The 15-Puzzle: From Solvability Check to Fast Sub-Optimal Solution Search

Bohdan Kostyk[1]

[1]Faculty of Applied Mathematics, Silesian University of Technology, Kaszubska 23, 44-100 Gliwice, Poland

Abstract

This article is divided into two parts. In the first, there is a description of the solvability condition of the 15-puzzle with the corresponding formulas provided. This is followed by a second part regarding the solution search for a valid 15-puzzle state. The article's goal is not to explore the mathematical depth of each part, but to provide a complete instruction or algorithm of how such a problem is solved using existing formulas and structures, and to explore the problem with a modern outlook

Keywords

15-puzzle, solvability analysis, heuristic search, A* algorithm, Manhattan distance, linear conflict, weighted cost function, sub-optimal search, optimality-time trade-off

1. Introduction

The problem of identifying the solvability of the 15-puzzle is well-known in the world of mathematics and computer science. This problem was first formally described by Wm. Woolsey Johnson and William E. Story [1]. Later, in 1999, Aaron F. Archer [2] provided an explanation of this problem in a modern context. The formulas presented here have their roots in these foundational articles. The particular goal of this part is to show the condition of solvability for a randomly generated 15-puzzle before the start of searching for a solution. This is necessary because an unsolvable 15-puzzle state will make the solution-search algorithm struggle and lead to an infinite search.

In this article's second part, a weighted cost function adapted from the work of Ira Pohl [3] was applied to search at high speed for a sub-optimal solution.

2. Solvability

2.1. Number of Inversions

Let T be a one-dimensional sequence representing a randomly generated state of the 15-puzzle.In a general state, T consists of 16 elements. The number of inversions in state T, omitting the zero element, can be defined by the formula:

$$I(T) = \sum_{i=0}^{14} \sum_{j=i+1}^{15} \mathbb{I}(t_i, t_j) \tag{1}$$

Where $\mathbb{I}(t_i, t_j)$ is indicator function with an additional condition:

$$\mathbb{I}(t_i, t_j) = \begin{cases} 1 & \text{If } t_i > t_j \wedge t_i \neq 0 \wedge t_j \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

IVUS 2026: 31st International Conference on Information Society and University Studies, 12 May 2026, Kaunas, Lithuania

bk318913@student.polsl.pl (B. Kostyk)

Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-06

CEUR-WS.org/Vol-4317/paper27.pdf

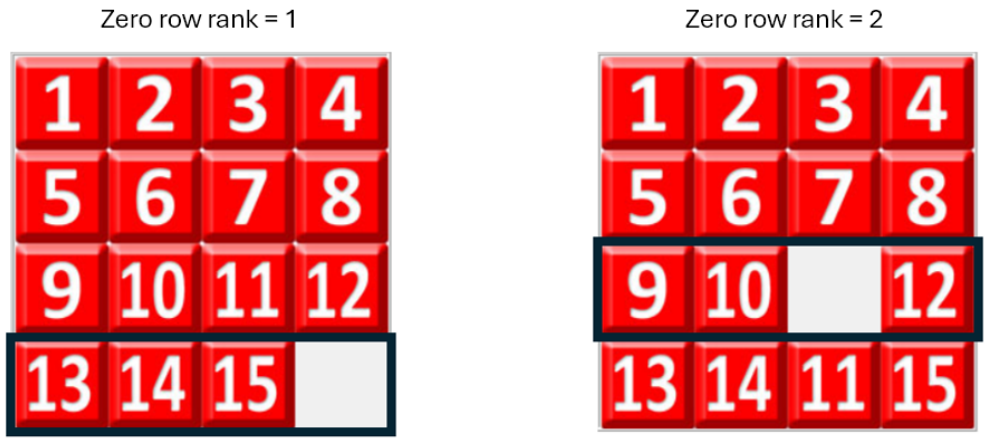


Figure 1: Identifying zero row rank

2.2. Zero Row Rank

If $pos(0)_T$ is an index of a blank element in one dimensional representation of a random state e.g. $T = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0\}$, where elements of T with indices from 0 to 3 belong to first row, elements of T with indices from 4 to 7 belong to second row, and so on, then the zero row rank is an ordinal number of the row, where blank element is located in state.

$$r_T = 4 - \left\lfloor \frac{pos(0)_T}{4} \right\rfloor \tag{3}$$

$\left\lfloor \frac{pos(0)}{4} \right\rfloor$ is a floor division to find a row where zero is located. Rows are counted from 1 to 4 beginning from the visually bottom row (see Fig. 1).

