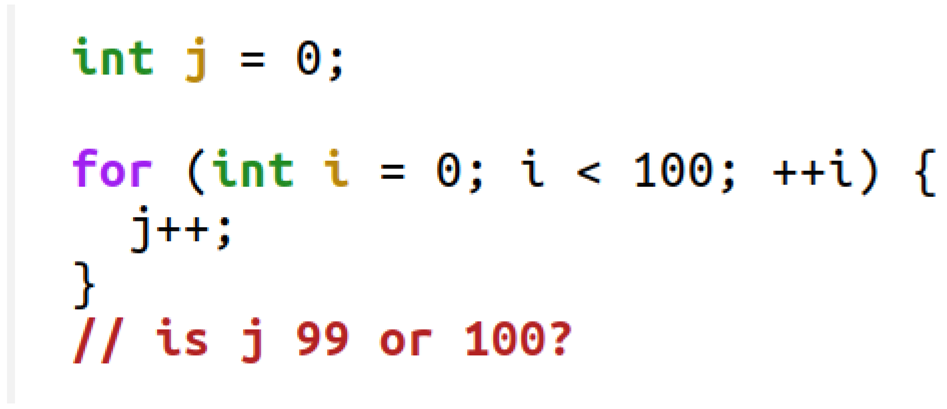
CS 4414: Recitation 6

Sagar Jha

Small mistakes in recitations ☹

- Tripped up on a simple question about loop unrolling code
- Mistakenly said that ASCII of ‘A’ is 97 when it’s 65. ASCII of ‘a’ is, in fact, 97
- Was confused about the following code:



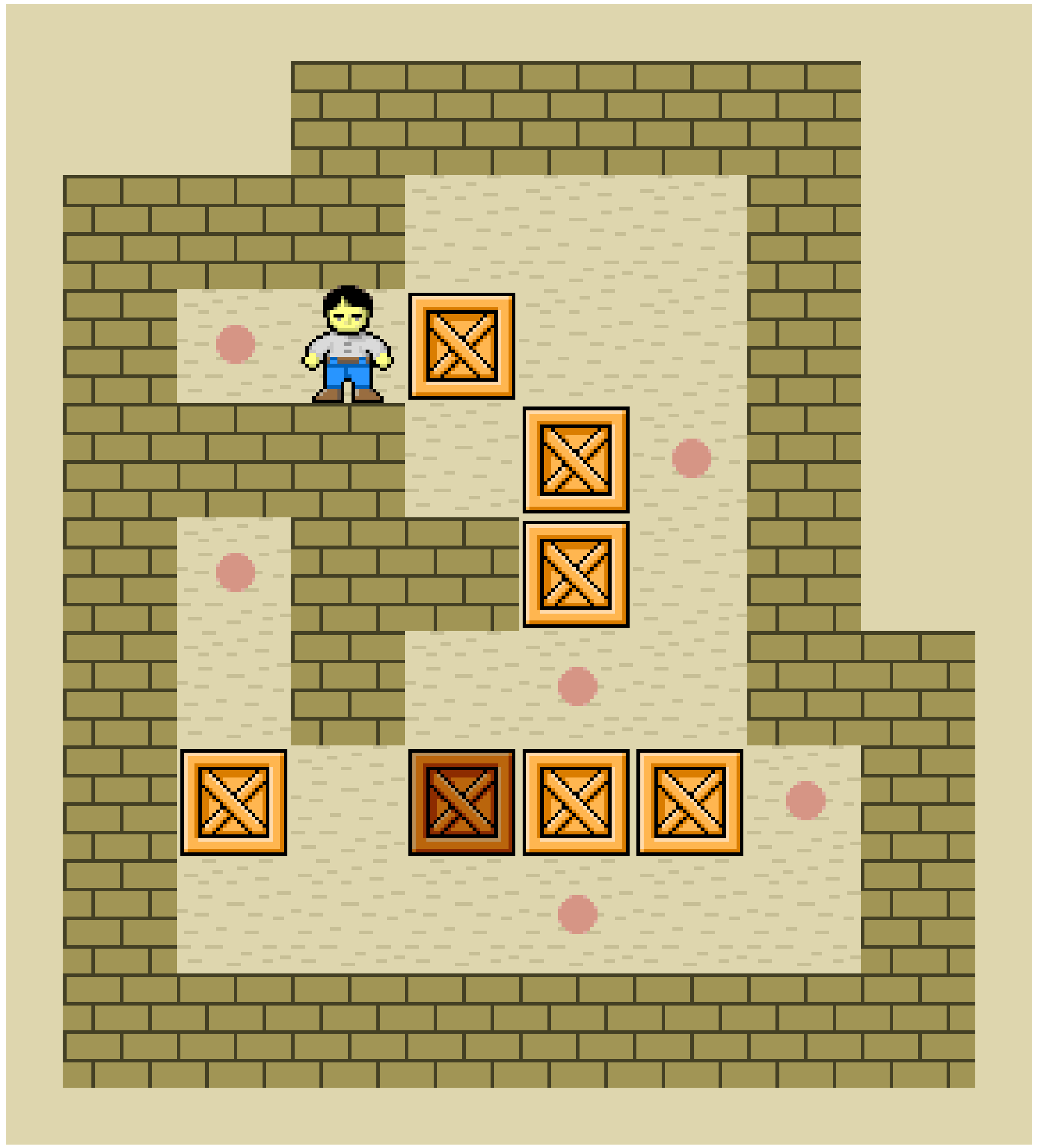
```
int j = 0;

for (int i = 0; i < 100; ++i) {
  j++;
}
// is j 99 or 100?
```

- If you are confused about something I say, feel free to clarify even if it’s minor!

