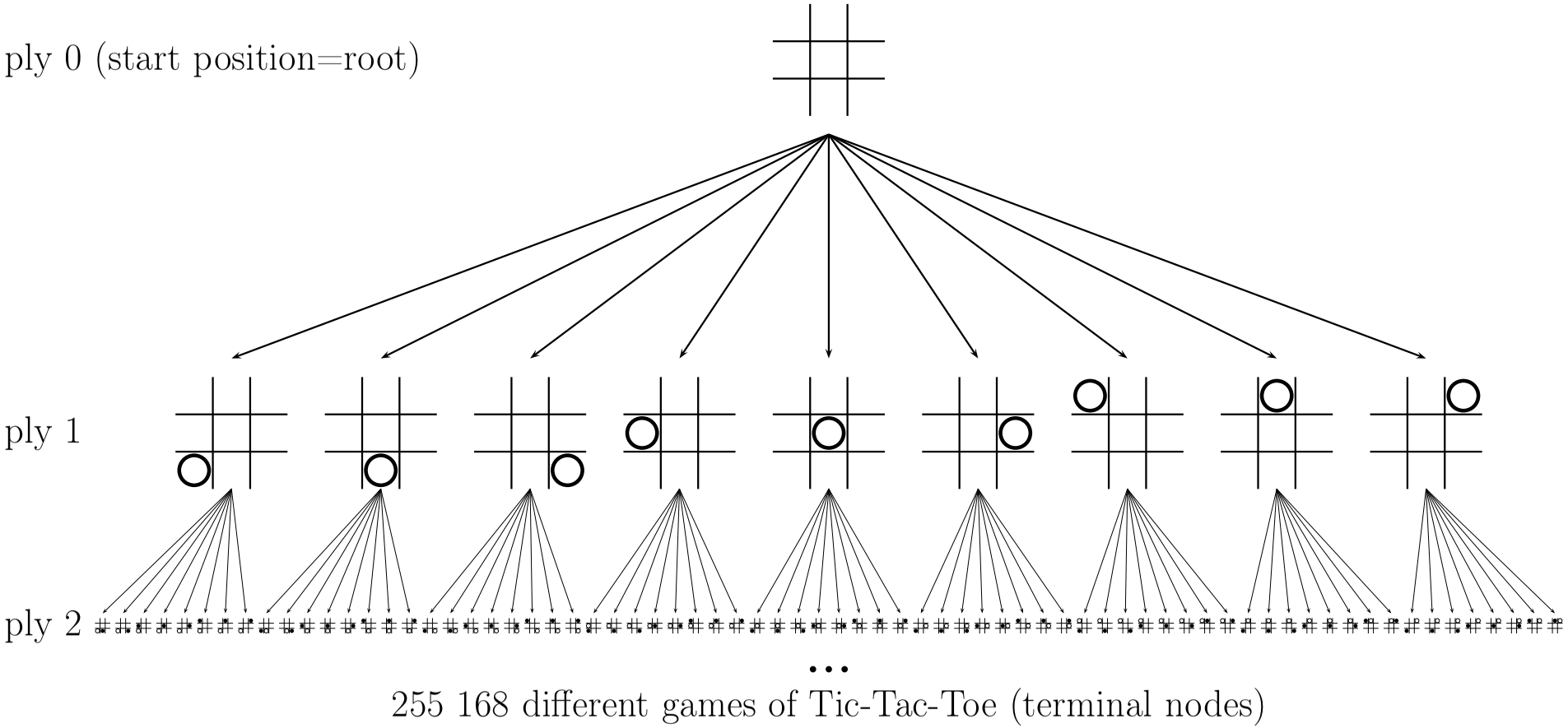
Simple search in game trees

Transposition table concept

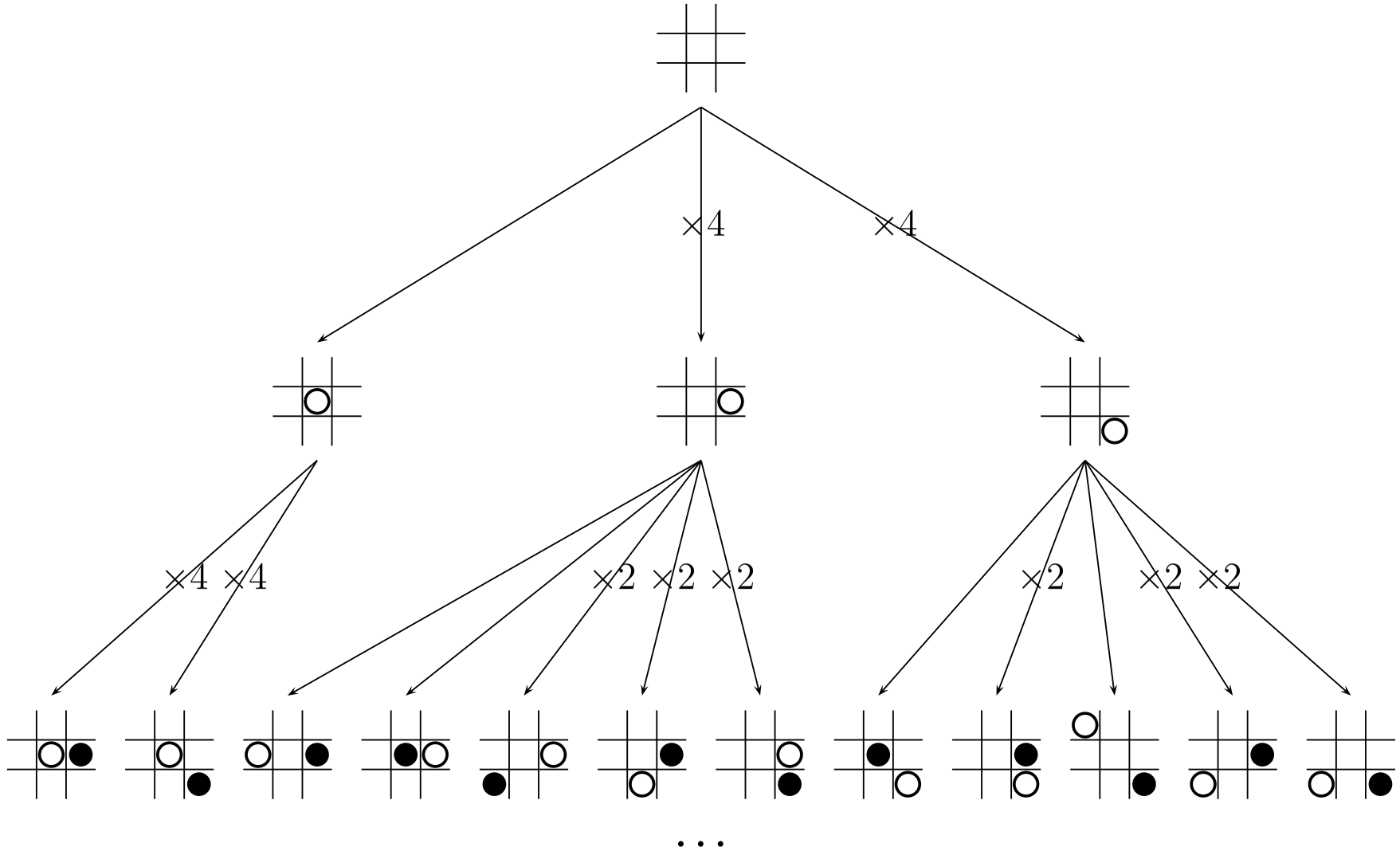
Alain Brobecker - *2024/06*

Many perfect information 1-player or 2-players games can be represented as a **game tree** (made of root, nodes and connections).



