We will be using the same board storage structure as GA1. This is because the previous structure we used was based on incoming moves and a dynamic hashable index can very easily be implemented on top of it.

A move is defined as: a particular player providing a line connecting two dots. We represent a line using only 3 atomic values:

1) x coordinate of top-left point of line
2) y coordinate of top-left point of line
3) Binary orientation of line 1: horizontal, 2: vertical

- This is both necessary and sufficient to implement & store moves. Initially, the moves database is empty. Once a player makes a move, we check if the move is actually valid, i.e it references a legal position and that line is not one which is already drawn. If the move is valid, we insert the move with its serial number. To perform operations faster; we use a 3 level hash structure:

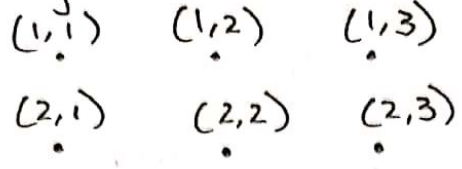
Level 1:
LHS: Value of x-coordinate of top-left point
RHS: Pointer to hash table block containing the corresponding y-coordinates of the top-left points

Level 2:
LHS: A value in hash table block of level 1; composite semantics being of a point — (x, y): storing the top-left corner point of the line
RHS: Pointer to hash table block mapping point to the orientation: horizontal or vertical.

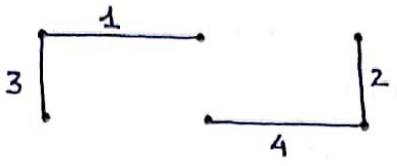
Level 3:
LHS: Composite semantics representing a complete line: (top-left-corner (x, y), orientation)
RHS: An atomic integer value having the move's serial number.

Move Table after 4 moves with number of horizontal & vertical dots being 3 & 2 respectively.

The Board :



After 4 moves:
(Edge Values Representing Move Numbers)

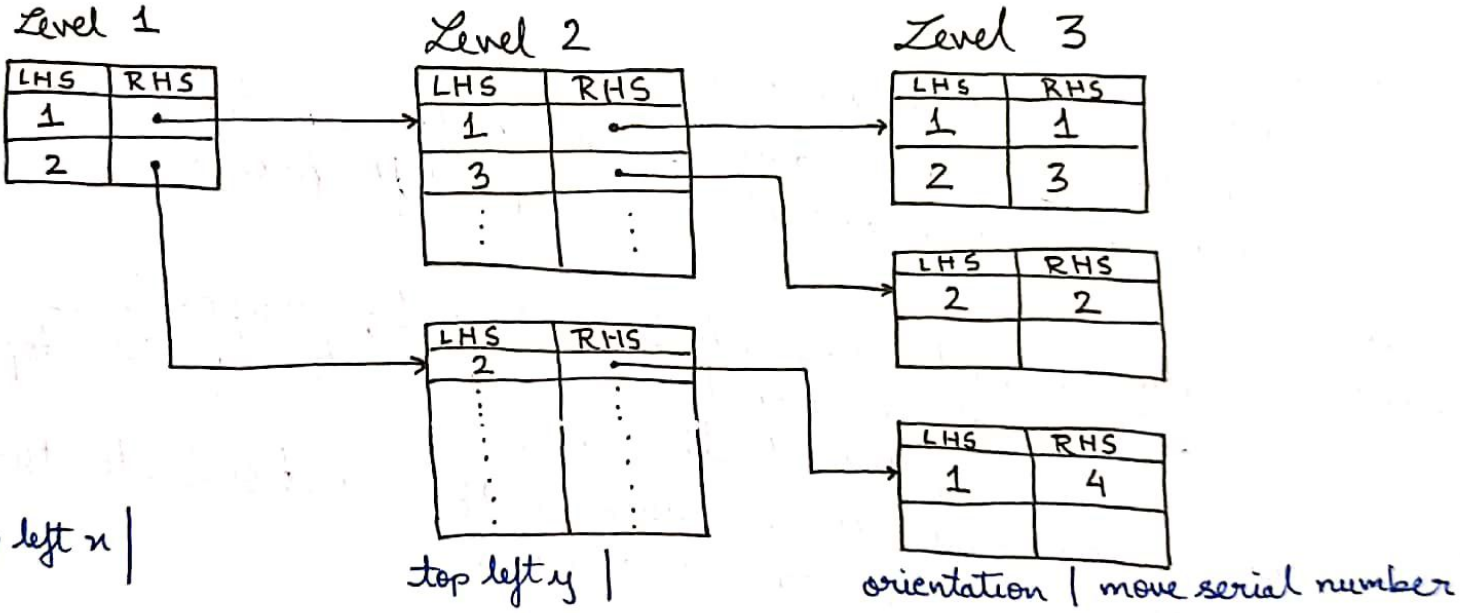


Move Representation:

1	1	1	:	1
1	3	2	:	2
1	1	2	:	3
2	2	1	:	4

- move is of the form: top left x top left y orientation
  ↳ mapping to the move serial number

The index implementation is as follows:



Note: It is not important to store which player has made which move in a typical dots game. The number of players can be anything in set: 2(1)∞ and only the person completing the box gets the point and is allowed to go again.