Sokoban game

- The board consists of squares, each square is either a floor or a wall
- Some floors contain boxes. An equal number are marked as goals (or storage locations)
- The aim is to move all the boxes to the goals
- Each box can only be pushed by the player to an empty floor
- According to Wikipedia, computationally solving a Sokoban puzzle is NP-hard and PSPACE-complete



My background with Sokoban

- Introduced to the game by my mom when I was in my 4th year of undergrad (late 2014)
- She used to play a version of the game on her phone. She could only solve about the first 7-8 levels out of a total of 15
- She asked me for help. The solutions were not obvious to me (and I didn’t want to think hard).
- I decided to write a C++ solver for the game

My background with Sokoban

- I wrote the first version in a couple of hours (v1)
- Then successively optimized it in versions v2, v3 and v4 in the next few days
- v3 and v4 were able to solve all the 15 levels in reasonable time
- My mom practiced the solutions and memorized them

Today: Sokoban solver optimizations and performance analysis walkthrough

Our goal:

- Learn the basic computational approach to solving the game
- Survey the 15 levels (or test cases) to understand the evaluation criteria
- Appreciate the performance characteristics of the versions and the successive optimizations

Basic strategy: Partial game tree search

Board Configuration

- An intermediate Sokoban position

Game tree

- The starting configuration is the root of the tree
- There is a directed edge from configuration A to configuration B if the player can make one move from A to reach B

Version 1: Depth-first search

Pseudo-code:

Function search(current_position)

1. If aim not complete and if possible, move **DOWN**. Recursively continue search from the resulting position.
2. If aim not complete and if possible, move **UP** and recursively continue search from the resulting position.
3. Similar to step 2 but try moving **RIGHT**
4. Similar to step 4 but try moving **LEFT**

Maintain a set of visited configurations to avoid duplication. Extra bookkeeping for traceability.

Version 1 is too slow!

- The program gets lost at high depths trying to find a solution
- Does not consider alternative moves and just goes deep with a given move
- Solves level 1 immediately, takes about 80 seconds for level 4. Times out (> 5 minutes) for all other levels
- Can probably solve some levels within a couple of hours. Solving some “computationally” harder levels might take years!
- A Sokoban puzzle contains just one solution at a (relatively) high depth. Breadth-first search for such a case is more appropriate.
- In my experience, a Sokoban puzzle involves moving boxes in tandem with each other, creating space and making progress.

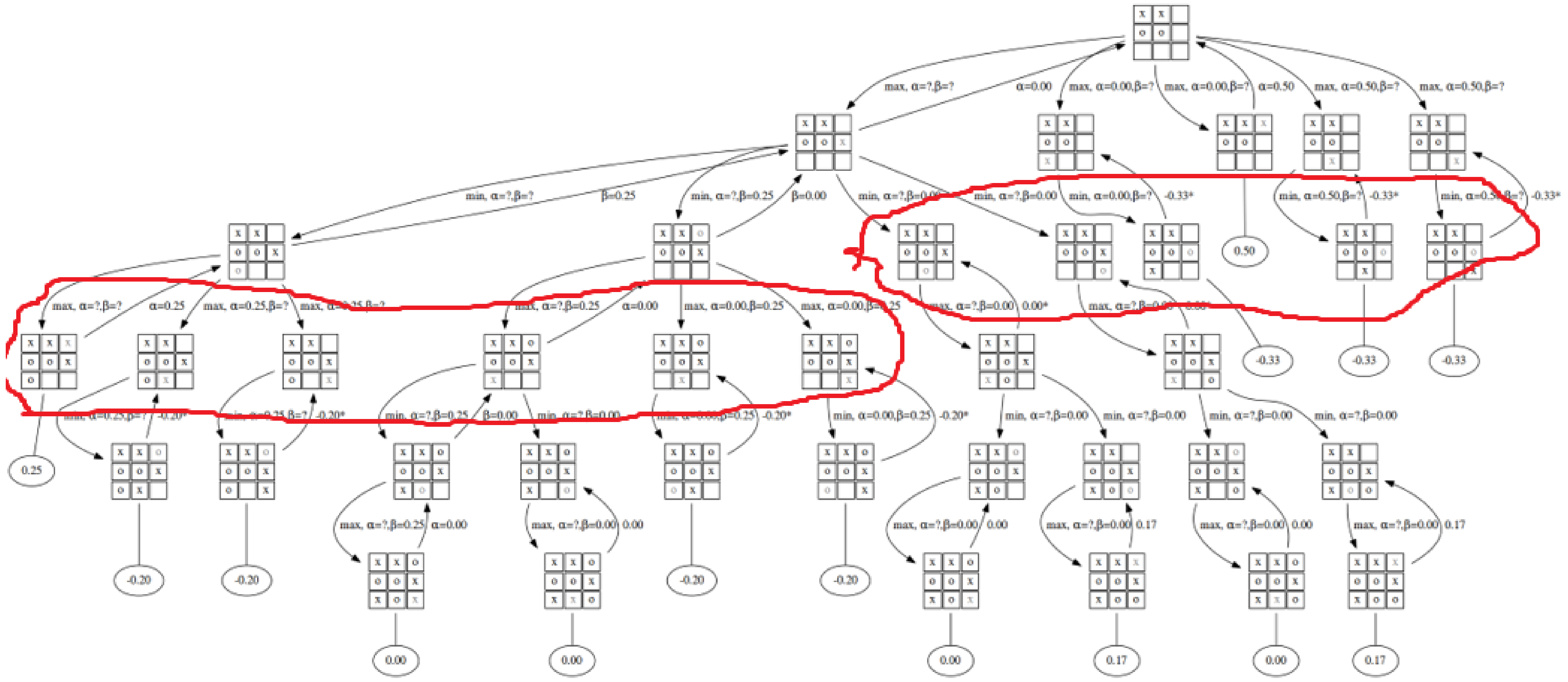
Version 2: Breadth-first search

Pseudo-code:

```
queue<config> positions;
while positions is not empty:
    1. Set current_position to the front of positions and remove this front element
    2. if target reached, break
    3. if possible, move DOWN and insert the resulting position to the end of positions
    4. Repeat step 2 but move UP
    5. Repeat step 2 but move RIGHT
    6. Repeat step 2 but move LEFT
```

Again, maintain a set of visited configurations to avoid duplication. Extra bookkeeping for traceability.

Version 2 takes too much memory!