Note: in 2 players games a move is often considered as 1st player's movement and 2nd player's answer when a ply (or single move) is only one movement of current player.

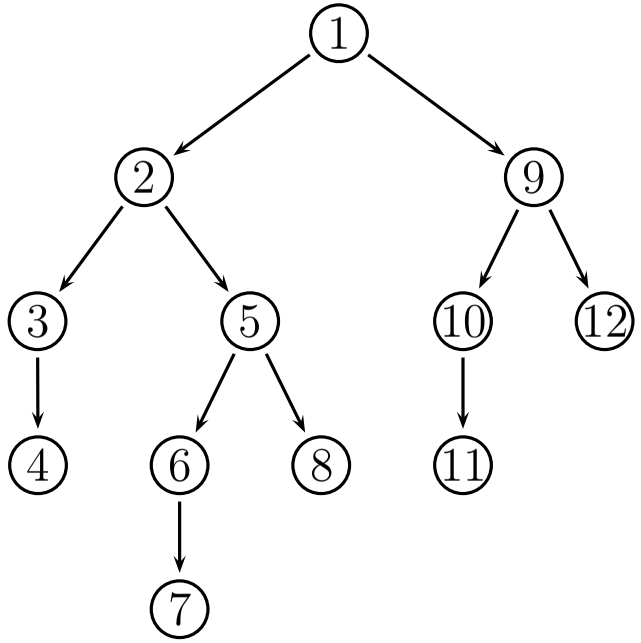
When exploring a game tree we can take into account any considerations allowing to reduce the quantity of data to process. Here the same tree with rotated and mirrored positions merged.



765 different positions (total nodes)

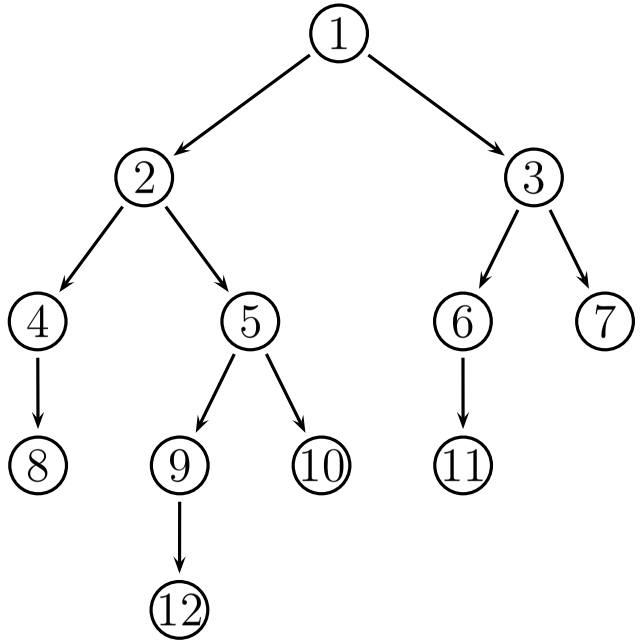
Two ways of exploring a game tree.

Depth first search (DFS)



Usually recursive: data is saved in stack with backtracking. Sometimes we save the move and use do_move() / undo_move() instead of saving the whole position.
Efficient to explore the whole tree.