The level 3 maps or hash blocks will have a maximum number of 2 records for the 2 types of orientations. Since we know this fixed length; we can populate disk blocks efficiently & use an unspanned organization for filling in blocks adhering to the deepest level of our index structure.

We have implemented a simulation of the game in python where our present index structure has been mimicked using python dictionaries.

To check if a move is valid; we check for its legalities given board dimensions & whether that line was marked before. In the 2 player setting, players alternate & a box completion can be checked in amortized $\Theta(1)$ complexity. Box completion can be achieved in one of 2 ways:
if horizontal move: check for boxes above & below — (i)
if vertical move: check for boxes left & right — (ii)
→ in either of (i) & (ii): we need to check for the existence of six lines in our index. We update scores which are stored in CPU registers (as only 2 players, a separate hash structure for player number mapped to current score can also be considered). This allows queries for checking validity as well percentage completion of a particular box in average complexity of $\Theta(1)$.

We have provided our code & it can be used to simulate the actual game with real time feedback & score updates.

Update:
- Player (current) gives a move: check if legal in $\Theta(1)$ — just range assertion & type checking.
: check if move is already present → if move_x in level 1 & move_y in (move_x ← level 1) & orientation in (move_y ← level 2) in $\Theta(1)$
: if valid move: insert level 1: move_x ⟶ level_2: move_y ⟶ level_3: orientation ⟶ move serial number.
: query 6 lines to check if 2, 1, or 0 boxes were completed due to the new line; if yes:- then update current players score accordingly and give next move to current player.

The 3 level structure allows usage of simple hash functions, practically zero collisions for small dimensions & allows for easy querying — we only check for orientation if line exists; we only check for line's top left y if the x coordinate exists. This dynamic hash ensures that we prevent redundant work while preserving simplicity.

As a bonus, we can show how computing & maintaining opportunities & pitfalls are easy with this index structure.

We define a hash structure: immediate-op: a set of boxes which only require 1 more line to be completed. This has the exact same 3 level structure as our index and can be maintained in real time. When a new-line is drawn, if it is already present in immediate-op; then remove it. Now a line can contribute to exactly two boxes (Above & Below if horizontal, left & right if vertical). Hence, a simple check for 6 lines in the hash structure will tell us how complete each box is. Now, this check is very similar to the one for checking if box has been completed. So, we do it in one go itself making it even more efficient. At any time a player can see all the available immediate line making opportunities. To find structures like big box & snake:

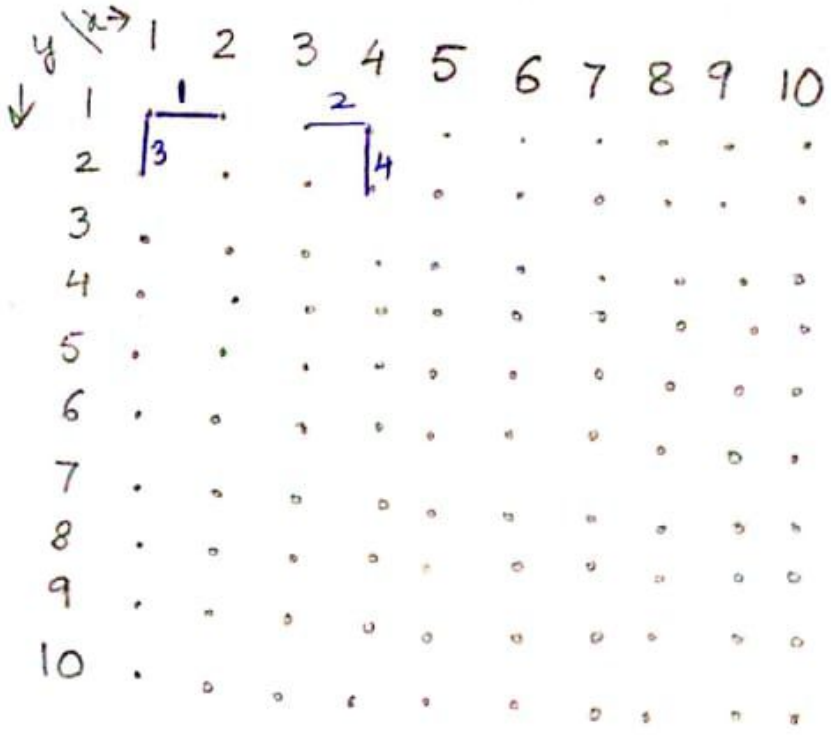
→ expand along opportunities: draw all lines from opportunities set
- place in index with move number as temp-1, temp-2, ... which updates opportunities as well: when no more opportunities remain, the number of temps or moves taken will give you the opportunity rank.

NOTE: If it is possible to draw K lines at one point; the order doesn't matter; you can always draw those K and possibly more (completing a square implies you get the next move). So no need of b.f.s or d.f.s; a simple bag model for completable opportunities will work.

