Complete Search of Sliding Tile Puzzles on a Personal Computer [Extended Abstract]

Oleg Tarakanov
Zenseact AB, Gothenburg, Sweden
light_ln2@hotmail.com

Abstract

The 4×4 and 8×2 Sliding Tile Puzzles have more than ten trillion solvable states, making complete brute force search very challenging: best existing solutions take weeks to run or require expensive hardware. We propose and implement a set of optimizations of the frontier search algorithm, that are efficient on a modern personal computer. We run a number of complete searches, each taking about 3 days on our hardware, for both puzzles in single-tile and multi-tile metrics, verifying previously known results about the radiuses of the puzzles. We also discover that the diameter of the 4×4 Puzzle is 80 single-tile moves, and the radius of the 8×2 Puzzle is 57 multi-tile moves.

Introduction

The famous "Fifteen" Sliding Tile Puzzle consists of a 4×4 rectangular frame with fifteen square tiles labeled from one to 15 and one "blank" position. In normal single-tile metric, the legal moves are to slide a tile up, down, left or right into that blank position. In multi-tile metric, one or more consequent moves in the same direction are considered as one move. The goal of the puzzle is to reach the initial state (which has all the tiles sorted numerically and the blank in the top-left corner) from a given state.

In 1999, Brungger proved that the radius of the 4×4 Puzzle is 80 (Brungger et al. 1999). In 2005 - 2008, R. Korf introduced the Disk-Based Frontier Search algorithm (DBFS), and performed complete searches of 4×4 and 8×2 puzzles (in single-tile metric), confirming Brungger's result and determining the radius of the 8×2 puzzle to be 140 moves (Korf 2005; Korf and Schultze 2005; Korf 2008). In 2010, B. Norskog reported that the radius of the 4×4 Puzzle in multi-tile metric is 43 multi-tile moves (unpublished).

We propose a number of optimizations of DBFS that improve its running time from weeks to days on a modern personal computer: choosing perfect hash function that makes calculation of horizontal moves very cheap; postponing expansion of horizontal moves to reduce disk I/O; processing vertical moves on GPU; compressing data stored on disk.

We implement them in a program which is written in C# language, and uses ILGPU library (www.ilgpu.net, written by Marcel Koester) for GPU calculations. Source code is freely available under MIT license at *github.com/lightln2/SlidingTilesSolver*.

Asymmetric Perfect Hash Function

A perfect hash function maps problem states into integer indexes. We represent a solvable state of a puzzle with dimensions $w \times h = S$ as a sequence $\{x_1, \dots, x_{S-1}; b\}$, where b is blank position and x_i are tile numbers as encountered in reading order (from top to bottom and from left to right, skipping the blank). We construct the hash function in the following way: first, we map the permutation of x_i to a number between 1 and $(S-1)!$, as described in (Korf 2008), then divide it by two (leaving only the integer part), multiply by S and add b. Division by two results in loss of one bit of information; when applying the inverse transform, we might get x_{S-2} and x_{S-1} in the wrong order. But, as only one of those states is reachable from the initial position, depending on the number of inversions of x_i (Johnson 1879), we can swap last two tiles if necessary, such that the resulting state is reachable. In practice, instead of multiplying by S, we multiply by 16, because dividing by a constant which is a power of two is much faster than dividing by a variable.

As horizontal moves only change blank position, they can be calculated very fast, without restoring the state.

Delayed Node Expansion

The original DBFS splits problem space into segments, based on the hash function, and stores list of states at current depth, with their used-operator bits, in files, one file per each

segment. In each iteration, it runs two stages: first, it expands the states at current depth and writes expanded states to an intermediate set of disk files. Next, it does in-memory de-duplication of these expanded states to calculate states and used-operator bits at next depth (Korf, 2008).

Our optimized version only expands vertical moves at the first stage. As horizontal moves never change the segment, they are expanded on the fly at the de-duplication stage, reducing disk I/O.

