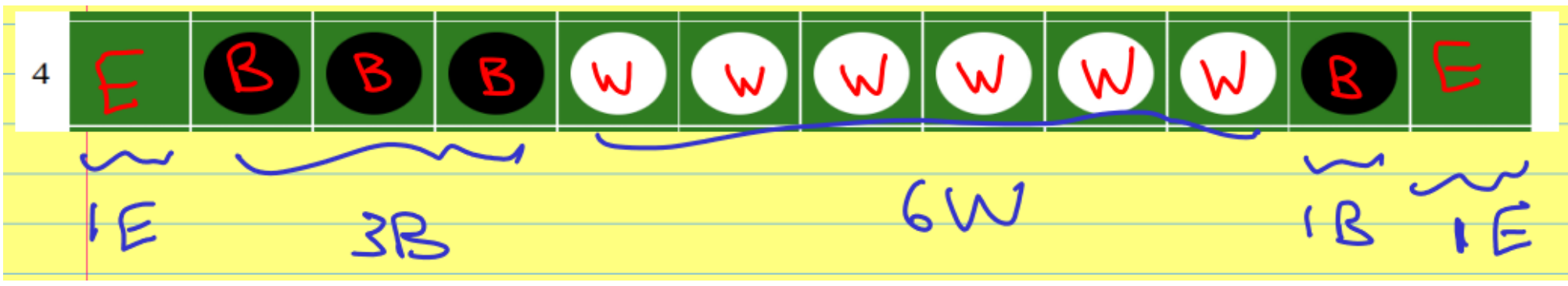
Let us say that we want to encode row 4 of the board.
Let B = black, W = white and E = empty
Way V1: The current configuration of row 4 in the 12 x 12 example is : EBBBWWWWWWBE which needs 12 characters to store



Better way V2: We observe that a lot of neighboring characters have the same value and hence, we can prevent some repetition by storing repeated values as a pair with entries

<number of times a value has been repeated, the repeated value>

For eg: compressed way of saving row 4 can be **1E3B6W1B1E** which has 10 characters only (we store 2 less characters).



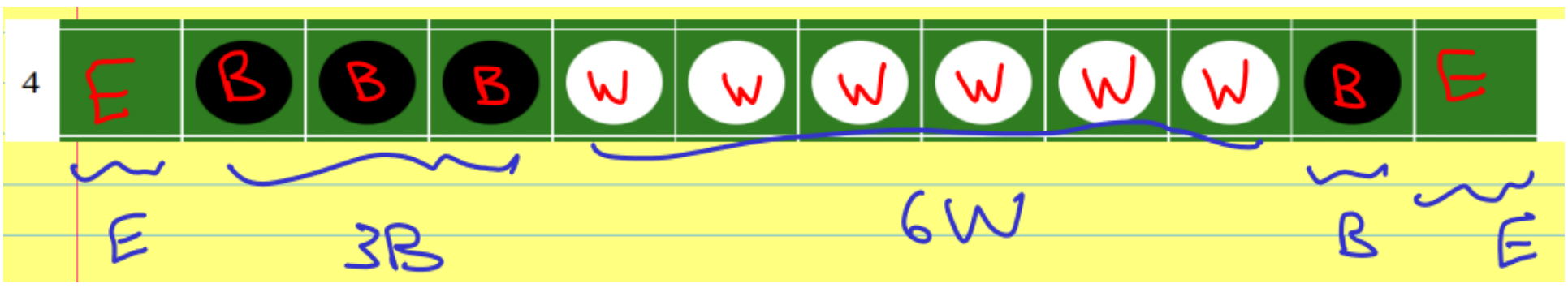
The amount of characters we will be saving would depend on the number of repetitions but given the nature of REVERSI, one expects a lot of neighboring squares to have the same values.

Improved way V3:

A small improvement would be to replace tuples of the form <1, VALUE V> from 1V to “V” only. This does not introduce any ambiguity in our notation. Eg: row 4 will be encoded as: **E3B6WBE** which has 7 characters only.

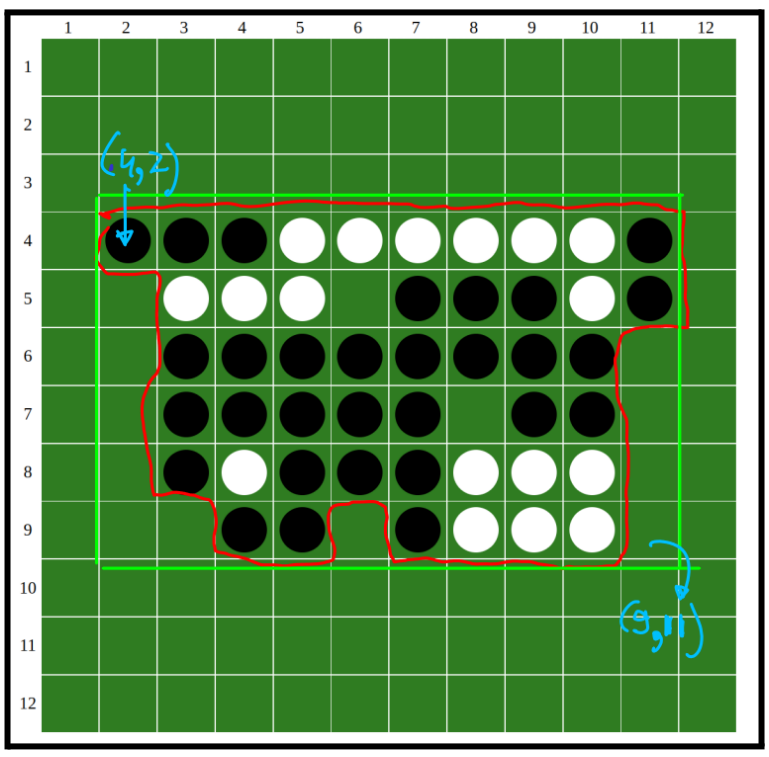
Whenever 2 alphabets are encountered consecutively, we can interpret the frequency of the second character to be “1”.

It is easy to see that way V3 will always occupy less than or equal space when compared to way V1.



Sample example of page:

State of the board



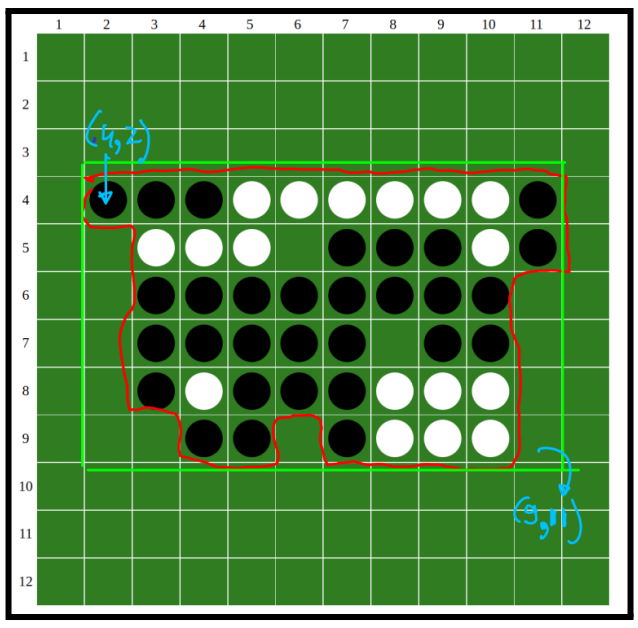
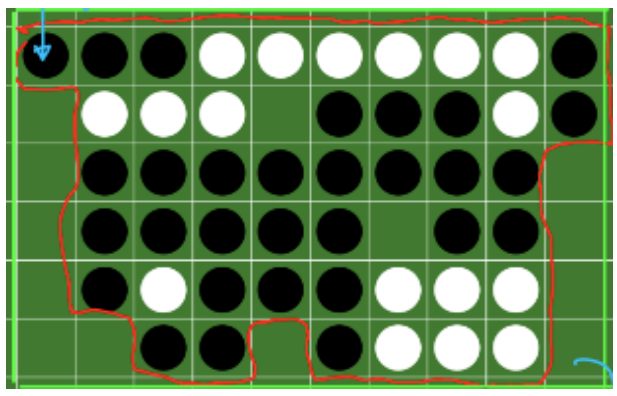
Encoding of the different rows:

Row 1	12E
Row 2	12E
Row 3	12E
Row 4	E3B6WBE
Row 5	2E3WE3BWBE
Row 6	2E8B2E
Row 7	2E5BE2B2E
Row 8	2EBW3B3W2E
Row 9	3E2BEB3W2E
Row 10	12E

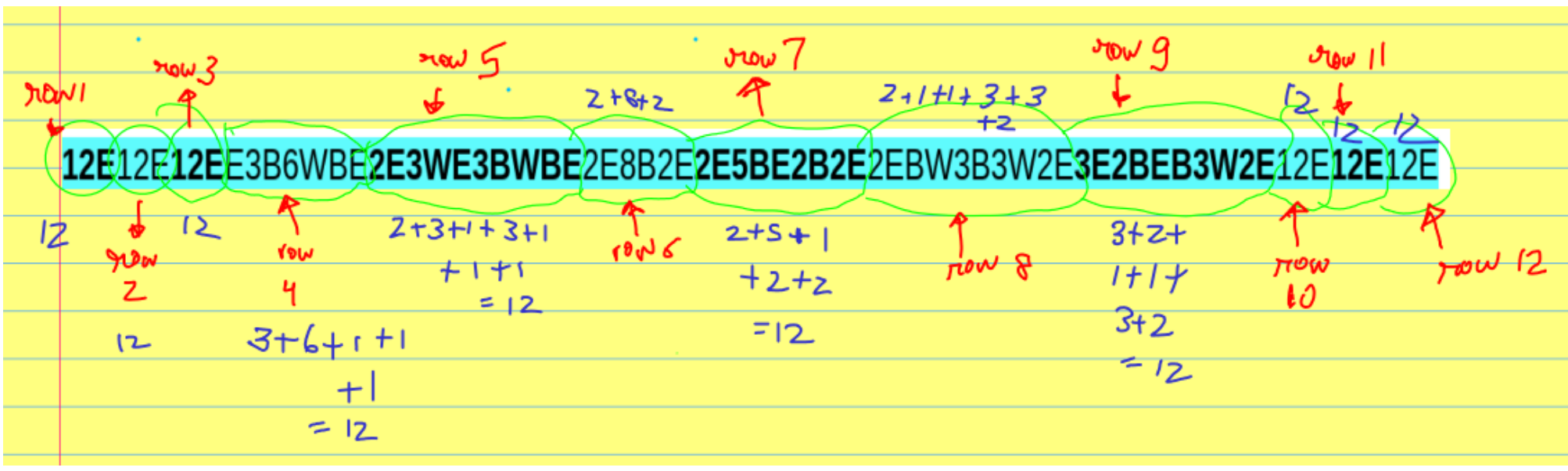
Row 11	12E
Row 12	12E

Now, **let STRING S (with @ as row based delimiter)** be:
<ENCODING OF row 1 >@<ENCODING OF row 2>...@<ENCODING OF row 12>
This string can be stored directly in pages (can be stored across contiguous pages).