Now, the concept of pitfalls arises only when you cannot complete any boxes or rather the size of immediate_op is 0. Pitfalls can be computed in a few ways:

1) A box is of the form: 4 lines: {(x, y, 1), (x, y, 2), (x+1, y, 2), (x, y+1, 1)}- Hence; if any 2 in the above set are present - drawing the third guarantees atleast one line.
Rank of pitfall can be 0, 1, 2 (0 meaning not a pitfall)

- Thus, while creating the opportunities index & computing scores we can make a pitfall index by checking if new line has completed 2 sides of the box. Indices for pitfalls & opportunities can map to the rank as well— in level 3: replace move number which is not needed with the rank. Using temp-move while calculating opportunity depth lets us know later that it actually is not an actual move. This reduces # deletions.
- Our system allows real time updates for all indices with expected $\theta(1)$ complexity.



Move 1. 1, 1, 1 → 1

Index

LHS	RHS
1	

LHS	RHS
1	

LHS	RHS
1	1

Move 2 3, 1, 1 → 2

LHS	RHS
1	
3	

LHS	RHS
1	
1	

LHS	RHS
1	1
1	2

Move 3 1, 1, 2 → 3

LHS	RHS
1	
3	
1	

LHS	RHS
1	
1	
1	

LHS	RHS
1	1
1	2
2	3

Move 4 4, 1, 2 → 4

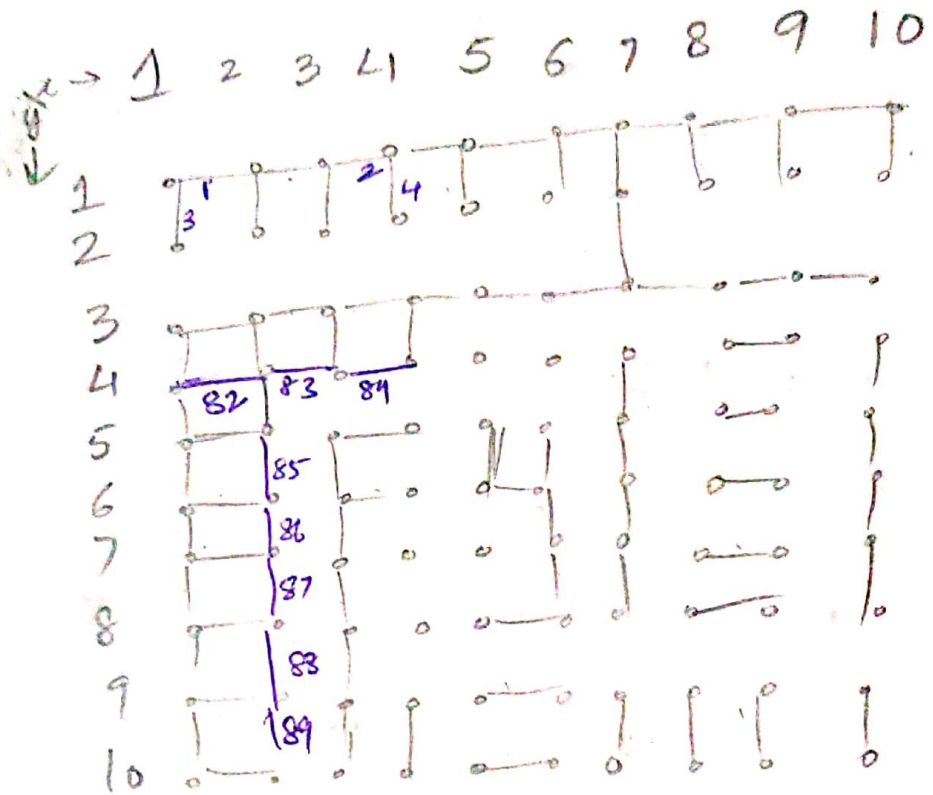
LHS	RHS
1	
3	
1	
4	

LHS	RHS
1	
1	
1	
1	

LHS	RHS
1	1
1	2
2	3
2	4

Note that, if we play optimally it'll take quite a few moves before we see any blocks/boxes being formed. The six line check will also break after a couple of checks at max in the beginning.

Let's skip to a position on the board where some boxes have to be formed. New lines will be marked in blue.



Move 82: 1, 4, 1 → 82
Move 83: 2, 4, 1 → 83
Move 84: 3, 4, 1 → 84
Move 85: 2, 5, 2 → 85
Move 86: 2, 6, 2 → 86
Move 87: 2, 7, 2 → 87
Move 88: 2, 8, 2 → 88
Move 89: 2, 9, 2 → 89

It is easy to see here that move 82 is chosen over all other possible moves (say a (7,4,1)) because opportunity is 2 (while for (7,4,1) it'll be 0). Ofcourse, as explained earlier, we could play move 83 first (which has an opportunity 1) because even then we have the option of playing (82) next. So, whenever we see an opportunity, i.e. we found opportunity >0 for any move in immediate-op, we take it and then search again for next best move.

Now, after move 89, we would not want to play (2,10,1) because (2,9,2) and (3,9,2) already exist in our index structure and the opportunity is 0. So, this is a pitfall (which benefits the opponent with 1 box)

[We say opportunity is 0 because (2,10,1) is not found in immediate-op (which is just one lookup)]

Let's look at what the index looks like after move 89.