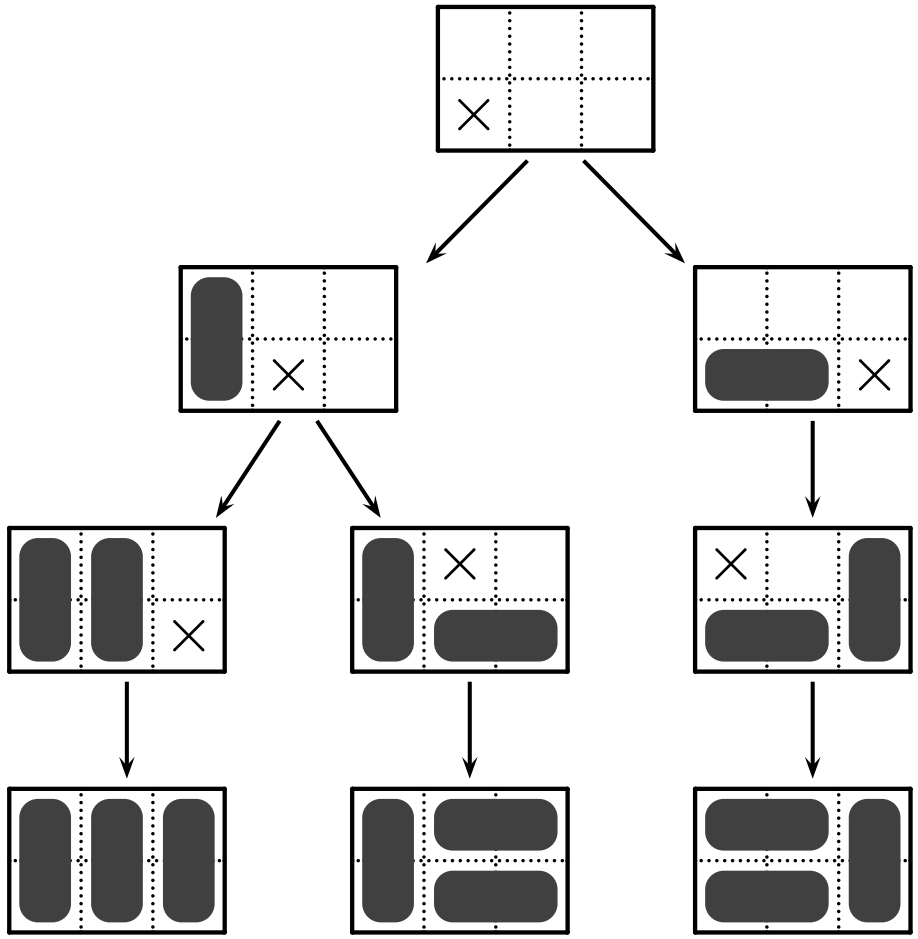
Breadth first search (BFS)



Usually needs more memory and to organise it, slower.
Best if we search for the shortest solution, as we can stop searching as soon as a solution is found.

More exploration methods exists: limited depth DFS, sampling, retrograde analysis, $\alpha\beta$ pruning...

Searching **all packings** of pieces by exploring a game tree using **depth first search**.

