A Memory-Efficient Method for Fast Computation of Short 15-Puzzle Solutions

Ian Parberry

Technical Report LARC-2014-02

Laboratory for Recreational Computing
Department of Computer Science & Engineering
University of North Texas
Denton, Texas, USA

April 2014

A Memory-Efficient Method for Fast Computation of Short 15-Puzzle Solutions

Ian Parberry
Dept. of Computer Science & Engineering
University of North Texas
Denton, TX, 76203–5017
http://larc.unt.edu/ian

***Abstract*—While the 15-puzzle has a long and interesting history dating back to the 1870s, it still continues to appear as apps on mobile devices and as minigames inside larger video games. We demonstrate a method for solving the 15-puzzle using only 4.7MB of tables that on a million random instances was able to find solutions of 65 moves on average and 95 moves in the worst case in under a tenth of a millisecond per solution on current desktop computing hardware. These numbers compare favorably to the worst-case upper bound of 80 moves and to the greedy algorithm published in 1995, which required 118 moves on average and 195 moves in the worst case.**

***Index Terms*—15-puzzle, 8-puzzle, breadth-first search, directed graph, divide and conquer, greedy algorithm.**

I. Introduction

The 15-puzzle has fifteen numbered tiles arranged in row-major order on a 4×4 grid leaving a blank space in the last position (see Fig. 1, left). A single move of the puzzle involves sliding an adjacent tile horizontally or vertically into the blank space. The aim is to return a randomized puzzle (for example, Fig. 1, right) to the initial configuration making the smallest possible number of moves. The 15-puzzle can obviously be generalized to the (n^2-1)-puzzle, for all integers $n \geq 2$. The smaller and more manageable 8-puzzle is of particular interest.

The 15-puzzle has a long and interesting history (see, for example, Hordern [1]) that is said to date back to the 1870s. More recently, it has appeared in the form of various apps on mobile devices and as minigames inside larger games. For example, the 15-puzzle can be found in the original *Final Fantasy* (Square Enix, 1987) and *The Legend of Zelda: The Windwaker* (Nintendo 2003), and the 8-puzzle can be found in *Machinarium* (Amanita Design, 2009).

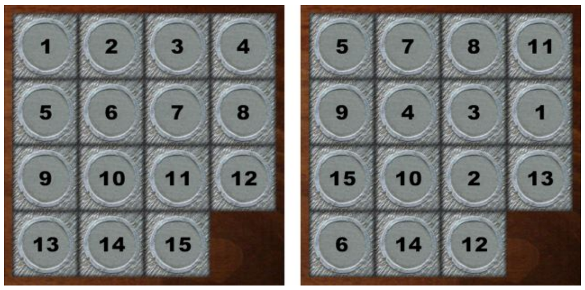

Fig. 1: The 15-puzzle in its solved configuration (left) and a random configuration (right).

A table of optimal moves for all 16! 15-puzzle configurations would take up 327GB (see Section IV), which is impractical for consumer use on current technology. We show how to compute 15-puzzle solutions of worst case length 93 and expected length less than 66 using only 4.7MB of tables, which is a practical memory requirement even on mobile platforms. The number of moves compares favorably to the worst-case upper bound of 80 moves from Brüngger et al. [2].

The remainder of this paper is divided into four sections. Section II sketches some prior work. Section III describes how to compute optimal solutions to the 8-puzzle. Section IV shows how to extend these results to the 15-puzzle by taking the advantage of the observation that most optimal 8-puzzle solutions appear to proceed in a very structured manner. Section V summarizes our results and gives open problems.

II. Prior Work

Reinefeld [3] showed that the 8-puzzle can be solved in at most 31 moves. Brüngger et al. [2] showed that the 15-puzzle can be solved in at most 80 moves. Ratner and Warmuth [4] have proved that the problem of finding the minimum number of moves for the (n^2-1)-puzzle is NP-hard, and they demonstrate that a polynomial time approximation algorithm exists. Kornhauser, Miller, and Spirakis [5] show an $O(n^3)$ time algorithm for the (n^2-1)-puzzle, which therefore uses $O(n^3)$ moves in the worst case.

Parberry [6] gave worst case upper and lower bounds of $5n^3$ and n^3 respectively on the number of moves required to solve the (n^2-1)-puzzle, and lower bounds of at least $2n^3/3$ and $0.264n^3$ respectively for the expected number of moves and the number of moves required for a random configuration with probability one. The upper bound of $5n^3$ moves was derived by using divide-and-conquer, first using a greedy algorithm to place the first row and column and then recursing on the rest of the puzzle down to the 3-puzzle, which is trivial to solve. We will refer to this *the greedy algorithm* for the rest of this paper. More recently, Parberry [7] showed both theoretically and experimentally that the greedy algorithm solves the (n^2-1)-puzzle in expected number of moves $\frac{8}{3}n^3 + O(n^2)$.