A representative game tree for tic-tac-toe. Source: Wikimedia Commons

Version 2 takes too much memory

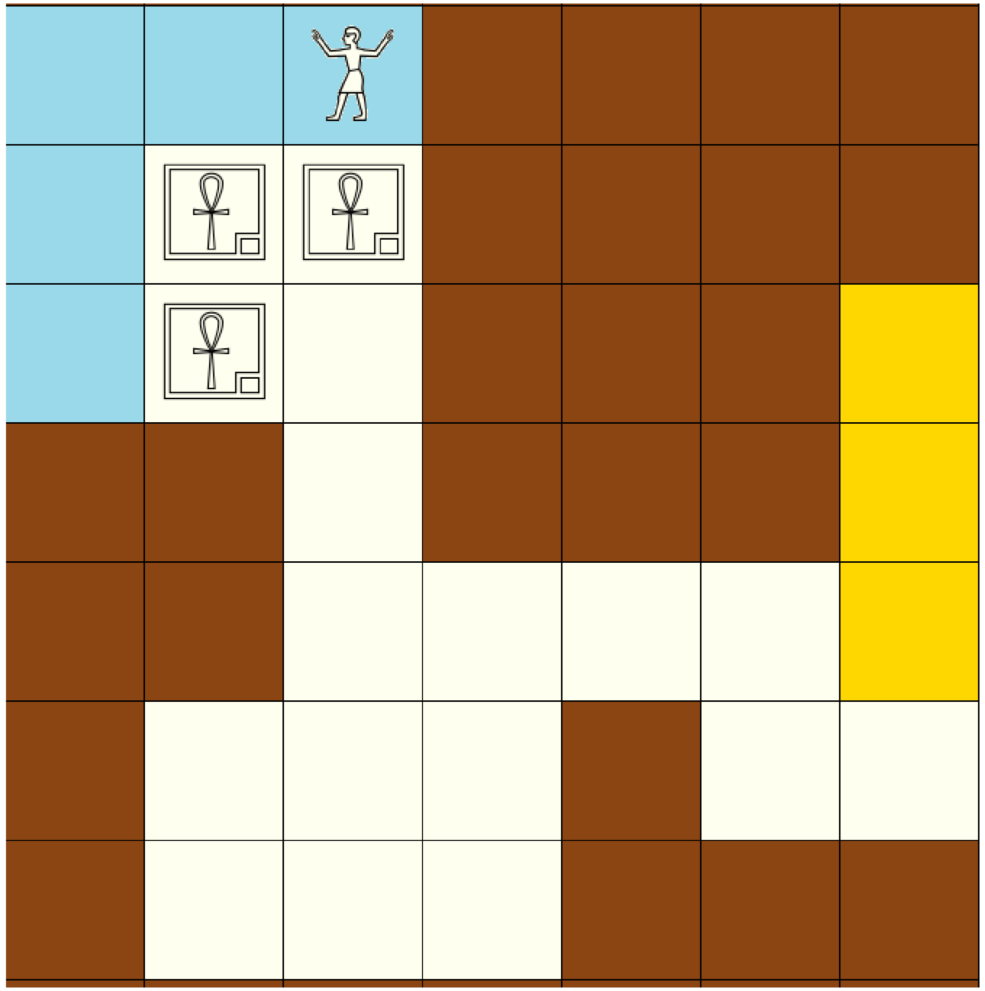
- Solves the first level successfully, runs out of memory for the rest on my laptop with 16 GB RAM. Back then I had only 4 GB RAM!
- Killed by Linux's OOM Killer (Out of Memory Killer), a process in Linux that runs when the system is critically low on memory
- Why does it not solve the initial few levels? Turns out to be a bug!
- I forgot to populate set of visited configurations!
- Even after the bugfix, memory usage remains high for more complex levels
- The number of positions at a given depth increases exponentially!

Depth-first search vs breadth-first search

- If the game tree is very deep and solutions are rare, DFS might take an extremely long time
- If the tree is very wide (large fanout), BFS might require a lot of memory, to the point of being impractical
- If there are many solutions but are located deep in the tree, BFS could again be impractical
- Aside: Iterative deepening depth-first search – Combines DFS’s space-efficiency and BFS’s completeness
- Source: https://stackoverflow.com/questions/3332947/when-is-it-practical-to-use-depth-first-search-dfs-vs-breadth-first-search-bf
- Also read: https://stackoverflow.com/questions/20429310/why-is-depth-first-search-claimed-to-be-space-efficient