2.3. Final Statement

State T is solvable if and only if the parity of the number of inversions is opposite to the parity of the zero row rank:

$$I(T) + r_T \equiv 1 \pmod 2 \tag{4}$$

3. Structures

The solving process utilizes a few key structures that provide the necessary information for the algorithm.

3.1. Zero Possible Moves Table

In any programming language, a precomputed table of possible moves is necessary to provide information on where the algorithm can move the blank element from the current state.

3.2. Node

A Node in this particular implementation is a structure that preserves information about the state of the 15-puzzle. It includes:

- The current state configuration.
- The parent state (the state immediately preceding the current one).
- The zero position (used to identify valid moves via the Zero Possible Moves Table).
- The heuristic value.
- The number of moves made to reach this state.

3.3. Priority Queue

The priority queue used in the algorithm is modified for higher performance, based on a min-heap structure. The root of the tree always contains the element with the highest priority, which is a pair consisting of a node and its weighted cost function value (5). This value determines the priority: the element with the smallest cost function value sits at the root and is the first to be extracted from the heap.

3.4. Visited Nodes Set

The Visited Nodes Set keeps track of nodes that have already been processed to prevent duplicates from being added to the priority queue. This structure ensures that the priority queue does not store redundant nodes, thereby keeping it smaller and making enqueue and dequeue operations more effective.

4. Solving Method

The solution search is based on the A* algorithm, which is enhanced by the following techniques:

- **Heuristic function** - $h(n)$:
  - **Manhattan Distance**: The sum of the vertical and horizontal distances of each tile from its target position in the solved state. This distance is calculated by summing the absolute differences of the row and column coordinates.
  - **Linear Conflict**: An enhancement that adds a penalty (+2) to the overall heuristic value when two tiles are in their correct target row or column but are reversed relative to each other. This creates a ”conflict” because one tile must temporarily move out of the row/column to allow the other to pass, requiring at least two extra moves [4].
- **Searching strategy**:
  - **Weighted Cost Function**:
    $$f(n) = g(n) + w \cdot h(n) \tag{5}$$
    Where:
    - $g(n)$ - The number of moves made to reach the current state.
    - $h(n)$ - The heuristic function applied to the current state.
    - w - A weight factor ($w > 1$) that deliberately overestimates the heuristic importance. This speeds up the searching process significantly, though at the cost of solution optimality (admissibility).
  - **Priority Queue**: Usage of a priority queue based on a min-heap structure for efficient node retrieval.
- **Optimization**:
  - **Visited Nodes Set**: A structure to prevent processing the same state multiple times.

Table 1
Moves and time comparison for different weights used in weighted cost function

Weight	Moves	Time
w = 1.6	90	0.28s
w = 1.7	96	1.28s
w = 1.5	88	62.79s



Figure 2: One of the 17 states which need minimum 80 moves to be solved

5. Sub-optimal and Optimal Search Comparison

5.1. The Worst Case

In this comparison, different weights were applied to the weighted cost function to observe how many moves would be required to solve the same state and how much time would elapse before a sub-optimal solution was found. For this specific comparison, I utilized one of the 17 distinct 15-puzzle states known to require the maximum number of moves (specifically, states that require at least 80 moves).

Using Herbert Kociemba's application [5] with the largest heuristic database (consuming approximately 550 MB of memory), the optimal solution for the specific 15-puzzle state (see Fig. 2) was found in 0.77s. All sub-optimal searches were performed using custom code written in Python.

It becomes clear that, even when using an interpreted high-level language like Python, the results of the sub-optimal search are noticeably fast. Furthermore, this solution does not require the creation of large heuristic databases (pattern databases), unlike the fastest optimal 15-puzzle solvers [6]. (See Tab. 1).

5.2. Performance Analysis

In this comparison, a weighted cost function with weight of w = 1.5 was applied to different randomly generated states. Table 2 presents the number of moves required to reach the sub-optimal solution, the computation time elapsed, and the optimal number of moves required to solve the initial state.

6. Discussion

The weighted cost function has distinct advantages over the standard cost function, especially in the specific problem space of the 15-puzzle, where a few additional moves do not significantly complicate the solution. Given that the expected value for the optimal solution length of the 15-puzzle is approximately 52.59 moves (see Fig. 3), the weighted cost function typically yields a solution only 1–2 moves longer than optimal, or sometimes matches the optimal length exactly for common states. The weights can be adjusted if an even faster search is required. This algorithm has demonstrated high performance even when implemented in Python, an interpreted high-level language which is typically not the primary choice for computationally intensive algorithms. Furthermore, no external database is necessary for this approach. In contrast, fast optimal solution searches generally require precomputed pattern databases.