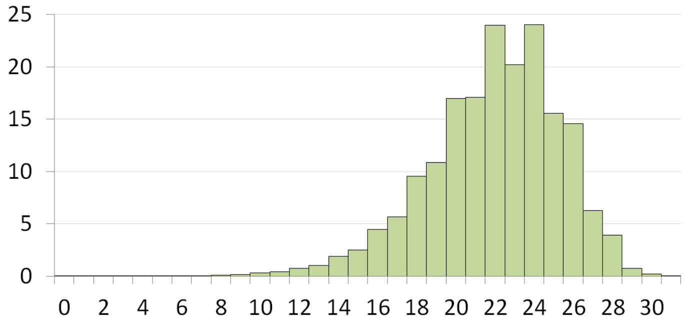

Fig. 2: The distribution of shortest solution lengths for the 8-puzzle. The horizontal axis shows the possible lengths of shortest paths, and the vertical axis shows the number of puzzle configurations that have a shortest path of that length, measured in thousands.

22.15	21.90	22.12
21.90	21.65	21.89
22.12	21.89	22.14

30	29	30
29	30	31
30	31	30

Fig. 3: The expected (left) and worst-case (right) solution lengths for the 8-puzzle when the blank is in each of the 9 possible initial positions.

III. The 8-Puzzle

In 1993 Reinefeld [3] showed that the 8-puzzle can be solved in no more than 31 moves. It is interesting to ask how much CPU time it would take to duplicate this result using current technology.

Define the *solution graph* of the 8-puzzle to be a directed graph with a vertex v_s for each configuration s, and an edge from vertex v_s to vertex v_t if there is a legal move from configuration s to configuration t. For example, consider the three configurations in Fig. 4. The configuration at left is the initial configuration. The center configuration is reachable from the initial configuration by sliding tile 6 down into the blank space (which can also be viewed as sliding the blank up). The right configuration is reachable from the initial configuration by sliding tile 8 to the right into the blank space (which can also be viewed as sliding the blank left).

Each of these configurations can be stored in row-major order in a 64-bit integer using eight hexadecimal digits, with tile i represented by digit $i-1$ and the blank represented by the digit 8, as shown below each of the configurations in Fig. 4. Fig. 5 shows part of the solution graph with nodes representing these three configurations, and edges labelled with “L”, “R”, “U”, and “D” when the edge represents a move of the blank one place left, right, up, and down, respectively.

A solution for all configurations of the 8-puzzle can be found by performing a breadth-first search (Moore [8], Lee [9]) on the solution graph starting at vertex

`0x012345678`.

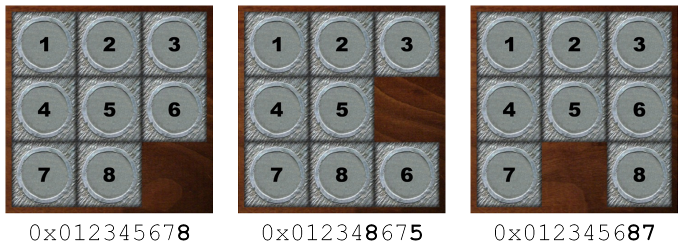

Fig. 4: Three configurations of the 8-puzzle with their representations as hexadecimal numbers.

$$\begin{tikzcd}
\boxed{12345678} \arrow[r, shift left, "U"] \arrow[rr, bend right, "L"'] & \boxed{12348675} \arrow[l, shift left, "D"] & \boxed{12345687} \arrow[ll, bend right, "R"']
\end{tikzcd}$$

Fig. 5: Part of the solution graph with vertices representing the configurations in Fig. 4 and edges representing moves between them.

We can then recover, for each configuration, the next move on the shortest path to the solution. These can be stored in a data file containing optimal next move for each of the $9! = 362,880$ configurations (including, for convenience sake, the unsolvable ones). Each entry takes up 2 bits, giving a total of less than 90KB of memory.

A straightforward implementation in C++ took 250ms of CPU time to run using a single core of an Intel® Core$^{\text{TM}}$ i7-3930K CPU @ 3.2GHz to determine the shortest solutions for all 181,440 solvable configurations of the 8-puzzle (the even permutations of the tiles). Fig. 2 shows the distribution of shortest solution lengths, and Fig. 3 shows the expected (left) and worst-case (right) solution lengths when the blank is in each of the 9 possible initial positions. We were able to verify that 31 moves are required in the worst case, and determined that the expected number of moves is slightly less than 22.