We **do NOT actually need to store the delimiter**. The end of the ROW CAN EASILY BE INFERRED WITHOUT DELIMITER as we CAN KEEP TRACK OF THE number of cells read so far. The moment we have N cells in a row (N=12 cells for the above example), we can interpret a row change.

Without truncation	**With truncation**
	

Final string to be stored across pages for the above example (WITHOUT TRUNCATION):
12E12E**12E**E3B6WBE**2E3WE3BWBE**2E8B2E**2E5BE2B2E**2EBW3B3W2E**3E2BEB3W2E**12E**12E**12E



For ease of reading above, every alternate row's encoding has been bolded. Also, we can see that the sum of value frequencies within each row's encoding adds up to 12 which helps us detect when a new row starts.

Final string to be stored across pages for the above example (after TRUNCATION of the 12 x 12 board into a 6 x 10 rectangle):
3B6WB**E3WE3BWB**E8BE**E5BE2BE**EBW3B3WE**2E2BEB3WE**

NOTE: That this design can be inconvenient to visualize/make updates on. That is why we use it only for storage.

Page Design 2 (Update based Page)

It is not easy to make updates based on moves in the design used in page design 1. Hence, we switch to a different page design on which updates will be much easier.

- When a game is being played, its current state "S" will be stored on disk in form of page design 2.
- When a move is made, the changes are made in the cells in-place i.e. within the same pages.
- When all queries regarding the game has been fulfilled, the space allocated to it to store it's boards for updates is EMPTIED.

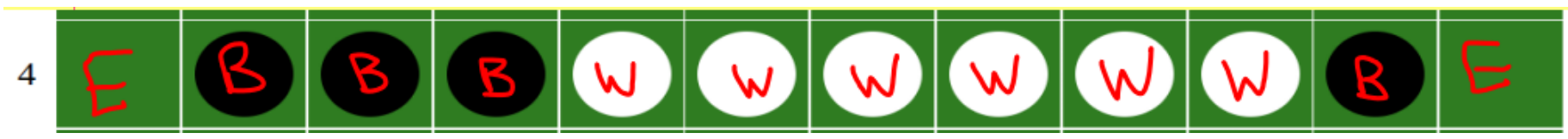
Let's say a game "G" has 100 moves and snapshots are already available for states after every 10 moves i.e. for (S_{10},S_{20} ,S_{30} ,S_{40} ,S_{50} ,S_{60} , S_{70}, S_{80}, S_{90}, S_{100}).

Let's say a user wants to visualize the game position after move 67 and then wants to move step by step till move 68. The following will happen:

- 60 is the nearest integer to the left of 67 for which snapshot was stored.
- A copy of the state of the game after move 60 (S_{60}) is loaded and converted to page design 2 (which is proposed below) and stored into disk (as page with page design 2).
- Moves 61-67 are executed with changes being made in the copy of S60 stored via page design 2.
- State of board after game 67 is displayed.
- State of the board is updated (in the copy stored as per page design 2) as per move 68.
- State of board after game 68 is displayed.
- Now, since the user has left, all the pages (of page design 2) corresponding to game "G" are removed i.e. their storage is freed.

An important point to note here is, that the storage of the state we will be updating will be negligible as compared to the storage of the whole system (millions of game, with each game storing state at different checkpoints). **This will be only one (temporary) state that will be updated again and again.**

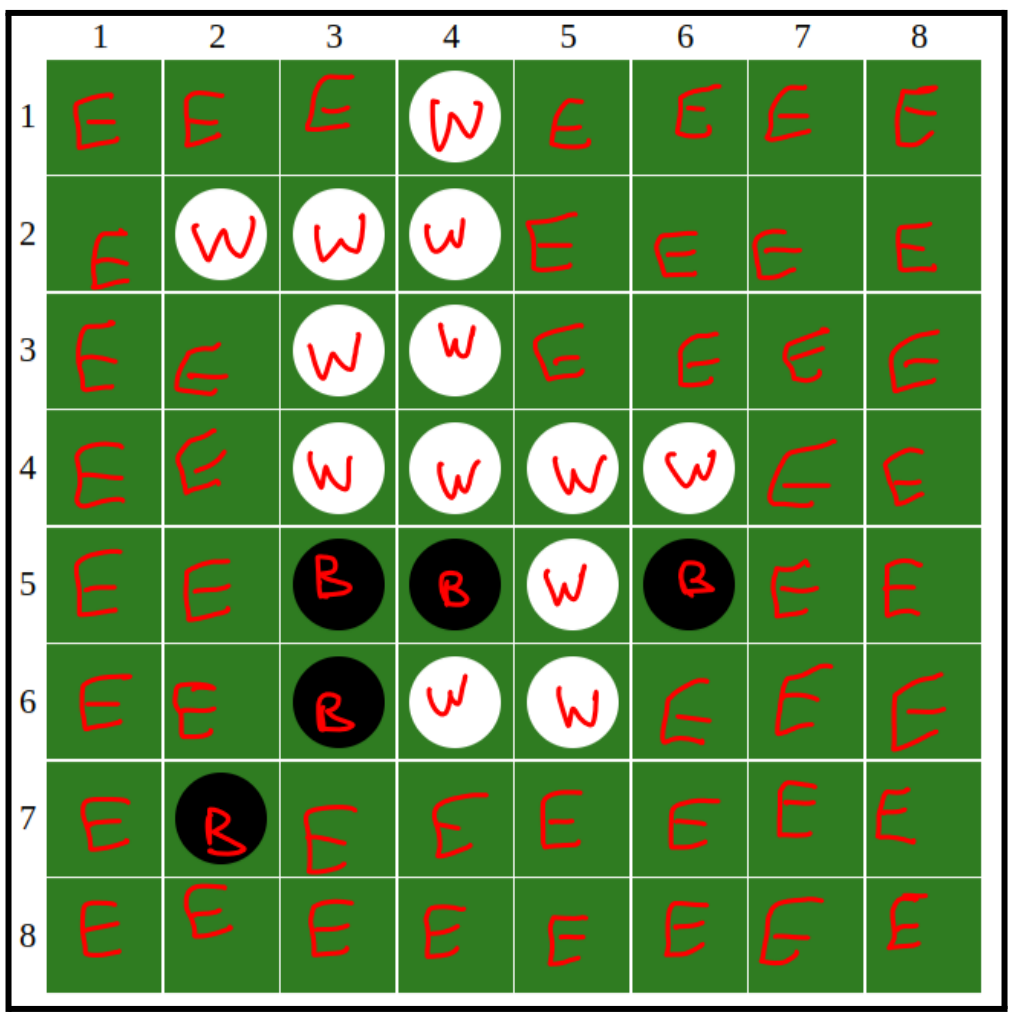
For the 12 x 12 example discussed earlier in the document,
if a row looks like:



We'll simply store EBBBWWWWWWBE, **without any optimized encoding** (as opposed to the case of storage-based page design) since optimized encoding makes updates complicated.

Now, since the whole board can't be stored in a single page, we need a way to break them. And, there are mainly two ways to do so:

Let us consider the following **8 x 8** example (and a page size of **9 bytes**) to demonstrate the design:



Proposed update-based page design 1:

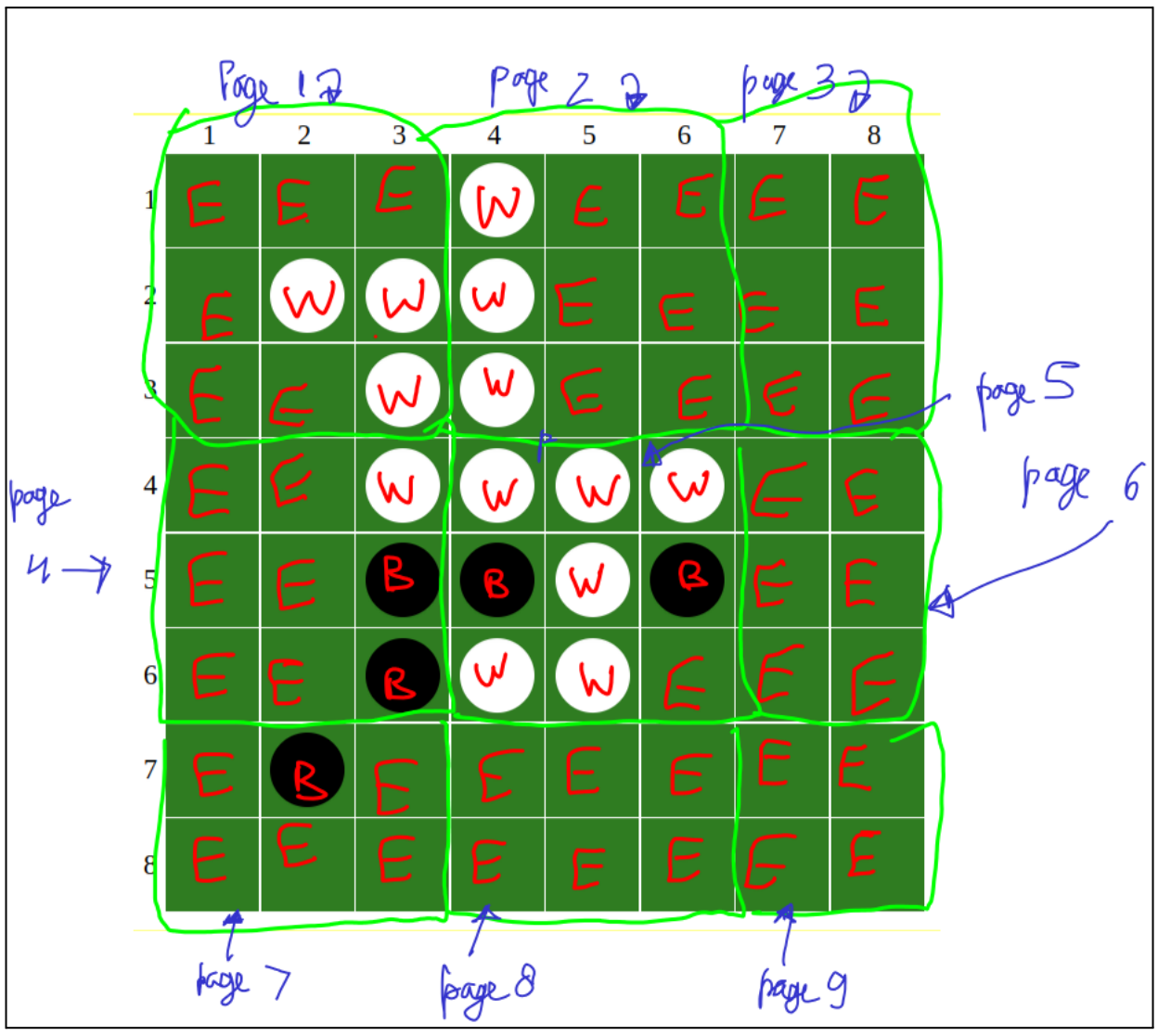
Flatten the 2D board into a 1D board and store a character "E","W" and "B" depending on the state of the cell. Each character occupies one byte. So, if a page can store "T" bytes, then ceil(N^2/T) pages will be needed.