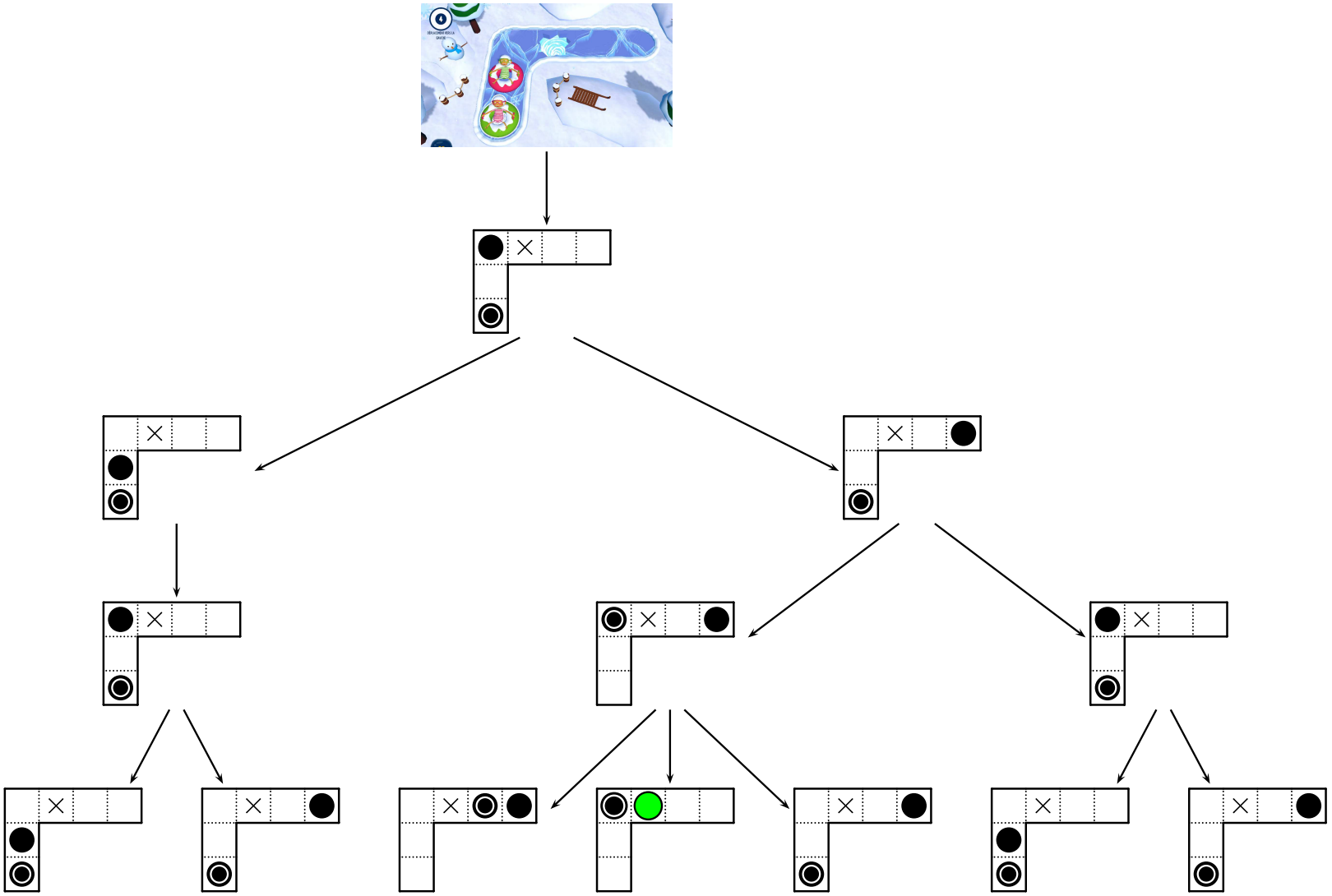


De Bruijn's Algorithm:

```
function DeBruijn_PutOnePiece(depth)
 | search position of first empty space
 | for all pieces still to place
 |   | mark this piece as already placed
 |   | for every orientation of that piece
 |   |   | try to place the piece
 |   |   | if piece correctly placed
 |   |   |   | if  board still has empty spaces
 |   |   |   |   | call DeBruijn_PutOnePiece(depth+1)
 |   |   |   | else
 |   |   |   |   | packing found, save it
 |   |   |   | endif
 |   |   | endif
 |   | next orientation
 |   | unmark this piece as already placed
 | next piece
end function
```