Main optimization idea

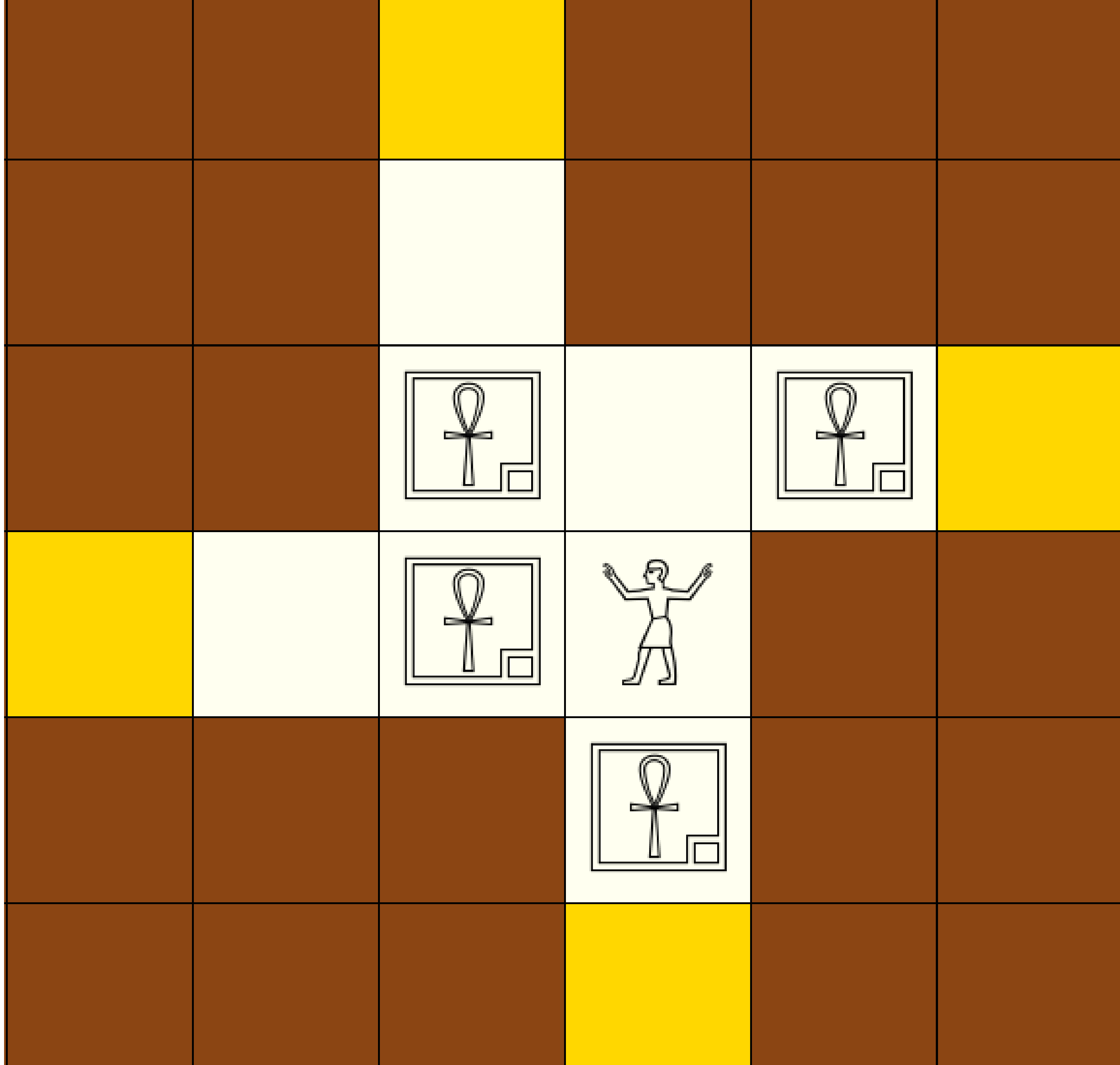
- We don’t need to track all the player moves
- Only configurations in which the boxes are at different positions are truly different!
- In level 2 on the right, it doesn’t matter which of the blue-marked floors the player is on
- Consider only sequences of player moves that end in one box moved



Detour: A survey of the levels

Level 1 is the simplest. There is always only one way to move boxes.

Sokoban visualizer credit: https://www.cs.rochester.edu/u/kautz/Sokoban/Sokoban.html

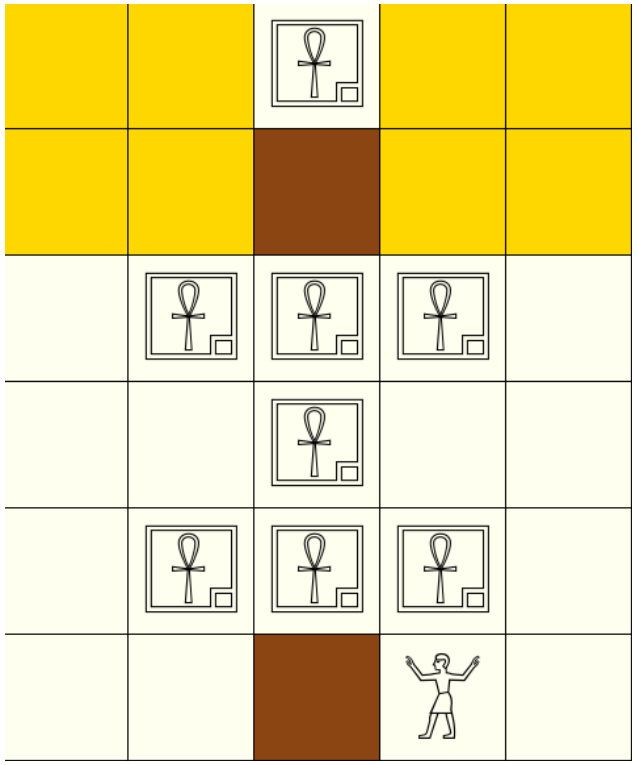


A survey of the levels

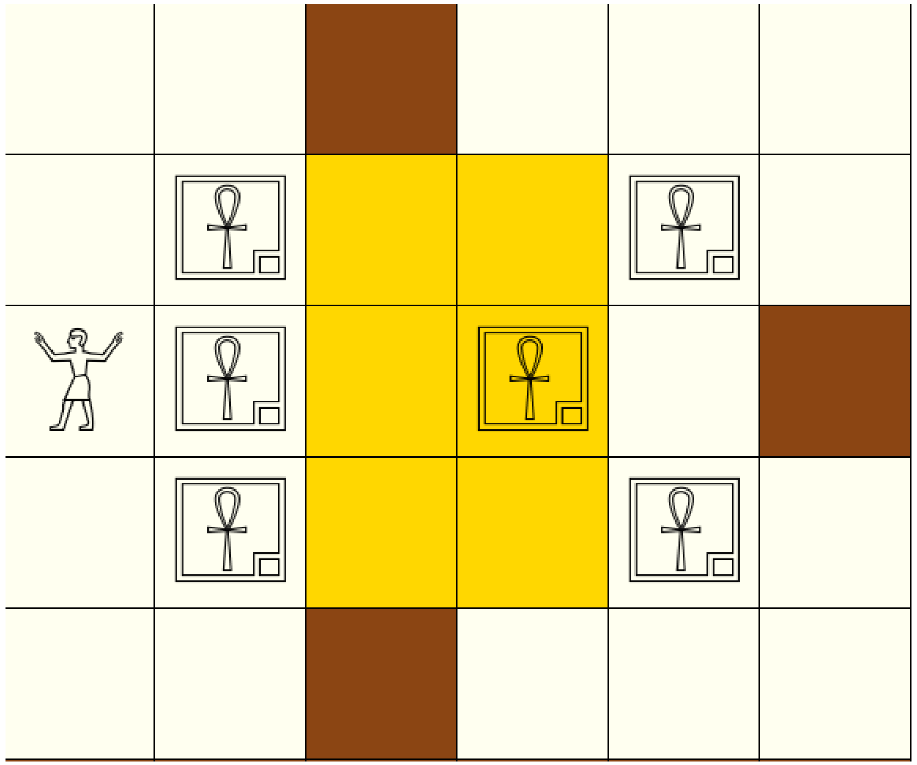
Number of boxes	Levels	Minimum player moves	Levels	Minimum box moves	Levels
3	1, 2, 4, 15	< 15	1	< 10	1, 6
4	6, 7, 9, 11	30-40	3, 5, 6	10-20	3, 4, 5, 7, 8, 9, 13, 15
5	3, 5, 12, 13, 14	50-75	4, 7, 8, 9, 10, 13	23-25	10, 12
6	8	90-105	2, 12, 15	31	2
8	10	190-200	11, 14	42	11, 14

