DOI: http://dx.doi.org/10.26483/ijarcs.v16i5.7348

Volume 16, No. 5, September-October 2025

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

AN EXPERIMENT OF THE COMPLEXITY OF SLIDING BLOCK PUZZLES BY 2D HEAT FLOW IN PARAMODULATION

Ruo Ando
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430 Japan

Yoshiyasu Takefuji
Musashino University
3-3-3 Ariake, Koto-Ku, Tokyo 135-8181, Japan

Abstract: In this paper, we present a curious experiment with the hot list strategy in solving sliding block puzzles by paramodulation. The hot list strategy is one of the look-ahead strategies using paramodulation in automated reasoning. We define two heat flows in the reasoning process - vertical with the hot list of permutations along the Y-axis and horizontal along the X-axis. In the experiment, we have generated 500 * 8 puzzles under the test of the solvability checking by counting inversions. We have obtained curious 2D and 3D plots of the complexity by defining heat flow with hot lists. We can distinguish a few groups in 500 boards (puzzles) based on the concept of heat-resisting.

Keywords: Paramodulation, Hot List Strategy, Sliding Block Puzzle, Given Clause Algorithm, Heat Flow, Complexity Measurement

I. INTRODAUCTION

A sliding puzzle (also called a sliding block puzzle) is a combination puzzle where a player slides pieces along certain routes on a board to reach a certain end configuration (state).

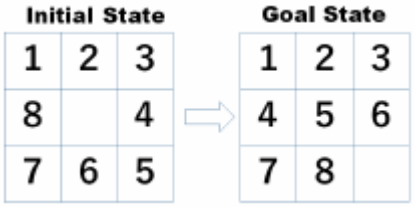

Figure 1. Initial state and goal state of 8 puzzle.

In sliding puzzles, a player is prohibited from lifting any piece off the board. This constraint separates sliding puzzles from rearrangement puzzles. Consequently, discovering routes opened up by each move with the two-dimensional confines of the board is an interesting point of solving sliding block puzzles. Figure 1 shows the example of a sliding puzzle. The puzzle has 9 square slots on a square board.

The first eight slots have square pieces. The 9th slot is empty. Sliding block can be represented as the permutation. A permutation of a set S is a bijection from S onto itself. If the set we permuting is $ A = {1,2, ..., n} $, it is often convenient to represent a permutation sigma as follows:

$$\sigma = \begin{Bmatrix} 1, & 2, & 3, & \dots & \\ \sigma(1), & \sigma(2), & \sigma(3), & \dots, & \sigma(n) \end{Bmatrix} \quad (1)$$

For instance, consider the set A = {1,2,3,4,5,6}. Then the permutation π,

$$\pi = \begin{Bmatrix} 1, & 2, & 3, & 4, & 5, & 6 \\ 4, & 1, & 5, & 2, & 3, & 6 \end{Bmatrix} \quad (2)$$

sends 1 to 4, 2 to 1, 3 to 5 and fixes, or leaves unchanged, the element 6.

The theorem prover OTTER (Organized Techniques for Theorem-proving and Effective Research) has been developed by W. McCune as a product of Argonne National Laboratory. OTTER is based on earlier work by E. Lusk, R. Overbeek, and others [1]. OTTER adopts the given-clause algorithm and implements the set of support strategy [2]. In this paper we use OTTER for our experiments.

Algorithm 1 Given clause algorithm

Input: SOS, Usable List
Output: Proof

```
1:  while until SoS is empty do
2:    choose a given clause G from SoS;
3:    move the clause g to Usable List;
4:    while  c_1, ..., c_n in Usable List do
5:      while R(c_1, ..c_i, G, c_{i+1}, ..c_n)exists do
6:        A ⇐ R(c_1, ..c_i, G, c_{i+1}, ..c_n);
7:        if A is the goal then
8:          report the proof;
9:          stop
10:       else {A is new odd}
11:         add A to SoS X
12:       end if
13:     end while
14:   end while
15: end while
```

Algorithm 1. Given clause algorithm

II. GIVEN CLAUSE ALGORITHM

OTTER adopts given-clause algorithm in which the program attempts to use any and all combinations from axioms in the given clause. In other words, the combinations of the clause are generated from given clauses which have been focused on. Given clause algorithm is shown in Algorithm 1.