String S which can be stored across pages is (alternate rows have been bolded for convenience):
EEEWEEEEEWWWEEEE**EEWWEEEE**EEWWWWEE**EEBBWBEE**EEBWWEEE**EBEEEEEE**EEEEEEEEE

In the above example, different colors represent the contents of different pages assuming a page size of 9 bytes.

Proposed update-based page design 2 (much improved):

Store a sub-grid in each page. So if, let's say, the page size allows us to store *p* elements, we'll store a grid of $\sqrt{p}$ X $\sqrt{p}$ in each page. Since we have assumed page size to be 9 bytes, p =3 for our example.



NOTE: Since all elements are fixed length (1 bytes), knowing what page and offset a particular cell is present in is straightforward.

We would choose proposed design 2 for our purpose as:

Design 1 of flattening the array	**Design 2 of breaking matrix into submatrices**
• Row based updates are **highly efficient** as due to the flattening, all elements in the same row are in the same page (or spread across neighboring pages if row size >> page size) resulting in **minimal page accesses**. • Column updates & diagonal updates are **highly inefficient**. 2 neighboring elements in the same column MIGHT BE in different pages, As a result, even updating 2 neighboring elements of the same column can take 2 page accesses. • Locality of reference based advantage present for row-based updates only.	• All updates row--based, column-based and diagonal based are moderately efficient (reasonable number of pages accesses) as most cells are in the same PAGE as their row neighbors, col neighbors and diagonal neighbors. • Locality of reference based advantage present for all 3 types of updates: row-based col based and diagonal based.

Another small (but effective) optimization using bits

Without much increase in the time or implementation complexity of the state update, we can decrease the storage space required. We notice that we are dealing with **only 3 unique characters**: B, W and E. We can encode 3 unique characters with just 2 bits.

So, instead of using an entire character (1 byte) for each state, we can represent them using 2 bits **(4 fold reduction in space).** See the below table for more details:

STATE	Earlier representation **(8 bits)**	Proposed bit representation **(2 bits)**
WHITE CELL	ascii of 'W': 01010111	01
BLACK CELL	ascii of 'B': 01000010	10
EMPTY CELL	ascii of 'E': 01000101	00

This data can simply be treated as bit stream, which can then be encoded into bytes for storage (explained in detail below).

Storing the moves of each game in a new relation

Let each game have a GAME ID which is **incremental** in nature. For resolving the state of the board between checkpoints (intermediate states), all moves of the game need to be stored. For this, we maintain a separate table with this page design as explained in “proposed approach”: **Since the player turns are alternative, we need not store the player ID of the player who made the move (**an even move is always by player 1 and odd move is always by player 2**).**

Proposed Approach

- Each move can be uniquely identified by which cell the disk was placed: a (row, column) pair.
- Instead of adding a separator between row and column (and, between moves), we’ll keep them fixed size. This will also help to optimize queries like what was the 1000th move.
- It’s tempting to store the number as a normal integer (or a string), it’ll take 4 bytes. So, a total of 8 bytes will be used per move.
- **But, we’ll store the sequence of moves as a sequence of bits.**
- Since the number (row or column) can go upto 1000, 10 bits will be enough to store a row or column number (since 2^{10} = 1024 > 1000).
- So, in total, storing the location of a cell (and hence, also one move) will require only **20 bits or 2.5 bytes.**

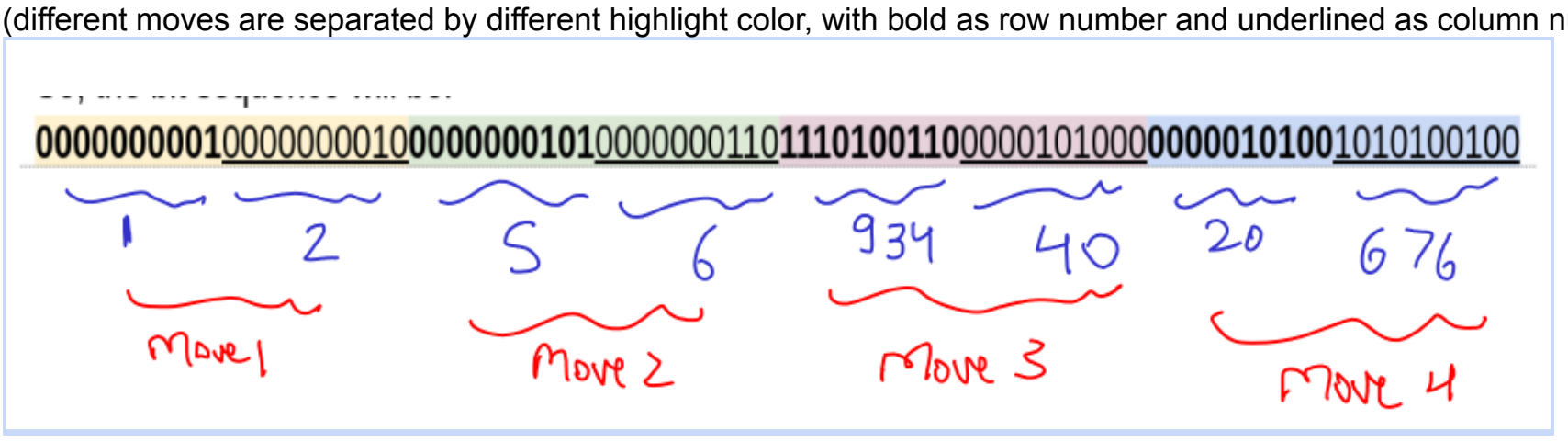
Example:

Consider an arbitrary (not necessarily valid) sequence of moves:
Move 1: Black disk placed at (1,2)
Move 2: White disk placed at (5,6)
Move 3: Black disk placed at (934,40)
Move 4: White disk placed at (20,676)

The 10 bit representation of the numbers are shown in the table.

Number	Binary Representation (10 bits fixed length)
1	0000000001
2	0000000010
5	0000000101
6	0000000110
934	1110100110
40	0000101000
20	0000010100
676	1010100100

So, the bit sequence will be:
00000000010000000010**0000000101**0000000110**1110100110**0000101000**0000010100**1010100100
(different moves are separated by different highlight color, with bold as row number and underlined as column number)



Now, to store this bit sequence, we can **group them into sizes of 8**, and store its character representation. So, the above bit sequence will be grouped as:
(00000000) (01000000) (00100000) (00010100) (00000110) (11101001) (10000010) (10000000) (01010010) (10100100)

When converted to integers, it’ll become: 0 64 32 20 6 233 130 128 82 164
Now, we can use these integers as ASCII value for the characters we’ll store (we don’t show them here as some of them are not printable).

As we can see, we store 10 characters (= 10 bytes) for storing 4 moves. While reading, we can simply decode the characters back to bit string and then find the required row and column number.

In the page for this relation, one can continually store the elements ordered by GAME ID as:
<Game Id 1>**:**<Unfixed length bit string of moves>**&**<Game Id 2>:<Unfixed length bit string of moves> ……. and so on

Since an element of such a relation CANNOT be modified after insertion, maintaining an ordered unfixed relation is easy.

Additional data may be stored per page to store information such as the list of game IDs whose moves are present within the page and so on.

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

I have given stepwise memory consumed by the different steps of the pipeline in respective sections. Recapping:
Let size of board : N x N
Total number of moves = TotMoves

Term 1: Memory consumed to store the moves
Ans: a 4 byte fixed length integer to store the game ID + 2 one byte delimiters + TotMoves x 2.5 bytes
= 4 + 2 + TotMoves x 2.5 = 6 + (TotMoves x 2.5)

Term 2: Storing snapshots of the game
Basically, since we are storing the smallest sized rectangle for the checkpoints. We are storing k = √TotMoves ~ N checkpoints and each checkpoint would occupy atmax N^2 bytes (one for each cell of the board) at worst (I have provided a better bound using AM-GM equality in the detailed section).
The worst memory needed here: N^3 approx
Usually, much less memory would be needed here due to the compression of neighboring cells in the same row.
We would also need (4 x √TotMoves) bytes to store the coordinates of the corners of the rectangles as well.

Term 3: Storage needed to navigate the game in real time (temporary storage ie can be relinquished after user has finished analyzed the game)
There are N x N cells. Assuming a character based encoding of empty, white and black, each cell will occupy one byte.
Let a page be capable of storing “p” bytes. Then, it can store √p elements across either dimension.
Therefore, memory needed would be:
$[ceil(N/\sqrt{p})]^2 \times p$

Total memory = sum of all the above terms

Summarizing,

Recapping the various benefits of our design,

- Due to truncation + compression in the storage based page design (#1), we are storing as minimal storage as needed per snapshot.
- The feature of taking snapshots is helping us answer queries of type Q3 very efficiently .
- The usage of page design (#2) for updates makes updates extremely quick. Storing the square submatrices in each page instead of flattening the board grid helps us take advantage of locality of reference during disk-page accesses for all 3 types of updates: row-based, column-based and diagonal based updates. This is especially helpful for Reversi where all updates are on contiguous neighboring cells possible across all 6 directions.
- Using Huffman encoding based compression for WHITE, BLACK and EMPTY helps us reduce our storage for the state of each board cell by 4 times (from 1 byte to 2 bits).