By comparison, in 10^9 random trials the greedy algorithm solved the 8-puzzle in used expected number of moves 44.76 (just over double the optimum) and a maximum of 82 moves (just over 2.65 times the optimum).

IV. The 15-Puzzle

While the maximum number of moves required to solve the 15-puzzle is known to be 80 (Brüngger et al. [2]), a table-based approach similar to that used in Section III is technologically infeasible for the average user since the 15-puzzle has $15! \approx 1.3 \times 10^{12}$ solvable configurations, which at 2 bits per configuration would require a table of size of approximately 327GB. Our aim is to reduce this table size at small cost to the solution length.

Observing the optimal solutions for the 8-puzzle on some random examples, one is struck by the fact that many of them solve the first row and column, then proceed to the remaining 2×2 sub-puzzle while leaving the first row and column in

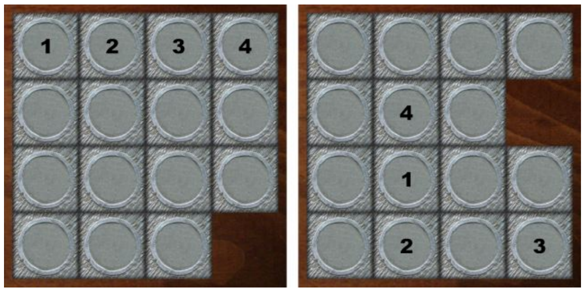

Fig. 6: Stage 1 of the 15-puzzle in its solved configuration (left) and a random configuration (right).

place. This makes sense because the first row and column are the tiles that are furthest away from the home position of the blank in the lower right-hand corner. Furthermore, most of these either solve the first row then the first column while leaving the first row in place, or vice-versa.

We propose to solve the 15-puzzle in much the same way, that is, solve the first row (which we will call *Stage 1*, Section IV-A), then solve the first column without disturbing the first row (which we will call *Stage 2*, Section IV-B), and finally solve the remaining 3×3 sub-puzzle optimally using the method of Section III. We repeat with the roles of "row" and "column" interchanged, and take the shortest solution of the pair. Section IV-C will analyze the resulting algorithm

A. *Stage 1 of the 15-Puzzle*

Optimal solutions for the first row of the 15-puzzle can be computed using the breadth-first search code from Section III with the appropriate start vertex, as follows. Consider a variant of the 15-puzzle in which only tiles 1 through 4 are numbered. The remaining tiles are unnumbered and may be placed in any position in the solved configuration provided the first row is in place. Fig. 6 shows two examples.

Stage 1 configurations can represented as hexadecimal numbers by using digit 4 for all of the tiles not in the first row when solved, for example, the solved configuration of Fig. 6 is represented by

```
0x012344444444444F
```

and the right-hand configuration is represented by

```
0x444443204414444F.
```

The solution graph for Stage 1 of the 15-puzzle is defined similarly to that of the 8-puzzle described in Section III. We then run breadth-first search on this solution graph starting from vertices

```
0x0123F4444444444F
0x01234F444444444F
0x012344F44444444F
0x0123444F4444444F
```

which represent the first row tiles in the correct places and the blank in the second row. Note that the breadth-first search algorithm works without modification on multiple start vertices. (To see this, imagine a new ghost start vertex with edges of cost zero leading to the multiple start vertices.)

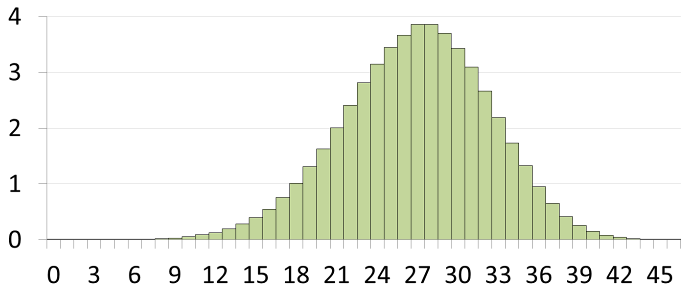

Fig. 7: The distribution of shortest solution lengths for Stage 1 of the 15-puzzle. The horizontal axis shows the possible lengths of shortest paths, and the vertical axis shows the number of puzzle configurations that have a shortest path of that length, measured in tens of thousands.