At line 2, given clause G is extracted from SoS (Set of Support). Line 4 and 5 is a loop to use any and all combinations of the given clause and Usable List. In detail, \cite{Slaney} discuss the basic framework of the given clause algorithm.

III. PARAMODULATON

A. *Formulation*

Paramodulation, which is introduced by [3], is a powerful method of equational reasoning. Paramodulation takes two clauses of which at least cone contains positive equality literal r.

$$\frac{\{L_1, L_2, ...L_n\} \quad \{l \equiv r, K_2...K_m\}\exists p, \sigma.\sigma = mgu(L1/p, l)}{\{L_1/p \leftarrow r, L_2, ..., L_n, K_2, ..., K_m\}\sigma} \tag{3}$$

According to [4], paramodulation is a realization of Leibniz's substitution of equals by equals. In the case that L1 contains a subterm at a position p which is unifiable with l, the paramount can be computed with the unifier sigma. For example, consider two clauses.

$$\begin{array}{l}\{P(\gamma, h(f(\alpha, y), \beta))\} \\ \{f(x, \gamma) \equiv g(x), Q(x)\}\end{array} \tag{4}$$

(4) generates the new clause.

$$\{P(\gamma, h(g(\alpha, \beta)), Q(\alpha)\} \tag{5}$$

From the mathematical rules x+0=x and -y+y=0.

$$\begin{array}{l}\{plus(x, 0) \equiv x\} \\ \{plus(minus(y), y) \equiv 0\}\end{array} \tag{6}$$

generates from the first into the second of the clauses.

$$\{minus(0) \equiv 0\} \tag{7}$$

Paramodulation uses unification, while demodulation adopts matching.

Algorithm 2 para_into (paramodulation into given clause)

Input: given clause
Output: subterm list

```
1:  into_literal = given_clause → first_literal
2:  while into_literal != NULL do
3:    subterm_list = into_literal → term → list
4:    while subterm_list != NULL do
5:      subterm_list → path = 1
6:      para_into_terms(subterm_list, into_literal)
7:      subterm_list → path = 0
8:      subterm_list = subterm_list → next
9:      into_literal = into_literal
10:   end while
11: end while
```

Algorithm 2. Paramodulation

B. *Implementation*

Concerning the implementation of OTTER, in paramodulation, two parents and a child are processed. The parent clauses contain the equality applied for the replacement. The parent clauses are divided into two: from parent and from clause. If equality comes from the literal, the side of equality unifies with the term, which is replaced with from the term. The replaced term is called the into the term. The literal containing the replaced term is also called the into literal. Also, the parent containing the replaced term is called the into the parent or into clause. Paramodulation is divided into two procedures: para_into and para_from.

- **para_into.** Paramodulation into the given clause. When we make an inference by the para_into rule, we paramodulte into the given clause from containing positive equality and on the usable list.
- **para_from.** Paramodulation from the given clause. When we make an inference by the para_from rule, the given clause contains positive equality, and the inference is made by paramodulating using this equality into a clause.

In this paper, we use the rule of para_into. The procedure of para_into is invoked from infer_and_process taking the given clause.

Algorithm 2 has two loops. The first one (lines 2 to 11) is over literals. The second one (lines 4 to 10) is over terms. Clauses, literals, and terms are defined as follows.

- Clause is an expression formed from a finite collection of literals (atoms or their negations).
- Literal is an atomic formula (atom) or its negation.
- A variable, a constant and an n-ary function symbol applied to n terms
-

At line 6, OTTER computes paramodulants over current subterm lists.

IV. HOT LIST STRATEGY

The hot list strategy [5] is one of the look-ahead strategies. Look-ahead strategies are designed to enable the program to evade many CPU hours to draw conclusions. The conclusion to draw may require focusing on a retained clause.

Definition of the hot list strategy. The hot list strategy enables the program to specify the facts by revisiting the hot clause repeatedly in the context of completing the application of an inference rule. For implementing the hot list strategy, the main loop based on the given clause algorithm should be modified. The main loop for inferring and processing clauses and searching for a refutation operates mainly on the lists usable and SoS.

- Choose appropriate given_clause in SoS;
- Move given_clause from list(SoS) to list(usable)
- Infer and process new clauses using the inference rules set.
- Newly generated clause must have the \$given_clause\$.
- Do the retention test on new clauses and append those to list(SoS).