1. Most likely, higher levels are "intellectually" more challenging
2. Minimum box moves is much lower than minimum player moves (computed using v4). Considering only box moves significantly reduces the search depth.
3. "Computationally", the higher the number of boxes, the longer it takes
4. Thus levels 8 and 10 are the most notorious!

Notable levels

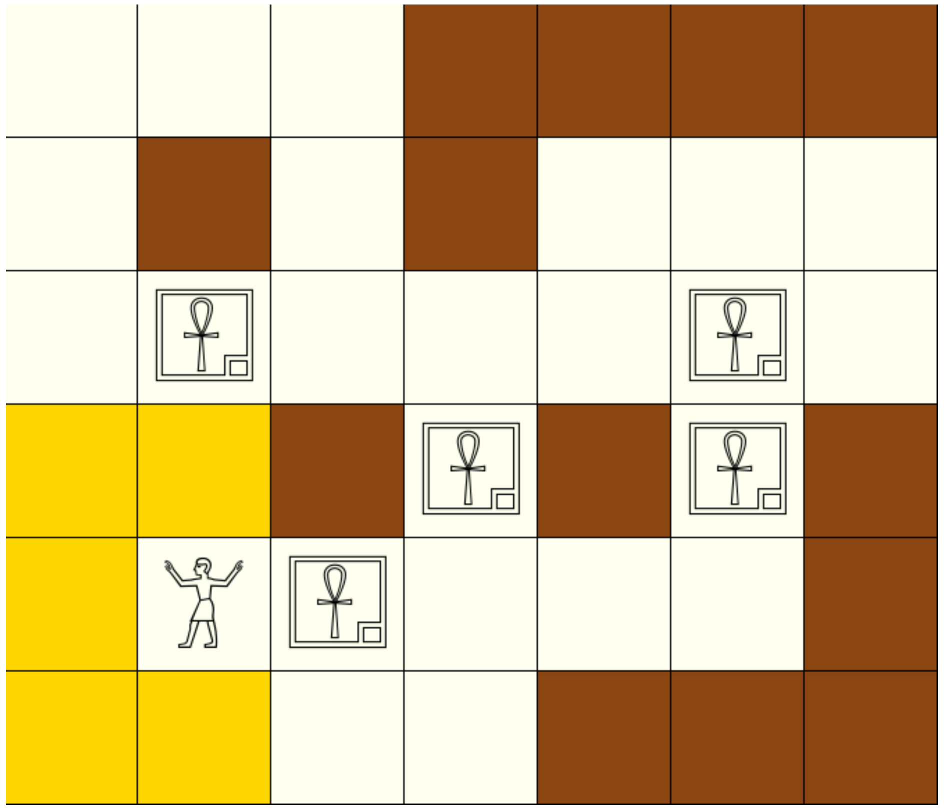


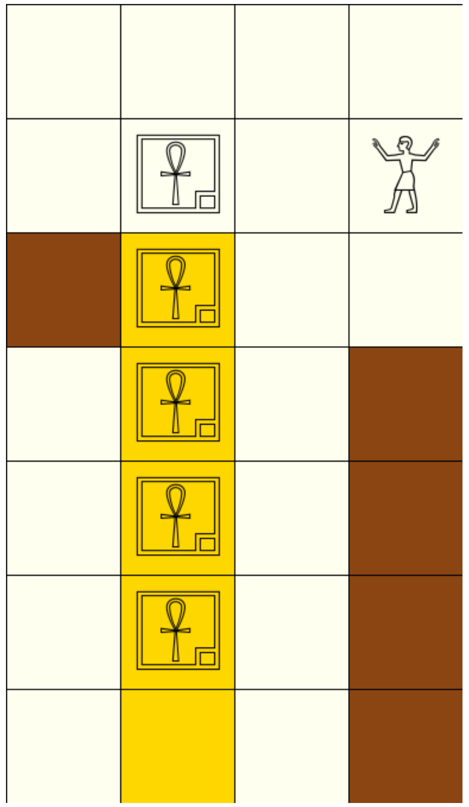
Level 10
Largest number of boxes (8)
Fairly large solution depth (25)



Level 8
Second largest number of boxes (6)
Medium solution depth (15)