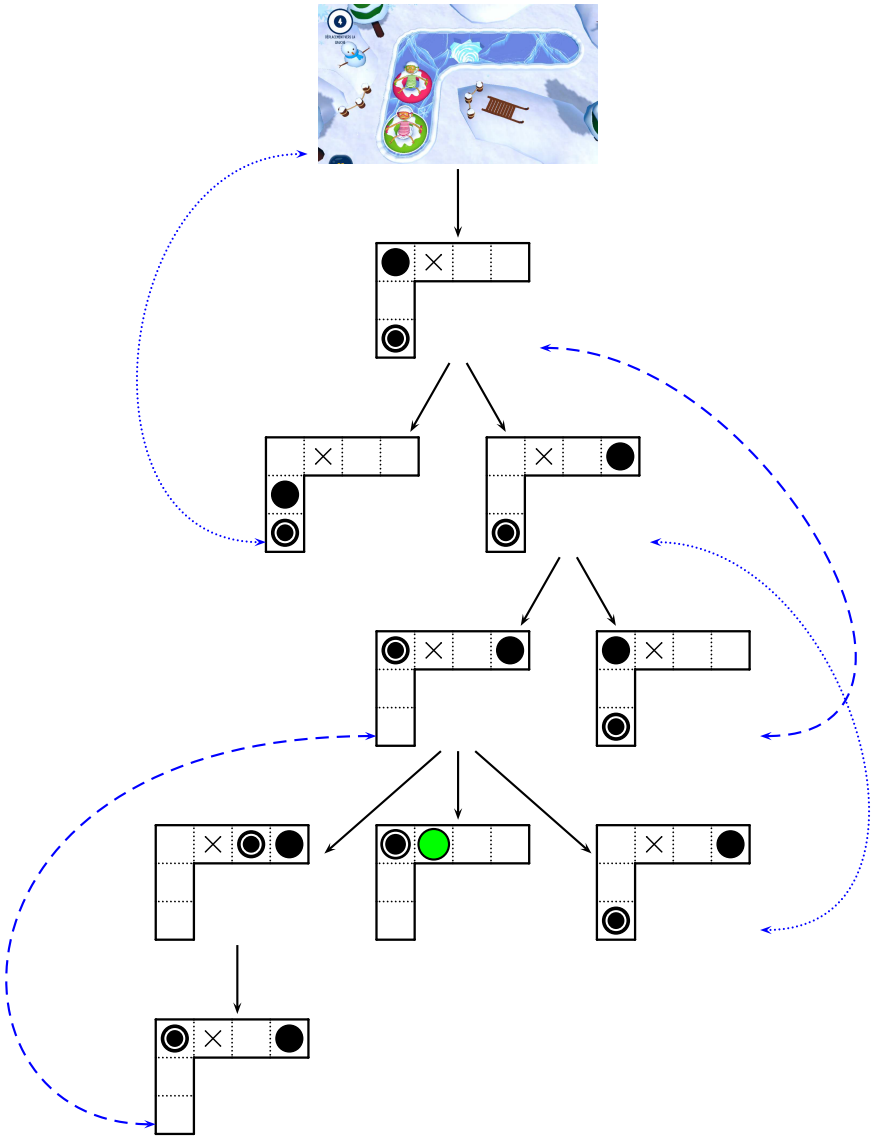
Cameron Browne pointed out that sometimes bit boards can greatly speed things up.

Finding shortest solution(s) of a sequential movement puzzle by exploring the game tree using **breadth first search**.



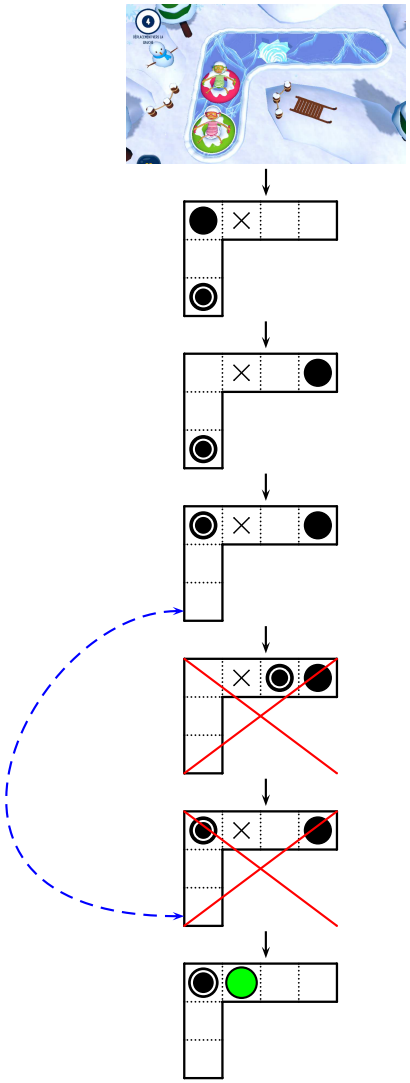
Game is Slidings (2010) reworked by Smart as Bump'in (2022)

In the previous game tree, some solutions were repeated.
We can use **transposition tables** to remember already seen positions.