Page & Algorithm Design for Query Execution for assisting Reversi/Othello players

- Anmol Agarwal (2019101068), Sanchit Arora (2019101047)

Some terminology/notation:

- **N** = side of the board => N x N = number of cells on the board | N<=1e3

- For ease of reference, without loss of generality, we consider the initial configuration of the board played i.e. having 4 disks as the first four moves. Hence, MAX MOVES in our case: N x N (since we are considering the initial configuration of the game to consume 4 moves)
- **G**: game ID
- **TotMoves:** total number of moves played in the game.
- $\mathbf{M_i}$: ith move of the game where 1<=i<=TotMoves. (NOTE: M1, M2, M3 and M4 are fixed since they correspond to the initial configuration of the game)
- $\mathbf{S_i}$ = state of the board after Move “i”
- **TotGames**: Total number of games in the database so far
- K = Number of nearest neighbours to be considered (Let k<=50)
- Intermediate_K: number of nearest neighbours retained in the pruning step: (let Intermediate_K be of the order <= 500)

Approach to finding K nearest Neighbours and revised page design:

Part 0: Defining a subproblem whose solution helps us solve the later part of the assignment

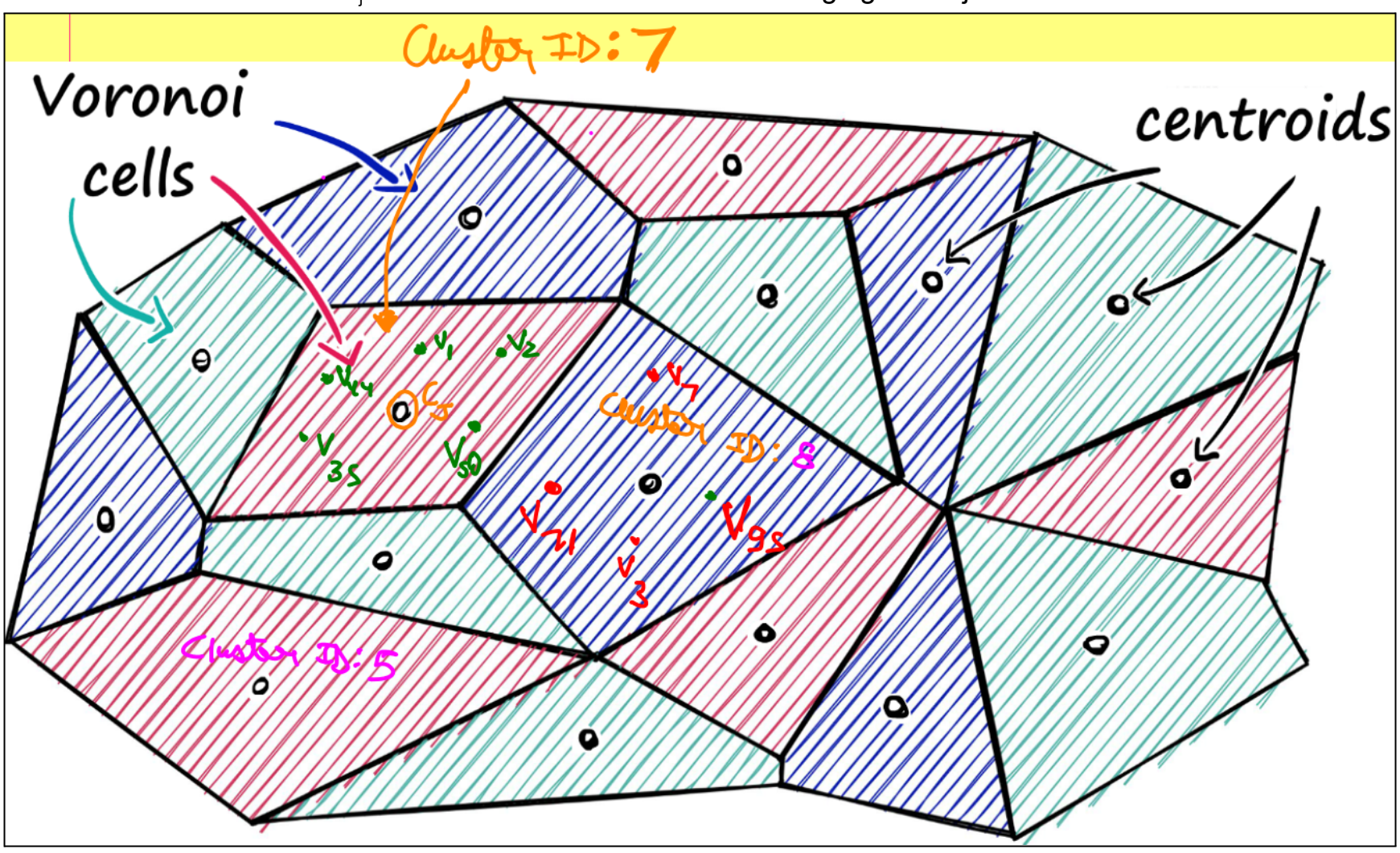
First, we will describe a new unrelated subproblem and then, we will use subproblem in our method to find the K nearest neighbours.

Subproblem: Let V = {V_1, …, V_t} be a set of “t” vectors, each having “d” dimensions. We need to find the “S" approximate nearest neighbours to a new query vector V_{query} .

Approach:

Inspired by FAISS and Voronoi Tessellation, we solve this problem in the following way:

- For the sake of our example, we consider “t” to be “100”.
- STEP 1 (**One time preprocessing step**) : Firstly, we cluster these “t” vectors into some predetermined “J” number of clusters and denote C_j as the centroid of the vectors belonging in the jth cluster.

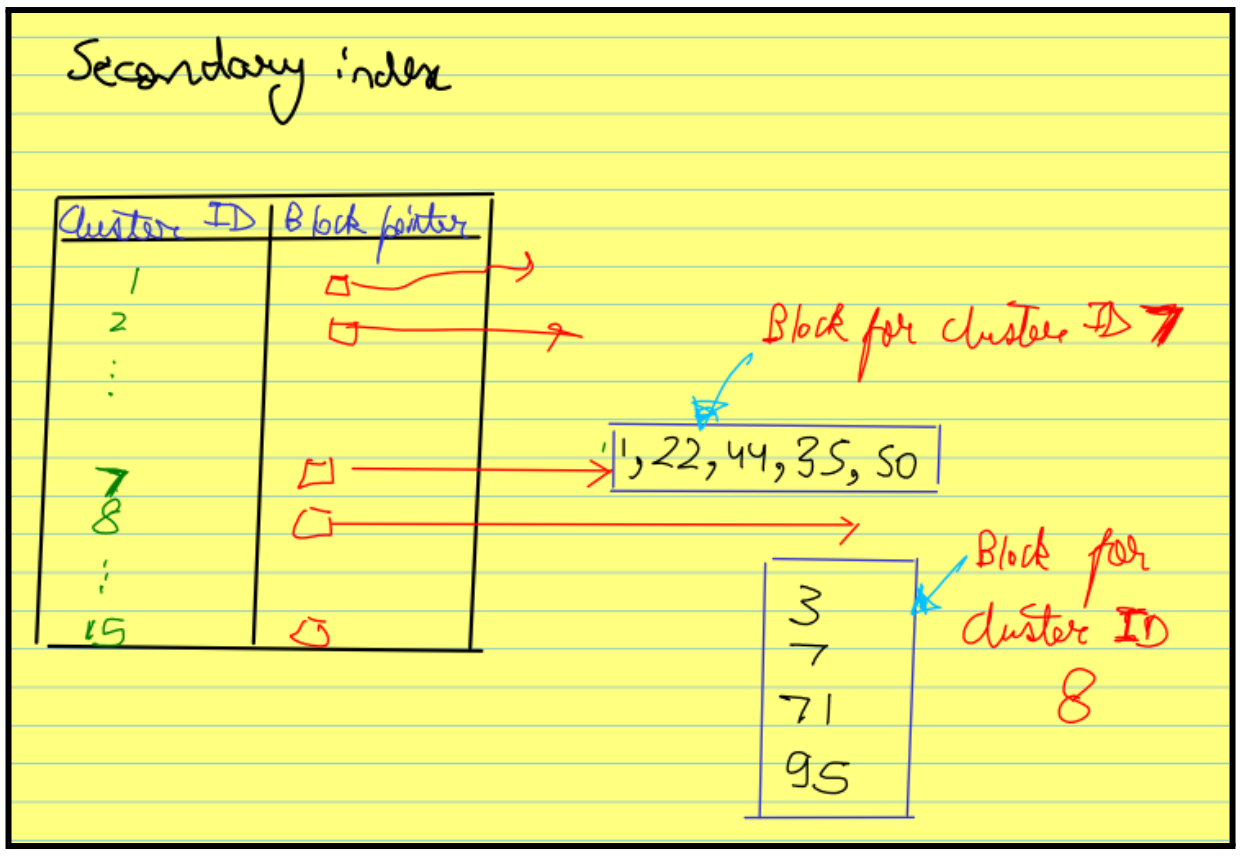


- In the above diagram, “J” i.e. number of clusters = 15.
  - Vectors V_1 , V_2, V_{44} V_{35} V_{50} belong to cluster ID: 7 ; C_7 is the centroid of the included vectors.
  - Vectors V_3 , V_7, V_{95} V_{71} belong to cluster ID: 8 ; C_8 is the centroid of the included vectors.
- STEP 2: Now, instead of naively searching for the nearest neighbour for V_{query} by comparing it with all “t” vectors, we compare it with only the “J” centroids (J<<t) and determine the cluster whose centroid is closest to it. Let’s the closest cluster have ID: “ClosestCluster”
- STEP 3: Then, we now, calculate the “S” nearest neighbours of V_{query} by comparing it with all members of cluster having ID: “ClosestCluster”

In order to retrieve vector members of the closest cluster (ID: “ClosestCluster”) ie STEP 3, we maintain a table with rows values: <Vector ID, Cluster ID in which vector is present>.

For efficient retrieval, we maintain a **secondary index** on the attribute (Clustering ID ie **nonkey, nonordering field**) where we store the following attribute in the secondary index: <Clustering ID, **block pointer** to block containing vector IDs belonging to the cluster>

For the above illustration, our secondary index has the following structure:

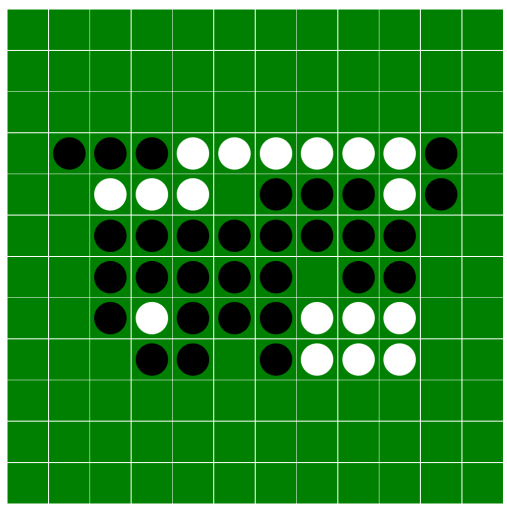


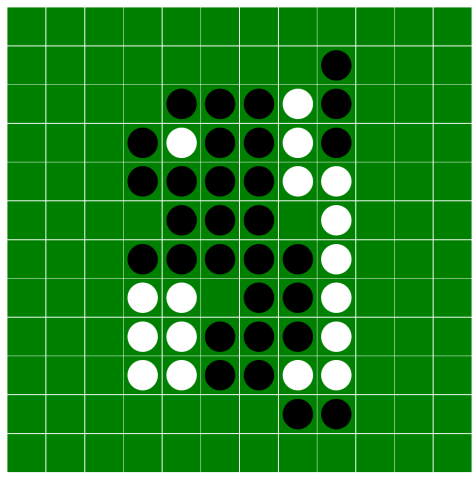
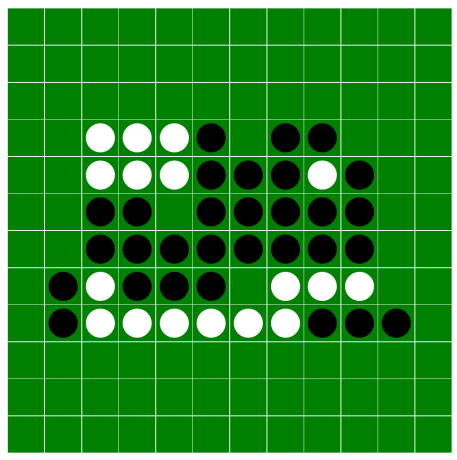
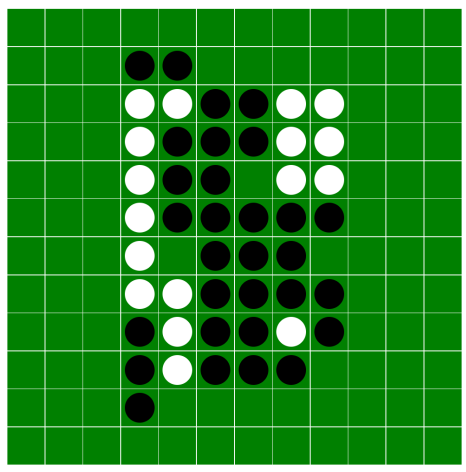
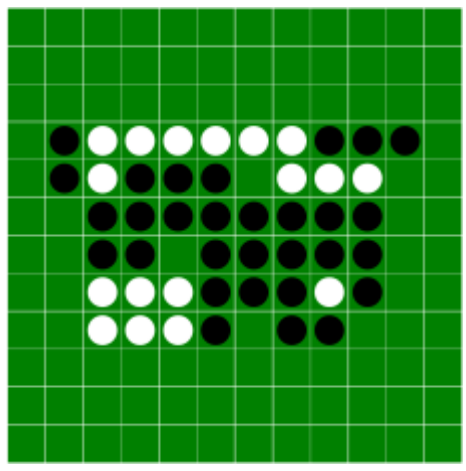
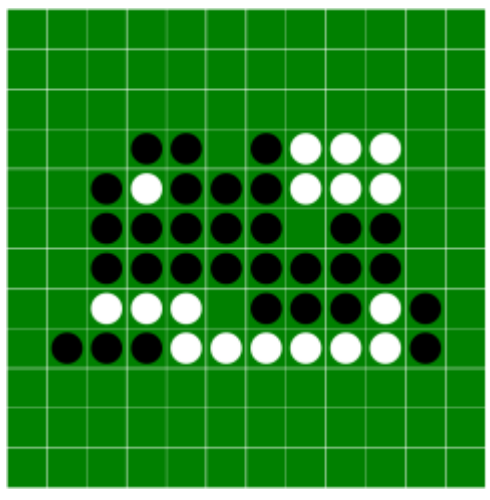
NOTE: if we wish to add newly played games to this index, we can **lazily** update/recreate the clusters after every “H” new games.

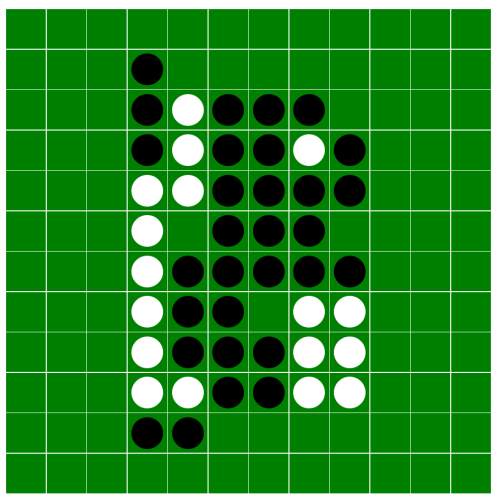
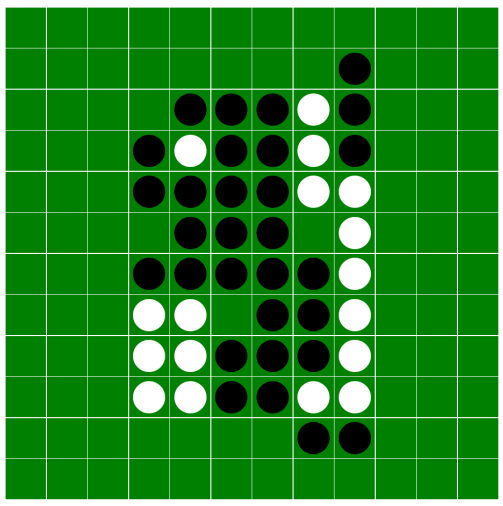
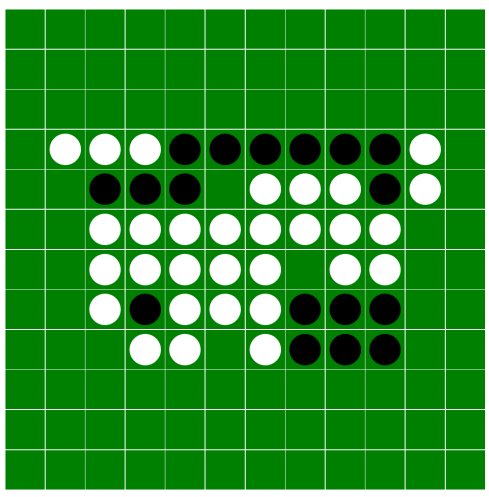
Query answering Time complexity :
O(*dimensions in each vector* X *number of clusters*) + O(*dimensions in each vector* X *number of vectors in selected cluster*) + O(Number of block accesses in the secondary index)

Part 1: What are the desirable properties our similarity metric should cover ?

The metric should show high similarity for $S_{G,i}$ with $S_{G,i}$ itself but also with the below versions of $S_{G,i}$ after rotation, flipping, colour inversion etc as can be seen below.

Operation performed	**Notation to depict the board state after the performed operation**	**Board state after operation performed**
Original Board	$S_{G,i}$	

Rotation by 90 degrees	$S^{90}_{G,\,i}$	
Rotation by 180 degrees	$S^{180}_{G,\,i}$	
Rotation by 270 degrees	$S^{270}_{G,\,i}$	
Horizontally flipped	$S^{H}_{G,\,i}$	
Vertically flipped	$S^{V}_{G,\,i}$	

Flipped across diagonal 1	$S^{D1}_{G, i}$	
Flipped across diagonal 2	$S^{D2}_{G, i}$	
Each cell's colour inverted	$S^{\#}_{G, i}$	

Part 2: Candidates for base exact similarity metric between 2 given states ie board configuration: P and Q ie BaseExactSim(P, Q)

Choice 1 for "Base Exact Similarity Metric": Number of unmatching cells needed

- Given states P and Q (each with N x N elements), when using this metric, we define
  - Let state of cell (i,j, P) = state of cell(i,j) in game state "P" =:
    - 0 if the cell is white
    - 1 if cell is black
    - 2 if cell is empty
  - **BaseExactSim(P, Q) :** number of cells in (i,j) where *state(i,j,P)!=state(i,j,Q)*

Time complexity (Assuming that states P and Q are already in main meory**):**

- Checking each cell will require *O(1)*
- There are total N^2 cells
- Therefore, time-complexity = $O(N^2)$

Choice 2 for "Base Exact Similarity Metric": Jaccard based similarity

Metric proposal

- Jaccard Row Similarity:
  - Let R_P = set of unique rows in board configuration "P"
  - Let R_Q = set of unique rows in board configuration "Q"

- **Jaccard_Rows(P, Q) =** $\frac{R_P \cap R_Q}{R_P \cup R_Q}$
- Similarly, Jaccard Column Similarity:
  - Let C_P = set of unique rows in board configuration “P”
  - Let C_Q = set of unique rows in board configuration “Q”
  - **Jaccard_Cols(P, Q) =** $\frac{C_P \cap C_Q}{C_P \cup C_Q}$
- Then,
  - **BaseExactSim(P, Q) :** Total Jaccard Similarity = **Jaccard_Rows(P, Q) + Jaccard_Cols(P, Q)**

How can (R_P U R_Q) and (R_P ∩R_Q) be calculated ?

- Each row is a sequence : E,W,B depending on whether the cells in the row are white, black or empty.
- We can convert this string of E,W and B into a binary string using Huffman Compression.
  -

STATE	Proposed bit representation **(2 bits)**
WHITE CELL (W)	01
BLACK CELL (B)	10
EMPTY CELL (E)	00

Our solution: Now that we can represent rows as binary strings, we can calculate intersections and unions using Dynamic Hashing OR Extendible Hashing on the binary strings (Essentially, we are interested in using a Trie type data structure. But since we cannot fit a trie in memory, we use the database equivalent of tries ie Extendible Hashing based data structure.)

Calculating number of elements in (R_P U R_Q):

- First, the hash data structure is empty.
- Iterate through all rows of R_P and insert into the data structure if the row already does not exist. If the row already does not exist, insert UnionCnt by 1.
- Iterate through all rows of R_Q and insert into the data structure if rows already does not exist. If the row already does not exist, insert UnionCnt by 1.

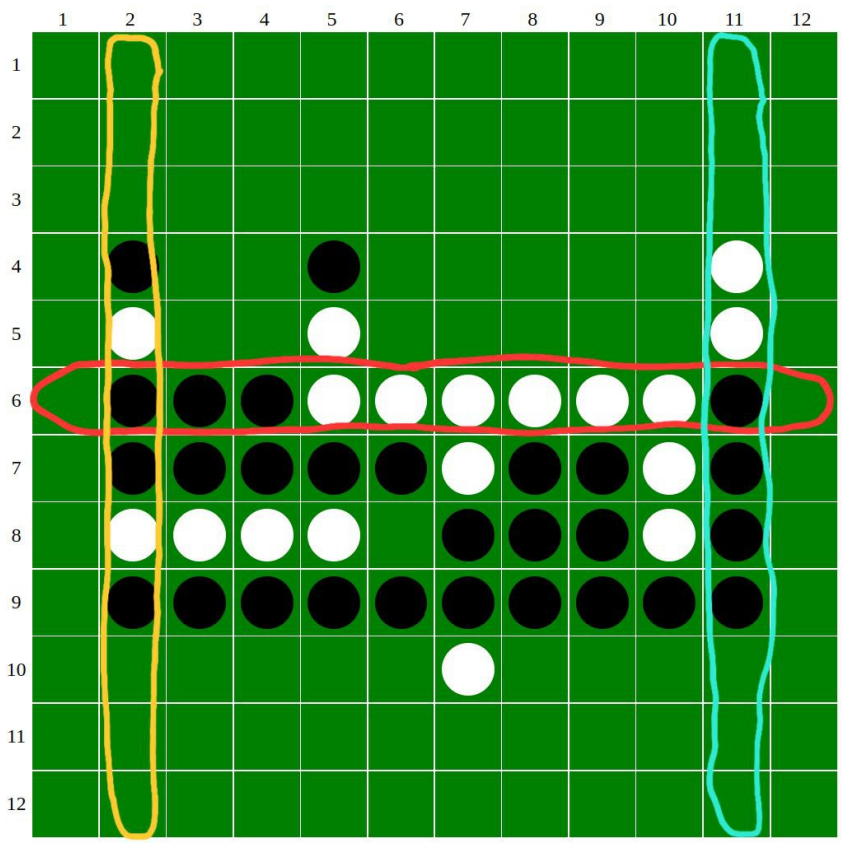
Calculating number of elements in (R_P ∩ R_Q):