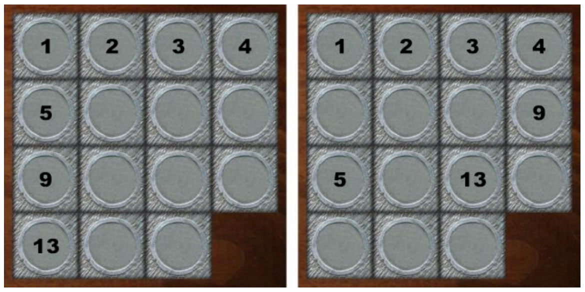

Fig. 8: Stage 2 of the 15-puzzle in its solved configuration (left) and a random configuration (right).

Stage 1 of the 15-puzzle has $16!/11! = 524,160$ configurations. Since the entries are so sparse, the best way of storing them is to use a list of configuration-move pairs managed using a hash table. Therefore at 9 bytes per entry, a solution table would require approximately 4.5MB.

Our implementation took 1031ms of CPU time to run using a single core of an Intel® Core^{TM} i7-3930K CPU @ 3.2GHz to determine the shortest solutions for all 524,160 configurations of Stage 1 of the 15-puzzle. Fig. 7 shows the distribution of shortest solution lengths. We found that 46 moves are required in the worst case, that the expected number of moves is approximately 26.87.

B. *Stage 2 of the 15-Puzzle*

Stage 2 of the 15-puzzle has only tiles 1–4, 5, 9, and 13 numbered, but tiles 1, 2, 3, and 4 are in their solved positions and are not free to move. Fig. 8 shows two examples. Stage 2 configurations can be represented as hexadecimal numbers by using digit 5 for all of the tiles not in the first row when solved. For example, the solved configuration of Fig. 8 is represented by

```
0x012345559555D55F
```

and the right-hand configuration is represented by

```
0x0123555945D5555F.
```

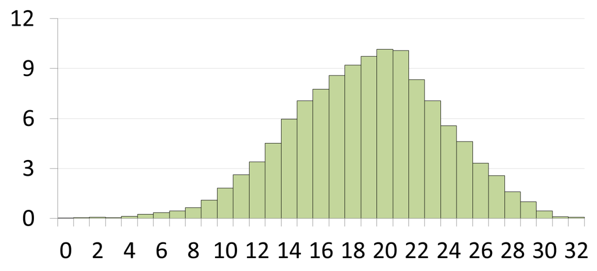

Fig. 9: The distribution of shortest solution lengths for Stage 2 of the 15-puzzle. The horizontal axis shows the possible lengths of shortest paths, and the vertical axis shows the number of puzzle configurations that have a shortest path of that length, measured in hundreds.

The solution graph for Stage 2 of the 15-puzzle is also defined similarly to the above.

We run breadth-first search on the solution graph for Stage 2 of the 15-puzzle with start vertices

```
0x01234F558555C555F
0x012345558F55C555F
0x012345558555CC55F
```

representing the first row and column tiles in the correct places and the blank in the second column but not the first row. We also need to add a line of code to ensure that only configurations with the first row in place get inserted into the breadth-first search vertex queue.

Stage 2 of the 15-puzzle has $12!/8! = 11,880$ configurations, and therefore at 9 bytes per entry, a solution table would require approximately 104KB.

Our implementation took 93ms of CPU time to run using a single core of an Intel® Core$^{\text{TM}}$ i7-3930K CPU @ 3.2GHz to determine the shortest solutions for all 11,880 configurations of Stage 2 of the 15-puzzle. Fig. 9 shows the distribution of shortest solution lengths. We found that 32 moves are required in the worst case, and that the expected number of moves is around 18.83.

C. Complete Solution of the 15-Puzzle

Our new algorithm, which solves in turn Stage 1 of the 15-puzzle, Stage 2 of the 15-puzzle, then the 8-puzzle therefore uses 4.7MB of tables to solve the 15-puzzle in at most 108 moves with expected number of moves at most 67.75 under the assumption that our solution to Stage 2 of the 15-puzzle leaves the remaining 8-puzzle in a state that is close to being drawn uniformly at random from the space of all configurations, which we will call the *Uniform Distribution Assumption*. These results are compared in Table II to the theoretical optimum bound from Brüngger et al. [2], and the expected and worst-case upper bounds of $2.66n^3$ and $5n^3$ from [6] and [7], respectively. Re-running the experiment of Section III for solutions of this type, we find that the worst case is 30 (down from 31), the average case is 22.04 (up slightly from 21,97), and the distribution of solution lengths is only slightly different (compare Fig. 10 to Fig. 2).