GPU Acceleration

When expanding vertical moves, the state indexes are collected in memory in large batches and sent to a GPU card. The card runs the CUDA kernel that applies the moves using thousands of GPU threads, which is more than 10 times faster than on CPU, and sends the resulting indexes back to the main memory. The CPU thread waits for the results, so, given enough threads, it adds no additional time to the algorithm run time.

Data Compression

The list of states is stored in files as 4-byte indexes in increasing order, so the differential encoding (storing differences between consequent indexes) combined with variable-length encoding can be used. We used Stream VByte encoding (Lemire, Kurz, and Rupp 2018). This encoding stores values in $1-4$ bytes depending on their size, plus adds overhead of two bits per value. In practice, the compression was close to theoretical optimum of 1.25 bytes per state.

Although the states' indexes in the intermediate disk files are not sorted, the same compression worked for them too, reducing 4 bytes to $2-3$ bytes per state.

Multi-Tile Metric

In multi-tile metric, there are exactly $w+h-2$ possible moves for each state, and horizontal and vertical moves alternate: if a state was arrived at via a vertical, respective horizontal move, it cannot be expanded vertically, respective horizontally. It means that states can be stored in two sets of files: those that can be expanded vertically or horizontally. These files are read at different stages of the iteration, reducing disk I/O.

In multi-tile metric, puzzles can have odd-length cycles (an odd number of states connected by moves forming a cycle). It requires removing current iteration's states at the de-duplication stage, increasing disk I/O. But we discovered that for puzzles of height 2, only states that can move horizontally should be removed, which reduces disk I/O.

Size	Radius	Run time	Disk I/O
4×4, Single-Tile	80	3 days	83TB
8×2, Single-Tile	140	2.6 days	72TB
4×4, Multi-Tile	43	3.5 days	86TB
8×2, Multi-Tile	57	1.8 days	48TB

Table 1: Complete Search of the Sliding-Tile Puzzles

Results

The program was run for all $w \times h \leq 16$ puzzles, for both single-tile and multi-tile metrics. Table 1 shows results for the fifteen puzzles. Last column "Disk I/O" shows total number of bytes read and written during the program run, ranging from 48 to 86 terabytes, $2.5-4$ times less than 200 TB which is estimated disk I/O of classic disk-based frontier search. Last line shows that the radius of the 8×2 puzzle is 57 multi-tile moves; this result is new, to the best of our knowledge.

We also used the program to find the diameter of the 4×4 Puzzle, that is, minimum number of steps that allow to reach any state from any other state. Due to symmetry, the diameter equals to the maximum of the radiuses from three initial states: with blank in the corner (the standard initial position), on the edge, and in the interior. To find it, we ran complete searches for the other two initial states and found all radiuses, and hence the diameter, to be 80 single-tile moves.

References

Brungger, A.; Marzetta, A.; Fukuda, K.; and Nievergelt, J. 1999. The parallel search bench ZRAM and its applications. *Annals of Operations Research* 90: 45–63. doi.org/10.1023/A:1018972901171.

Johnson, W. W.; and Story, W E. 1879. Notes on the "15" Puzzle. *American Journal of Mathematics* 2(4): 397–404. doi.org/10.2307/2369492.

Korf, R. E. Frontier search. 2005. *Journal of the ACM* 52(5): 715–748. doi.org/10.1145/1089023.1089024.

Korf, R. E.; and Schultze, P. 2005. Large-scale parallel breadth-first search. *In Proceedings of the 20th National Conference on Artificial Intelligence* 3: 1380–1385. dl.acm.org/doi/10.5555/1619499.1619555.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 55(6): 1–40. dl.acm.org/doi/10.1145/1455248.1455250.

Lemire, D.; Kurz, N.; and Rupp, C. 2018. Stream VByte: Faster Byte-Oriented Integer Compression. *Information Processing Letters* 130: 1–6. doi.org/10.1016/j.ipl.2017.09.011.