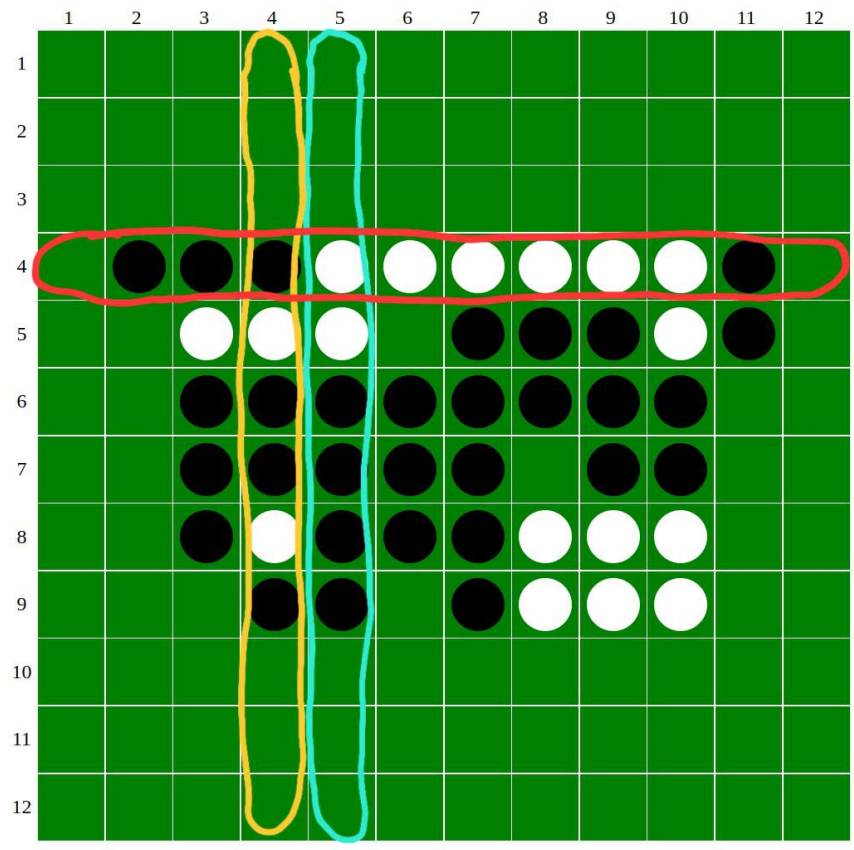
- First, create 2 extendible hash data structures (H1 and H2) is empty.
- Iterate through all rows of R_P and insert into the data structure H1 if the row already does not exist.
- Iterate through all rows of R_Q.
  - If the row existed in H1 but not in H2.
    - Then, increment IntersectCnt by 1.
    - Insert row in H2.

The Jaccard Value can be calculated using: **UnionCnt** and **IntersectCnt.**

Example:

Board 1 (P)	Board 2 (Q)





In the above example, as shown with the matching colours, these match:

- Row 6 in Board 1 with Row 4 in Board 2
- Column 2 in Board 1 with Column 4 in Board 2
- Column 11 in Board 1 with Column 5 in Board 2
- All empty rows/columns

Jaccard_Rows = 2/12 = 1/6
Jaccard_Cols = 3/19
BaseExactSim = 1/6 + 3/19 = 0.32456

Time complexity:

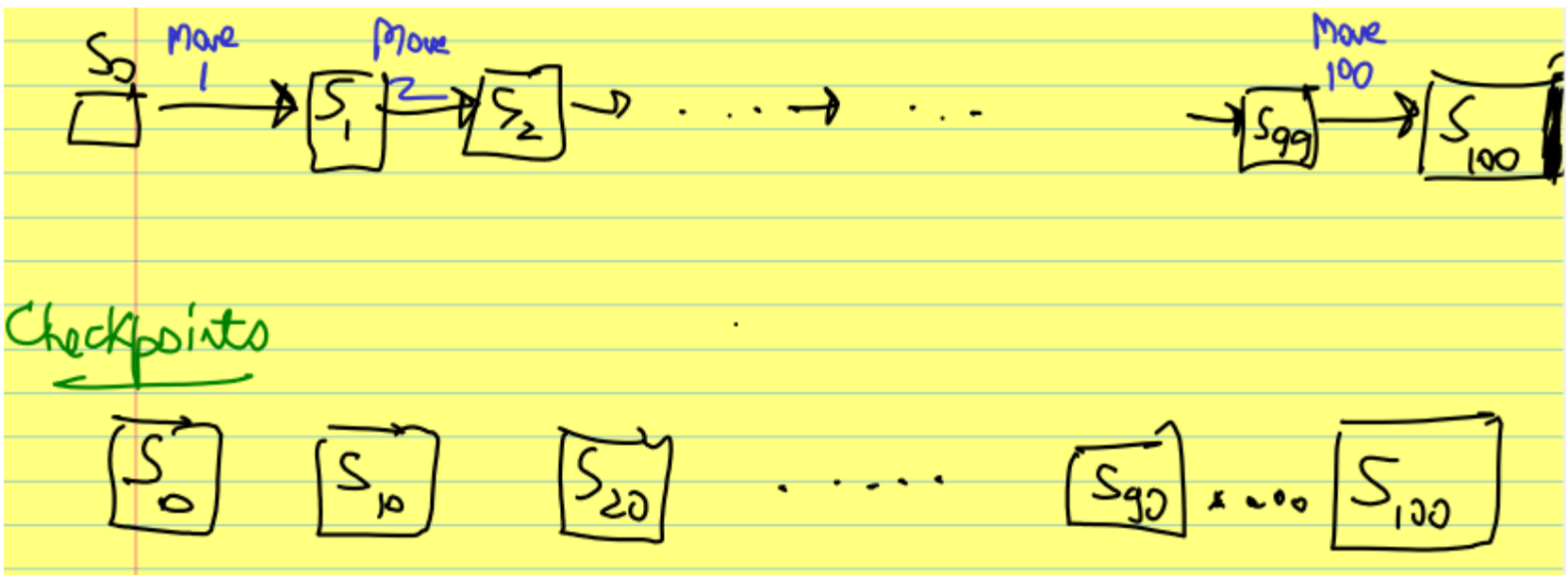
- Total number of rows in each state = *N*
- Hashing each row will require iterating through each row element (i.e. number of columns), therefore *O(N)*
- Therefore, time-complexity of creating the extendible hashing data structure = $O(N^2)$

Part 3: Defining alternate states of a state of the game

- Let Variations($S_{G,i}$) = set of states = {$S^{90}_{G,i}$,$S^{180}_{G,i}$, $S^{270}_{G,i}$, $S^{V}_{G,i}$,$S^{H}_{G,i}$, $S^{D1}_{G,i}$, $S^{D2}_{G,i}$}
- Then, let us define **Alts($S_{G,i}$)** = Variations($S_{G,i}$) U Variations($S^{\#}_{G,i}$)

Part 4: Pruning the search space for shortlisting candidates at move “i”:

- Recapping from our previous submission, for a game “G”, with TotMoves total moves, we were storing checkpoints of the game state after every √TotMoves. This helped us to retrieve the state of the game at any move “i” in O(√TotMoves) [how ?: explained in previous submission]
- For a game “G” with 100 total moves, we store a checkpoint every 10 moves.



Proposed algorithm for finding the K nearest neighbours of the ith state of game G ie $S_{G,i}$

Inputs:

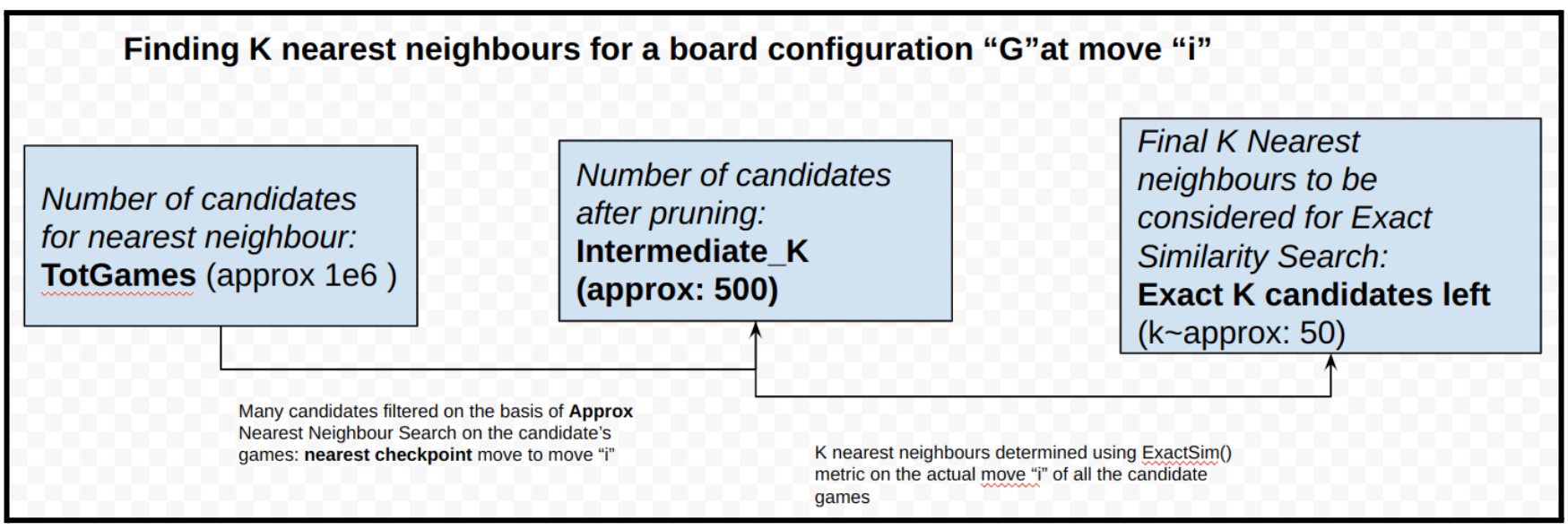
- Game ID: G
- State ID: i
- State of the game: $S_{G,i}$

Some notation:

- TotGames: number of games played in the past
- N : size of game board

Preprocessing:

- For **EACH** move IDs "M" in {0, √TotMoves, 2 * √TotMoves, …, √TotMoves x √TotMoves },
  - create a voronoi based index (as in part 0) using flattened vectors of {$S_{G1,M}$, …, $S_{GTotGames,M}$ } **and their variations Alts($S_{G,i}$)** [as described in Section 3] in with total "TotGames" vectors with "N x N" dimensions each with each vector element being:
    - 0 if white
    - 1 if black
    - 2 if empty
- There will be √TotMoves such voronoi based indexes.