Notable levels





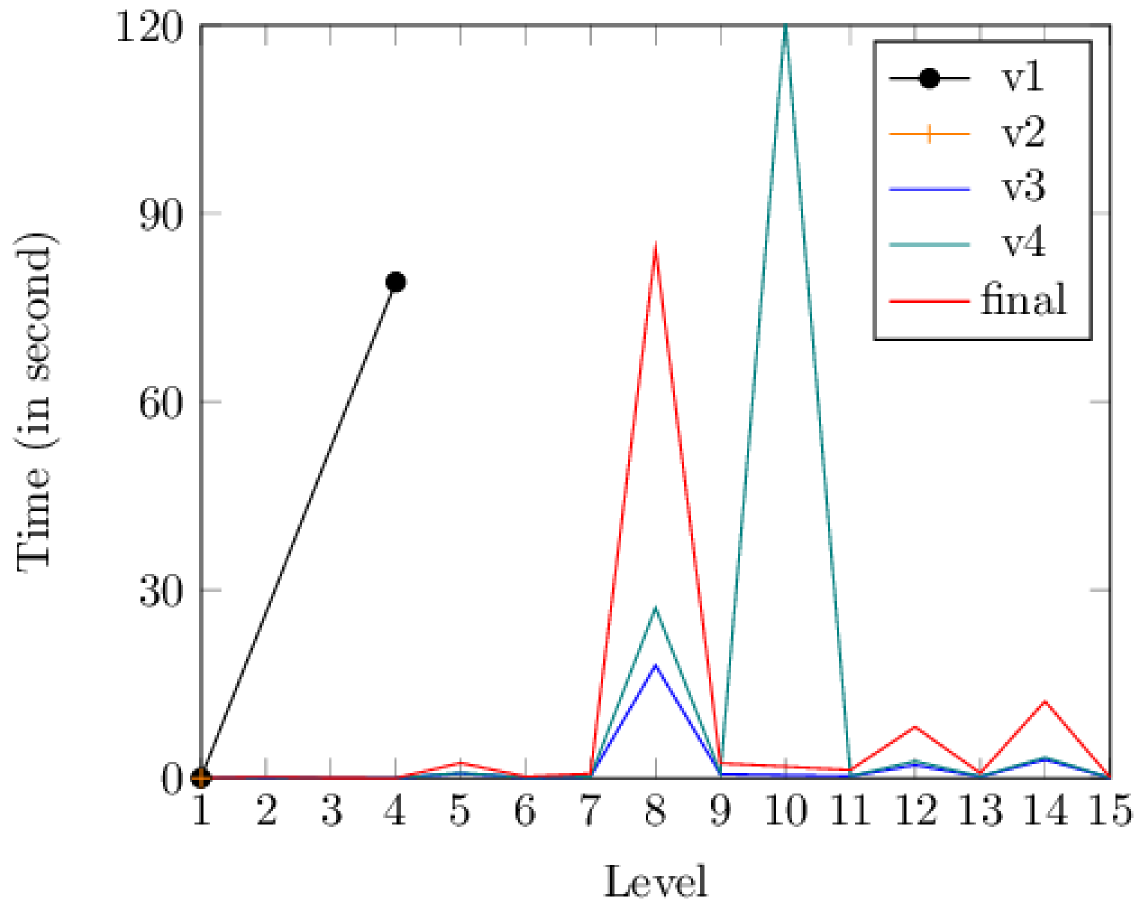
Level 14
Fairly large number of boxes (5)
Highest solution depth (42)

Level 12
Fairly large number of boxes (5)
Fairly large solution depth (23)

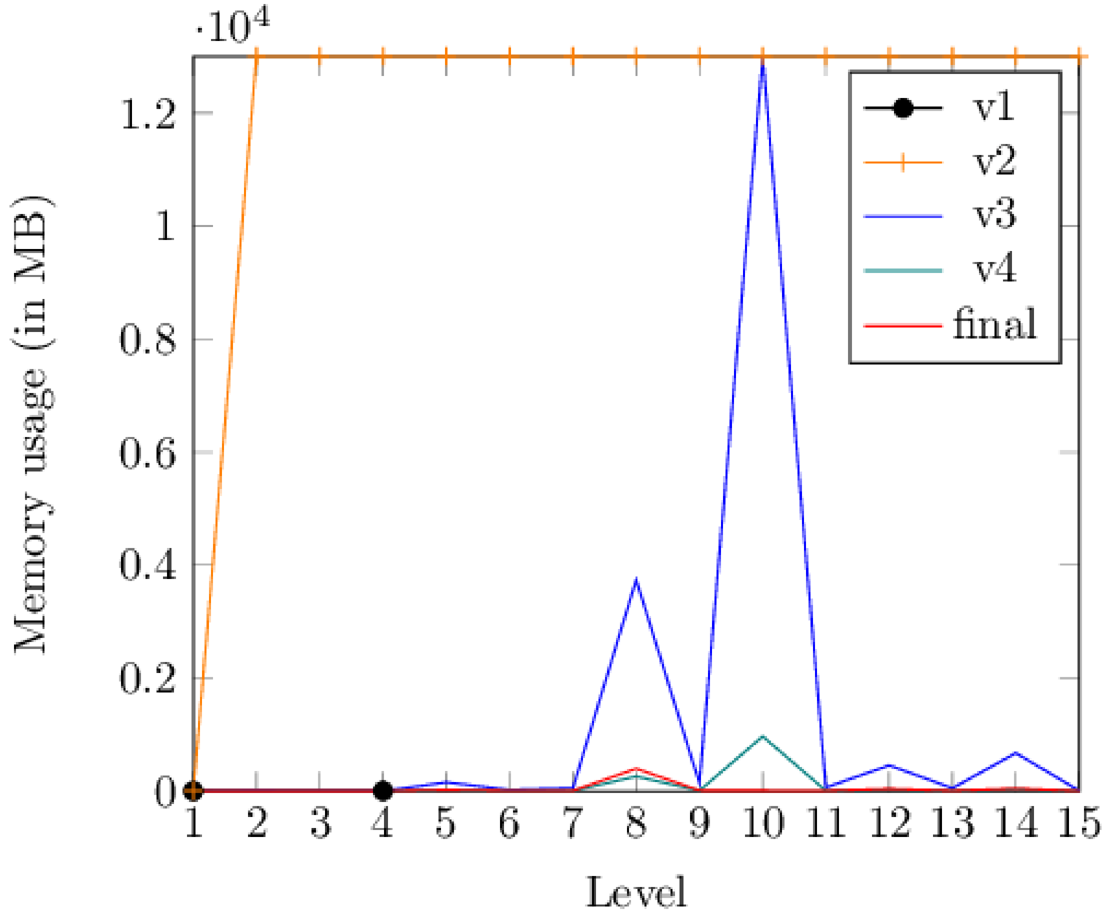
Considering only box moves: v3 and v4

- BFS > DFS established for our setting
- Both use breadth-first search considering only box moves
- They differ in how they find the number of possible one box move from a position
- v3 uses DFS, v4 uses BFS on player moves to find all one box moves
- Surprisingly v3 used a lot of memory, which is why I wrote v4
- v3 runs faster than v4, but results in a higher number of player moves (still the name number of box moves)
- A final version on top of v4 improves traceability by tracking all box moves instead of only tracking player moves. Uses vector<vector<char>> instead of vector<char>