When the position is seen again, we don't explore anymore that branch but just take all relevant information from position stored in the transposition table (eg: the evaluation of the position in multiplayers games, nothing if we search for shortest solution).

If player creates his own level and provides a (linear) solution,
we can use transposition tables to easily **remove loops**.



This of course does not guarantee to have the shortest solution,
but it can remove some useless moves made while the player was searching for a solution.

A **state** is all relevant informations at a precise moment of the game.

DreamKitten_Level_42_18_14_29_4_2024.json *(43 Kilobytes)*


```
{
  "FileName":"Level_42_18_14_29_4_2024",
  "Game":0,
  "Difficulty":0,
  "Grid":{"x":4,"y":4},
  "ActorList":[
    {"ProfileName":"DK_kitten","Type":"Player","Data":[{"Tile":{"Index":4},"Variation":0,"Rotation":0}]},
    {"ProfileName":"DK_mouse","Type":"Sheep","Data":[{"Tile":{"Index":6},"Variation":0,"Rotation":2}]},
    {"ProfileName":"DK_tileGoalStarter","Type":"Goal","Data":[{"Tile":{"Index":7},"Variation":0,"Rotation":2}]}
  ],
  "ActorListHalfGrid":[
    {"ProfileName":"DK_fence starter","Type":"Fence","Data":[
      {"Tile":{"Index":7},"Variation":0,"Rotation":0},
      {"Tile":{"Index":5},"Variation":0,"Rotation":1},
      {"Tile":{"Index":1},"Variation":0,"Rotation":1}
    ]}
  ],
  "DisabledTiles":[],
  "SaveDate":{"Year":2024,"Month":4,"Day":29,"Hour":14,"Minute":19,"Second":8},
  "IntVariables":[],
  "FloatVariables":[],
  "LevelIcon":[
    255,216,255,224,0,16,74,70,73,70,0,1,1,0,0,1,0,1,0,0,
    255,219,0,67,0,8,6,6,7,6,5,8,7,7,7,9,9,8,10,12,
    ...,217
  ]
}
```

A **state** is all relevant informations at a precise moment of the game.

- A state will usually contain the board & positions of pieces on the board, sometimes mention of special abilities depending on **the past of the game** (eg: in chess, castling or en passant capture possibilities, 50 moves rule as in Forsyth-Edwards Notation, when taking into account threefold repetition needs to know all moves played), and maybe extra informations...

- I advise that **similar states are always exactly identical**. As an example {"piece", 59, "board", -1, -1, -1, 0, 1, -1} and {"board", -1, -1, -1, 0, 1, -1, "piece", 59} while representing the same state are organised differently and would need extra work to check that they are similar, when if organisation is always the same (eg: sort tags alphabetically) a simpler byte by byte comparison is enough to track similar boardstates.

- The **number of moves** usually will be given aside the state, because in some games it is relevant (eg: Bump'in in which the number of moves to reach the goal is limited) while in many others it is not (eg: Chess).

- To speed up search in transposition tables, we can use a **hash function** that will give a short ID number to each position. A good hash function will have very few collisions.

A simple hash function made in 1999 for SockZ

```
b%=ASC(MID$(Name$,1,1))                          |       add  r0,r0,r0,lsl #8 ;ARM assembly version
c%=ASC(MID$(a$,LEN(Name$),1))                    |       eor  r0,r1,r0,lsr #4
d%=( (((b%<<8)+b%) EOR (c%<<4)) >>4) AND &ff     |       and  r0,r0,#&ff
```