Algorithm:

- Initially, there are TotGames candidates for nearest neighbours.
- We reduced number of candidates from "TotGames" to "Intermediate_K" by:
  - Let move "M" be nearest move to "i" at which checkpoint was taken
  - Heuristically, find the "Intermediate_K" nearest neighbours of $S_{G,M}$ (NOTE: this is NOT using $S_{G,i}$) via the Voronoi index constructed at move "M" using **Part 0.**
  - **Stats:**
    - Block accesses:
      - T1: O((N x N) X number of clusters) to find nearest cluster
      - T2: O(log(Number of blocks in secondary index)) + O(Number of blocks containing vectors of selected cluster)

        - **Ans: T1 + T2**
    - NOTE: these filtered neighbours, include all variations of a board: ie rotations, ,flips etc.
- Now, for each of these “Intermediate_K” neighbours, find the similarity using one of the COMPUTATIONALLY COSTLY choices of: **BaseExactSim(**P, Q**)** described in Part 2.
    - **Stats:**
        - Block accesses:
            - T1: O(N^2/p) for calculating similarity of $S_{G,i}$ with a candidate [refer to previous submission for “p”]
            - **Total:** T1 x (number of intermediate neighbours)
    - Among these “Intermediate_K” neighbours, **choose the final K nearest** boards using after sorting per “BaseExactSim($S_{G,i}$, Neighbour N)”.

ANSWERING QUERIES

For the rest of the assignment, let us consider that the player is making queries while the game is currently at move “C”.

Part 5: Page Design

Preserving previous Page Design:
We’ll be keeping our page design as proposed in the previous part as it is, i.e. we will be keeping: “storage based page design” and “update based page design” as it is.

Adding new page designs:
Other than that, we’ll be storing some extra pages for each game to optimise the new queries. The details for the new page design is below:

For each board, at each checkpoint, for each cell *(i,j)*, we’ll be storing the number of flips. As we do with the state of games, i.e. use the state stored at checkpoints to generate any intermediate state required, we’ll do the same for the number of flips as well.

Storing number of flips:
We have two options for storing number of flips at each checkpoint:
1. Store the number of flips from the starting of the game till the last move of that checkpoint.
2. Store the number of flips that happened after the last checkpoint.

Considering the size of the board we’re considering (i.e. of the order 1000 X 1000), and the nature of the game, there will be many cells that won’t change much after every checkpoint.

So, at one checkpoint we’ll store, let’s say, 1000 flips and now even if there is no change we’ll keep storing 1000 (or 100x if let’s say there are <= 9 flips). This results in storing unnecessarily high numbers, that too, for each cell, whereas in the 2nd approach, we can store smaller numbers (1 digit in case of the example).

Now let’s discuss exactly how we’ll put the data in page design.

Since at each checkpoint we choose to store only the number of flips that happened after the previous checkpoint, the **maximum number will be $\sqrt{TotMoves}$,** which is roughly 1000.

Storing an integer requires 4 bytes (or 32 bits), whereas we only need at maximum **10 bits** (2^{10} = 1024 > 1000). Therefore, we assign each number of flips a fixed size of 10 bits. These bit strings can then be taken into sets of 8 bits and converted to character for storage (as we did with storing the moves in our original page design, in previous assignment).

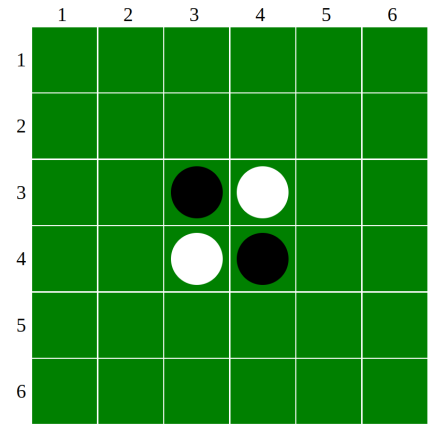
Let’s say the page size allows storing p^2 such flip numbers (or in other words, we can store $10p^2$ bits in a page). Then we’ll divide the whole board into subgrids of size ***ceil(N/p)* X *ceil(N/p)***, allowing each sub-grid to be of a size that we can fit in a single page.

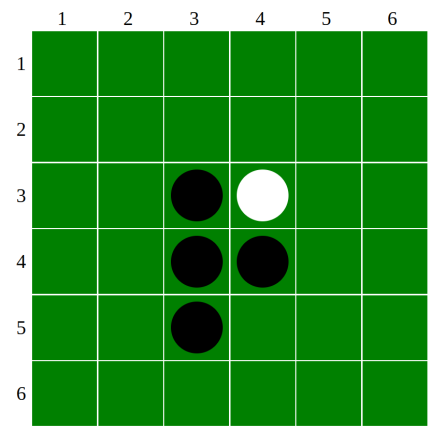
To optimise some queries (discussed later in the Queries section) even more, we’ll be storing the total number of flips for each position throughout the game as well.

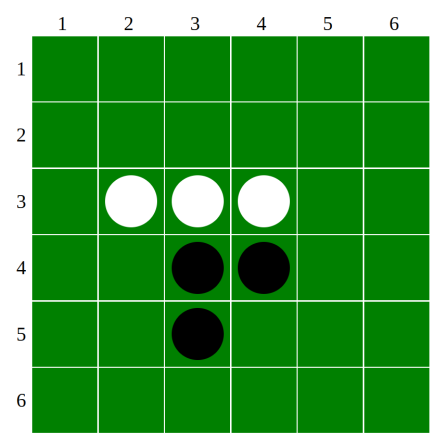
Since this requires storing only one number for each position for a whole game (rather than for each checkpoint), this storage will be negligible (~0.1%) compared to the total storage we're using. Hence, we need not think a lot to optimise storage in this case and any way should work well.

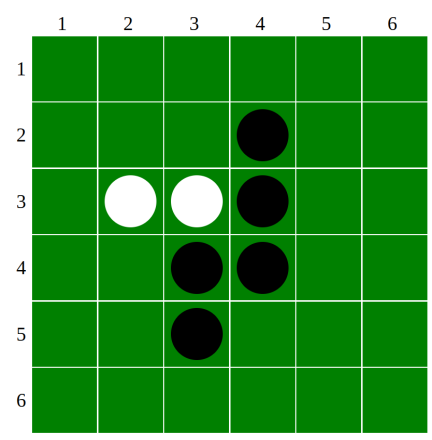
Example:
For the sake of example, let's take a board of size 6X6, which will store checkpoints after every 6 moves.

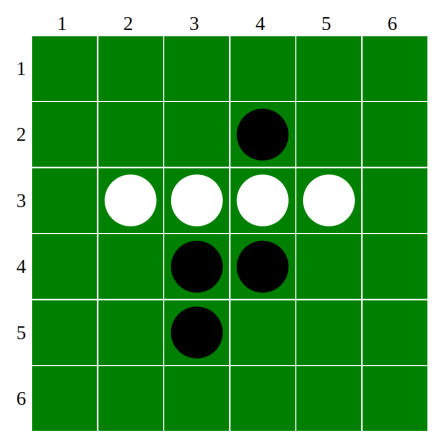
The game proceeds as follows:

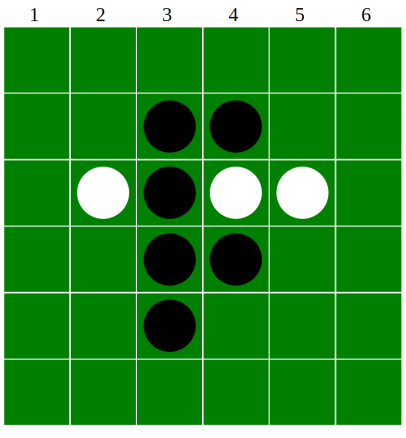
1. 

2. 

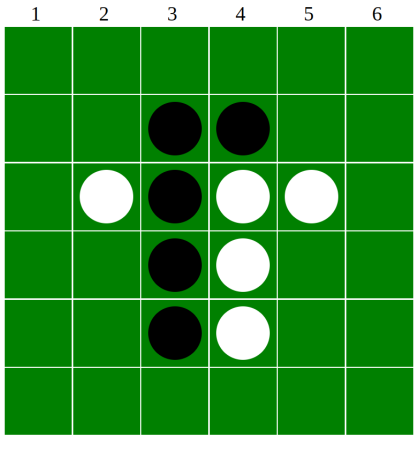
3. 

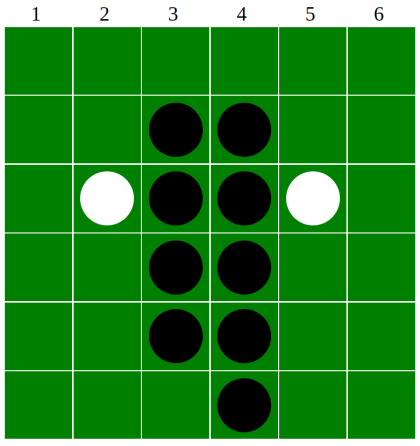
4. 

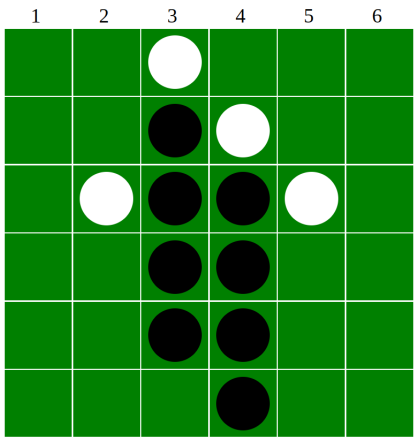
5. 

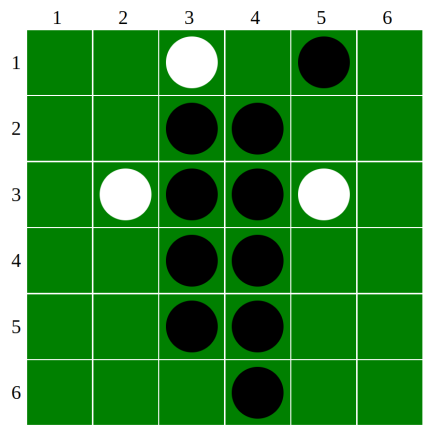
6. 