v3 vs. v4 vs. final



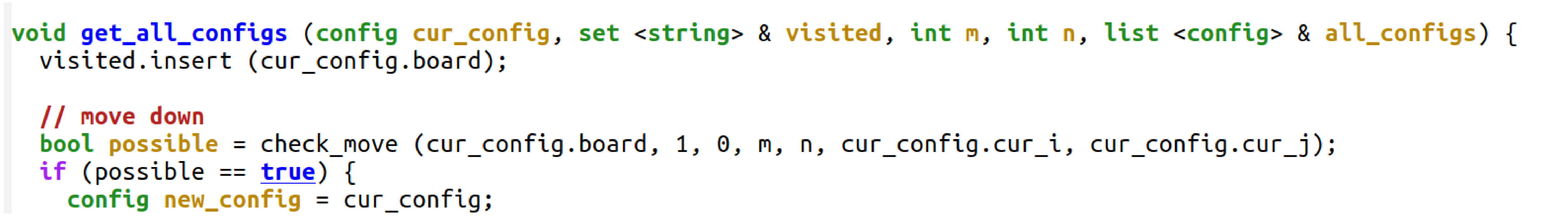
Time taken to solve vs. level



Memory usage vs. level

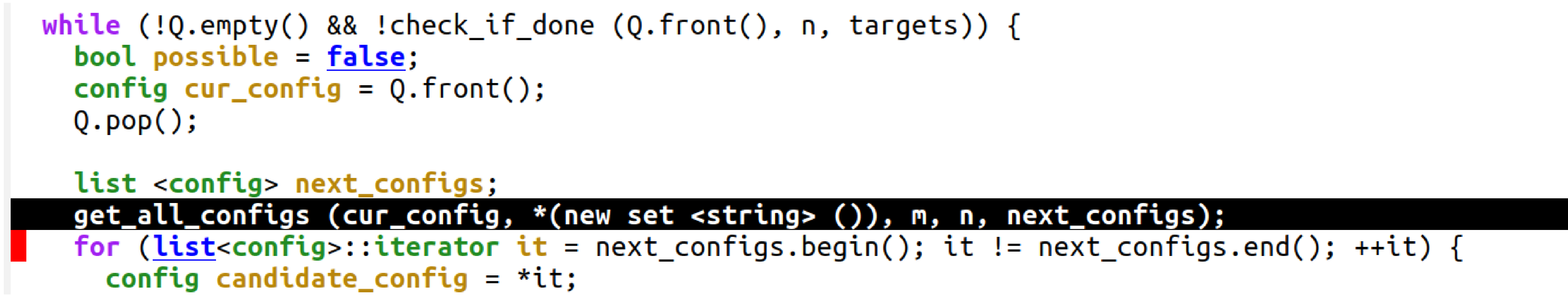
Why does v3 take so much memory?

Turns out to be a bug – memory leak!



```
void get_all_configs (config cur_config, set <string> & visited, int m, int n, list <config> & all_configs) {
  visited.insert (cur_config.board);

  // move down
  bool possible = check_move (cur_config.board, 1, 0, m, n, cur_config.cur_i, cur_config.cur_j);
  if (possible == true) {
    config new_config = cur_config;
```



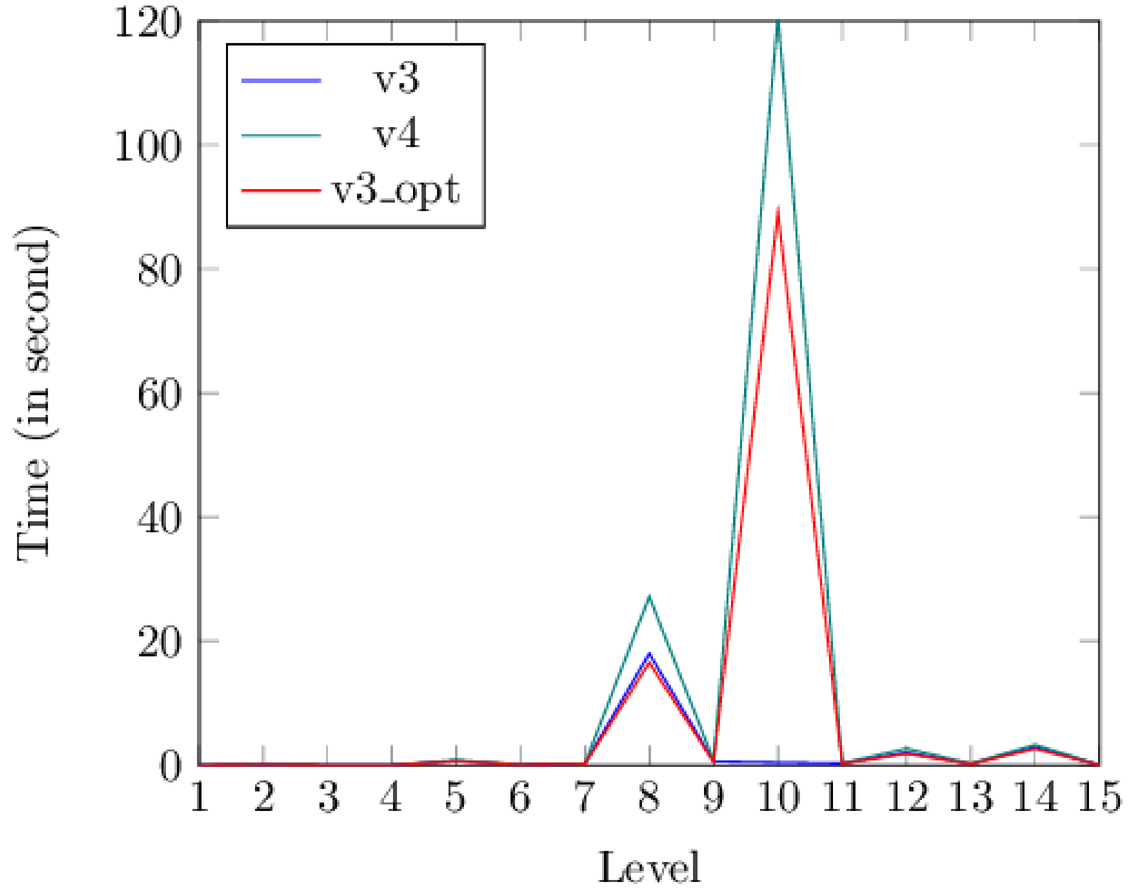
```
  while (!Q.empty() && !check_if_done (Q.front(), n, targets)) {
    bool possible = false;
    config cur_config = Q.front();
    Q.pop();

    list <config> next_configs;
    get_all_configs (cur_config, *(new set <string> ()), m, n, next_configs);
    for (list<config>::iterator it = next_configs.begin(); it != next_configs.end(); ++it) {
      config candidate_config = *it;
```