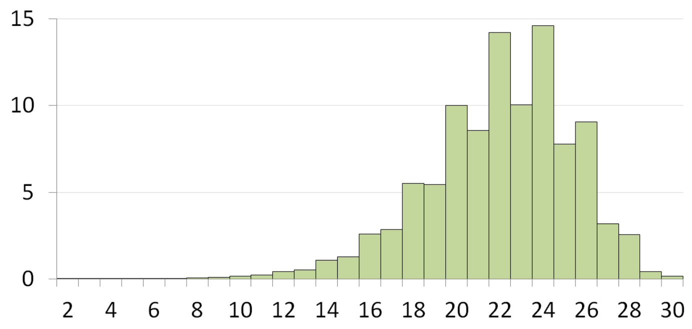

Fig. 10: The distribution of shortest solution lengths for the 8-puzzle when the blank is in the first row or column. The horizontal axis shows the possible lengths of shortest paths, and the vertical axis shows the number of puzzle configurations that have a shortest path of that length, measured in thousands. (Compare to Fig. 2)

	Expected	Max
Part 1	26.87	46
Part 2	18.83	32
3×3	22.04*	30
Total	67.75*	108

TABLE I: Theoretical number of moves for the new 15-puzzle algorithm. The starred value is under the Uniform Distribution Assumption.

	Expected	Max
Optimum	?	80
New	67.75*	108
Greedy	171	320

TABLE II: Theoretical number of moves made by the new 15-puzzle algorithm compared to the optimal number of moves from Brüngger et al. [2] and the number of moves made by the greedy algorithm. The starred value is under the Uniform Distribution Assumption.

	Min	Mean	Max
New	20	65.21	95
Greedy	40	118.16	195

TABLE III: Experimental number of moves made by the new 15-puzzle algorithm compared to the greedy algorithm on 10^6 random configurations.

We ran the greedy algorithm against the new algorithm on 10^6 random puzzle instances and found that in practice the new algorithm uses 49% as many moves as the greedy algorithm in the worst case and 55% as many moves on average (see Table III). The Uniform Distribution Assumption was validated by these results. While the greedy algorithm averaged less than 0.04ms per solution, the new algorithm averaged less than 0.07ms per solution on the aforementioned Intel® Core$^{\text{TM}}$ i7-3930K CPU @ 3.2GHz.

Some animations of the greedy algorithm and the new algorithm solving the same random instances of the 15-puzzle can be found online at the following URL.

`http://larc.unt.edu/ian/research/15puzzle`

V. Conclusion and Open Problems

We have shown how to find 15-puzzle solutions of worst case length 93 and expected length less than 66 in under 0.07ms per solution on current desktop computing hardware using 4.7MB of tables. These numbers compare favorably to the worst-case length upper bound of 80 from Brüngger et al. [2], and to the greedy algorithm of Parberry [6], [7] which experimentally used 190 moves in the worst case and 118 moves on average on 10^6 random configurations. Open problems include improving these figures, and determining the average number of moves used in optimal length solutions to the 15-puzzle.

References

[1] L. E. Hordern, *Sliding Piece Puzzles*. Oxford University Press, 1986.

[2] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt, "The parallel search bench ZRAM and its applications," *Annals of Operations Research*, vol. 90, pp. 45–63, 1999.

[3] A. Reinefeld, "Complete solution of the eight-puzzle and the benefit of node ordering in IDA*," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993, pp. 248–253.

[4] D. Ratner and M. Warmuth, "The $(n^2 - 1)$-puzzle and related relocation problems," *Journal of Symbolic Computation*, vol. 10, no. 2, pp. 111–137, Jul. 1990. [Online]. Available: http://dx.doi.org/10.1016/S0747-7171(08)80001-6

[5] D. M. Kornhauser, G. Miller, and P. Spirakis, "Coordinating pebble motion on graphs, the diameter of permutation groups, and applications," in *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 1984, pp. 241–250.

[6] I. Parberry, "A real-time algorithm for the $(n^2 - 1)$-puzzle," *Information Processing Letters*, vol. 56, pp. 23–28, 1995.

[7] ——, "Solving the $(n^2 - 1)$-puzzle in real time with $\frac{8}{3}n^3$ expected moves," Laboratory for Recreational Computing, Dept. of Computer Science & Engineering, Univ. of North Texas, Tech. Rep. LARC–2014–01, April 2014.

[8] E. F. Moore, "The shortest path through a maze," in *Proceedings of the International Symposium on the Theory of Switching*, 1959, pp. 285–292.

[9] C. Lee, "An algorithm for path connection and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

Ian Parberry received his PhD in Computer Science from the University of Warwick in England in 1984. He is currently a Professor in the Department of Computer Science and Engineering at the University of North Texas. He is the author of seven books and over 90 articles on subjects including game programming, procedural content generation, computer graphics, parallel computing, circuit complexity, sorting networks, mathematical puzzles, and neural networks.