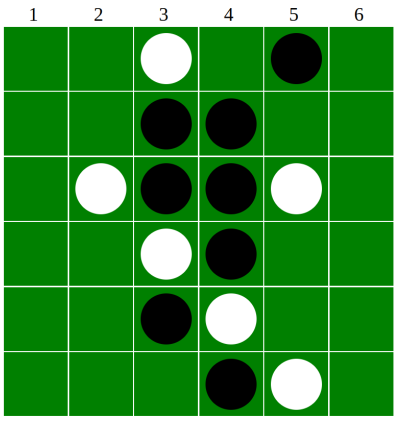
CHECKPOINT

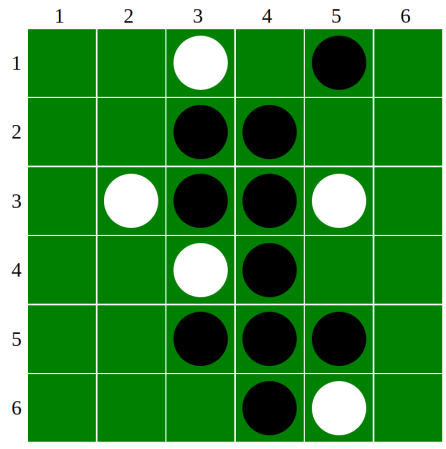
7. 

8. 

9. 

10. 

11. 

12. 

CHECKPOINT

For the sake of simplicity, we don’t show moves after 12.

Now, the flips at the first checkpoint (after move 6) will be:

0	0	0	0	0	0
0	0	1	1	0	0
0	1	2	2	1	0
0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0

We observe that till the first checkpoint, the cell ***(3,3)*** (highlighted in green) flipped twice, once in move 3 and second time in move 6.

Similarly we can find the flips for each cell at checkpoint 2 (after move 12) as well.

Query 1 [LONGEVITY]

You are given a position (i,j) and the move on which you are currently at. Then you need to answer the following:

a) Whether the color at position (i,j) will be flipped in the next move or not.
b) For how many moves you can guarantee that this position will not be flipped.
c) In the next move, all positions in the board that are going to become flippable.

Solving (a)

1. We’ll first find KNB using the current state of the game.
2. Proceed each of the KNB to 1 more step (we know the future of all those boards/games).
3. For each KNB, check if the position *(i,j)* flipped or not.
    - Empty cell -> White => Flip
    - Empty cell -> Black => Flip
    - White -> Black => Flip
    - Black -> White => Flip
    - In all the other cases => No Flip
4. There will be some boards saying that the given position will flip, and some will say not flipped, to decide the result **take majority**.
    - In case there is a tie (will only happen when *K* is even), **we’ll be pessimistic** and declare not flipped.

Time complexity (excluding the time taken to find KNB):

- For each board, going to next step will take $O(N^2)$ to simulate that move
- Total boards: number of nearest neighbours = K
- Therefore, time complexity = $O(KN^2)$

Solving (b)

In the proposed page design, at each checkpoint, for each cell we store the number of times the cell was flipped after the last checkpoint.

1. Do steps 2 to 4 for each of the KNB:
2. We'll loop over all the checkpoints to find the first next checkpoint which has non-zero flips for position *(i,j)*.
3. If, let's say, checkpoint p_i has non-zero flips, then find the first move between p_{i-1} and p_i where the flip happened.
4. Now, the number of moves with no flips is all the moves between current move and p_i + moves between p_i and p_{i+1} with no flips.
5. To guarantee no flips, **we'll be pessimistic and take the minimum of the results** given across all KNB.

Time complexity (excluding the time taken to find KNB):

- For each board $O(\sqrt{TotMoves})$ will be required to find the required checkpoint and then loop after that checkpoint to find the required move.
- In each iteration of loop (mentioned above), $O(N^2)$ will be required to simulate that move.
- This will be done for K boards.
- Therefore, time complexity = $O(K N^2 \sqrt{TotMoves})$

Alternate Approach

- An alternate approach would be to store the cumulative number of flips at each checkpoint (instead of storing only the change from the last checkpoint). This change can be made in PART 5 if implemented.
- This would result in reducing the time-complexity of finding the first next partition with the same number of flips (which now means that the change is 0) from $O(\sqrt{TotMoves})$ to $O(logTotMoves)$.
- But, finding the exact move within the partition will still take $O(\sqrt{TotMoves})$, so the worst case in this alternate approach would still be the same as before.

Solving (c)

This part is an enhanced version of part (a).

1. First find KNB using the current state of the game.
2. Advance each of the KNB to the next move.
3. For each position *(i,j)*, follow the first 3 steps of part (a).
4. To decide whether position *(i,j)* becomes flippable or not, if it is flipped in any of the KNB, we'll say it is flippable, otherwise not.

Time complexity (after finding KNB):

- For each board, going to next step will take $O(N^2)$ to simulate that move
- Total boards: K
- Therefore, time complexity = $O(KN^2)$

Query 2 [DIFF IN FLIPS]

Suppose you are at the ith move and when player 1 plays (white) it turns "k" blacks to white and subsequently when player 2 plays (black) it turns "l" whites to black. Then we can define gain for player 1 as (k-l). Now, you need to answer the following:

a) Return all the i's such that player 1 is gaining at least "b"
b) Return all the sequences of moves (i,i+1,i+2, ... i+p) such that player 1 is continuously gaining at least b in all these moves and p is the highest such number.

For both the parts, we assume returning *i* means returning the move number.

For the sake of simple explanation, we also assume that returning all means throughout the game, and not only from the current move number till the end of game. Even if the requirement is to return only future moves, the approach would be the same.

Solving (a)

1. Find KNB according to the current state of the game.
2. Mark each move as positive (simply keep an array of size 1e6, in this case).
3. For each KNB, do steps from 4 to 6.
4. Go to each positive move one by one.
    a. If the next positive move is in some later checkpoint, we can use the stored checkpoints to efficiently go to that state instead of incrementing moves one by one.
5. Check if the player is gaining at least *b*:
    a. This would require checking the next two moves.
6. If the player gains less than *b* for some move number, mark that move as negative.
7. Return all the move numbers that are marked as positive.

Time complexity (after finding KNB):

- Looping over moves will require $O(TotMoves)$
- For each such move, simulating next two moves will require $O(N^2)$
- This needs to be done for *K* boards
- Therefore, time complexity = $O(K\ N^2\ TotMoves)$

Solving (b)

This part is an extension of the above part.

1. Find KNB according to the current state of the game.
2. Give each move a value of an infinite number, let's say, 1e9 (simply keep an array of size 1e6, in this case).
3. For each KNB, do steps from 4 to 6.
4. Go to each move having value greater than 0
    a. If the next required move is in some later checkpoint, we can use the stored checkpoints to efficiently go to that state instead of incrementing moves one by one.
5. For each such move, find the maximum *p* such that at every move the player gains at least *b*
    a. This would require checking the next *p* moves.
    b. If we're doing it for consecutive moves, we can use the sliding window technique to optimise checking next *p* moves for all those consecutive moves.
6. Set the value of the move as minimum of the original value and the highest *p* we found in the above step.
7. Return all *(i,p)* pairs (*i* is the move number, and *p* is the value stored for that move) where *p* is non-zero.

Query 3 [MAX FLIPS]

Return all locations (i,j) which suffered the maximum number of flips which happened throughout the games played.

In the proposed page design, we're storing the total number of flips for each game/board, and for each position.

1. Find the KNB using the current state of the game.
2. For each position *(i,j)*, have a counter, initialised with 0.
3. Now for each KNB, for each position, increment the corresponding counter with the total number of flips of that position in that game.
4. To find the locations which suffered the maximum number of flips, take the position with the maximum value of the corresponding counter.
    a. In case of ties, return all such locations.

Time complexity (after finding KNB):

- For each board, we need to get total number of flips for each cell which will require $O(N^2)$
- This needs to be done for *K* boards
- Therefore, time-complexity = $O(K\ N^2)$

Query 4

For the given board orientations, give me all the positions (i,j) which will not change forever (based on the KNB) . That is No matter what you or the other player plays the color at this position (i,j) will not change.

Case 1: Forever means from beginning to end

This case boils down to a slightly different version of Query 3.

1. Find the KNB using the current state of the game.
2. Keep all the positions *(i,j)* marked as not changed.
3. For each of the KNB, for each position, if the total number of flips is more than one, then mark that position as changed.
    a. We take more than one, because one flip is counted when turning an empty cell to black/white cell.
4. Return all the positions marked as not changed.

Time complexity (after finding KNB):

- For each board, we need to get total number of flips for each cell which will require $O(N^2)$
- This needs to be done for K boards
- Therefore, time-complexity = $O(K\ N^2)$

Case 2: Forever means from this (or the very next) move to the end

- Very similar to the Case 1
- In step 3, instead of taking the total number of flips, we'll need to take the number of flips after the current move.
    - Since we're storing the number of flips at each checkpoint, we can compute this by looping over all the checkpoints.

Time complexity (after finding KNB):

- For finding the number of flips in all future checkpoints, $O(\sqrt{TotMoves})$ will be required, and $O(N^2\ \sqrt{TotMoves})$ for simulating moves and finding flips in the current checkpoint.
- This needs to be done for K boards
- Therefore, time-complexity = $O(K\ N^2\ \sqrt{TotMoves})$

OPEN ENDED queries

Other Query 1: Check if the current state is deterministic and the winning player is already decided

1. Find KNB according to the current state of the game.
2. Store two counter variables, one for each of the players, both of them initialised to zero.
3. For each board, we'll have the last state as one of the checkpoints, so simply check that to decide if the required player won or not.
4. Depending on which player won, increment the corresponding counter.
5. If either of the counters is still zero, return that the game is at a deterministic state.

Time complexity (after finding KNB):

- To check the winning player of each board, $O(N^2)$ will be required.
- This needs to be done for K boards.
- Therefore, time-complexity = $O(K\ N^2)$

Other Query 2: Check the maximum flips possible in the next move according to KNB. This will allow the player to aim somewhere near that value while taking a move.

1. Find KNB according to the current state of the game.
2. Proceed each board to the next move and find the number of flips.
3. Return the maximum number of flips possible over all the K boards.

Time complexity (after finding KNB):

- To proceed to the next move and find the number of flips $O(N^2)$ will be required.
- This needs to be done for K boards.
- Therefore, time-complexity = $O(K\ N^2)$

KEY TAKEAWAYS

- There is always a tradeoff between utilisation of time and space based resources.
- Query optimization, index optimization, and schema optimization go hand in hand.
- For query execution, though we did not have any logical AND\OR operations, in order for efficiently executing the **“K Nearest Neighbours”** query, we effectively use several optimizations like:
    - **Secondary index** for Voronoi Tessellation in Part 0
    - **Extendible hashing** for our Jaccard based similarity metric
- New page design for executing queries: query 1b, query 3 and query 4
- Sometimes, there maybe overlap in the operations involved in query execution for 2 queries
    - For eg: we use the result query 1a in implementing the retrieval for query(1c) as well.
- Our page design and indexing mechanism depended on our query execution algorithm.