Table 2
Performance of A* with weighted cost function in terms of moves and time

Initial State	Moves (Sub-optimal)	Time (s)	Moves (Optimal)
14 1 **0** 13 6 3 11 9 7 2 5 15 12 8 4 10	66	0.44	60
3 10 **0** 12 2 8 1 9 6 15 7 14 11 13 4 5	56	0.21	54
11 12 7 4 10 **0** 5 1 6 8 9 3 2 15 13 14	54	0.59	54

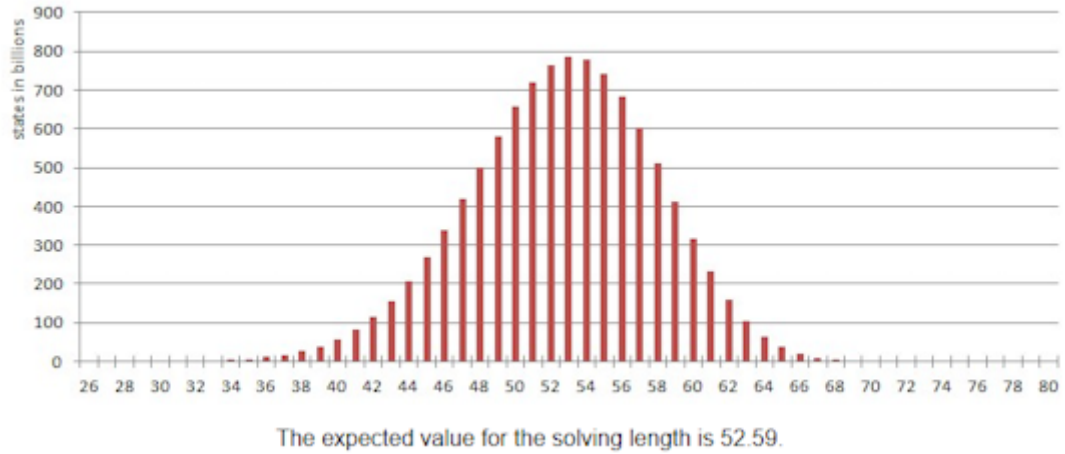


Figure 3: Distribution of all solvable states of 15-puzzle. X axis shows the number of moves need to be done to reach the optimal solution https://kociemba.org/themen/fifteen/fifteensolver.html

Generating process can take hours, and these databases can possibly demand significant static storage space. It can be argued that optimal solvers are superior, but when the goal is to solve any 15-puzzle state using a standalone script without external dependencies, the weighted cost function can be a better choice. A known disadvantage of the standard A* algorithm is its high memory consumption to track discovered states; however, the weighted cost function mitigates this risk by guiding the search more aggressively toward the solution, thereby reducing the number of stored nodes.

Declaration on Generative AI

During the preparation of this work, the author used Gemini 3.0 in order to: Grammar and spelling check, Paraphrase and reword, Plagiarism detection. After using this service, the author reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] W. W. Johnson, W. E. Story, Notes on the "15" puzzle, American Journal of Mathematics 2 (1879) 397–404.

[2] A. F. Archer, A modern treatment of the 15 puzzle, The American Mathematical Monthly 106 (1999) 793–799. URL: http://www.jstor.org/stable/2589612.

[3] I. Pohl, U. of Edinburgh. Department of Machine Intelligence & Perception, First Results on the Effect of Error in Heuristic Search, MIP-R-, Edinburgh University, Department of Machine Intelligence and Perception, 1969. URL: https://books.google.com/books?id=6TcKcgAACAAJ.
[4] O. Hansson, A. Mayer, M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, Information Sciences 63 (1992) 207–227. URL: https://www.sciencedirect.com/science/article/pii/002002559290070O. doi:https://doi.org/10.1016/0020-0255(92)90070-O.
[5] H. Kociemba, 15-puzzle optimal solver, 2018. URL: https://github.com/hkociemba/FifteenPuzzle?tab=readme-ov-file.
[6] A. Felner, R. E. Korf, S. Hanan, Additive pattern database heuristics, Journal of Artificial Intelligence Research 22 (2004) 279–318. URL: http://dx.doi.org/10.1613/jair.1480. doi:10.1613/jair.1480.

A. Online Resources

The source of the project implemented in Python

- GitHub