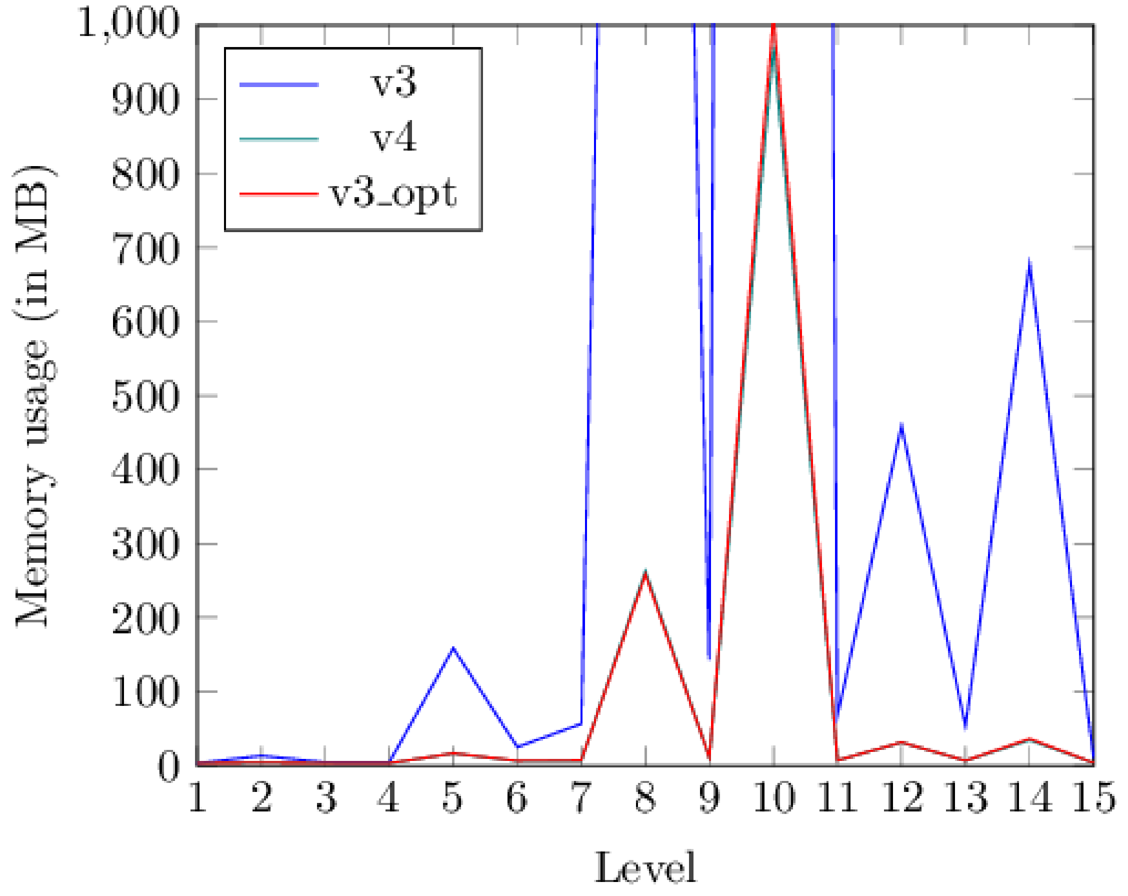
v3 vs v4 vs final

- Why does v3 take less time?
  - One possibility is that DFS using recursion is faster than maintaining a queue for searching single box moves
  - gprof analysis reveals that v4 spends considerable time in vector reallocations!
- v3 produces larger traces
  - A solution for level 10 is 138 player moves total, compared to only 66 for v4
  - The number of box moves is still optimal
- Why is final slower?
  - By using a vector<vector<char>> for better traceability, final spends a lot of time doing vector construction, destruction and reallocation as per gprof
  - Could be optimized further

v3 after the bugfix gives the best results



Time taken to solve vs. level



Memory usage vs. level

Further optimization opportunities

- Remove dead states
    - A box pushed to a corner of the board such that it cannot be pushed anymore results in a dead state (if that corner square is not a goal). There is no point exploring dead states further (which means exploring moving other boxes)
    - Interesting question: How many dead states are explored by v3_opt for level 10?
    - Similarly, a box may be constrained to be along an edge of the board. If there is no goal along the edge, we have a dead state
- Reachability analysis for finding one box moves
    - No need to do a DFS or a BFS
    - A square is reachable from a starting position if there is a path of player moves that move the player to this square (without moving any boxes)
    - Similar to BFS, but uses only one board configuration to calculate reachability

Further optimization opportunities

- Modernize code
  - My code is written in plain C++. Use C++-17
  - Current versions make sure to pass variables by references, but might be copying strings

- Next time
  - I will show a C++-17 Sokoban solver with more optimizations
  - We will look at its performance characteristics
  - We will learn and debug using gdb artificial or natural bugs