Figure 2 shows the chart flow of modifying the main loop for the hot list strategy. The hot list strategy is designed to make some set of clauses (hot lists) immediately considered with each newly retained clause. With the modification, if the program passes the branch on the lower side of Figure 2, which is ``hotlist exists?'', the paramodulation routine (para_into) is immediately invoked in the post-process.

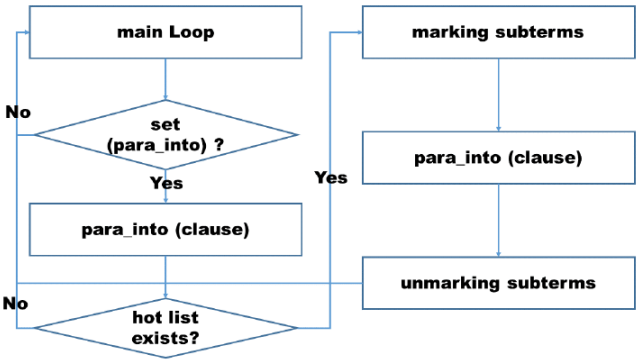

Figure 2. Hot list sttegy by modifying main loop

V. METHODOLOGY

Wherever Times is specified, Times Roman or Times New Roman may be used. If neither is available on your word processor, please use the font closest in appearance to Times. Avoid using bit-mapped fonts if possible. True-Type 1 or Open Type fonts are preferred. Please embed symbol fonts, as well, for math, etc.

A. Setting Inferene Rule set

As we discussed before, the basic inference mechanism of OTTER is based on the given-clause algorithm. Given-clause algorithm can be viewed as a simple implementation of the set of support strategy. OTTER maintains four lists of clauses: usable, SoS, demodulator, and passive. In our case, we cope with two kinds of clauses: usable and SoS. Horizontal sliding from row[i] to row[i+1] is represented as follows.

```
list(usable).
EQUAL(l(hole,l(n(x),y)),l(n(x),l(hole,y))).
end_of_list.
```

$$\sigma = \left\{ \begin{array}{cccc} 1 & hole & 2 & 3 \\ 1 & 2 & hole & 3 \end{array} \right\} \tag{8}$$

Vertical sliding from row[i] to row[i+4] is represented as follows.

```
list(usable).
EQUAL(l(hole,l(x,l(y,l(z,l(u,l(n(w),v)))))),
l(n(w),l(x,l(y,l(z,l(u,l(hole,v))))))).
end_of_list.
```

$$\sigma = \left\{ \begin{array}{cccccccc} 1 & hole & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 & 5 & hole & 6 & 7 \end{array} \right\} \tag{9}$$

B. Generating puzzles (boards)

In general, to check the solvability of N puzzles, the number of inversions of each number of N slots is calculated. For example, if we have the board configuration board [2,3,6,1,7,8,5,4, hole] (5,2,8,4,1,7, hole, 3,6), the number of inversions are as follows:

1) 2 precedes 1 - 1 inversions
2) 3 precedes 1 - 1 inversion
3) 6 precedes 1, 5, 4 - 3 inversions
4) 1 precedes none - 0 inversions
5) 7 precedes 5, 4 - 2 inversions
6) 8 precedes 5, 4 - 2 inversions
7) 5 precedes 4 - 1 inversions
8) 4 precedes none - 0 inversions

Total inversions 1+1+3+0+2+2+1+0 = 10 (Even Number) So this puzzle configuration is solvable. On the other hand, it is not possible to solve an instance of 8 puzzles if a number of inversions are odd in the input state.

Algorithm 3 Checking the solvability of N puzzles

Input: Board$[x_1, x_2, ..., x_n, hole]$
Output: SOLVABLE or UNSOLVABLE

1: $Board[X|XS] = Board[x_1, x_2, ..., x_n, hole]$
2: **while** XS in $Board[XlXS]$ is empty **do**
3: **for** i in XS **do**
4: statements..
5: **if** $(X \neq XS[i])$ **then**
6: $counter[i] + +$
7: **end if**
8: **end for**
9: **end while**
10: $line = check(Board[...] \subseteq hole)$
11: $sum = 0$
12: **for** i to n **do**
13: $sum+ = counter[i]$
14: **end for**
15: **if** $(line + sum\%2 == 0)$ **then**
16: $flag =$ SOLVABLE
17: **else**
18: $flag =$ UNSOLVABLE
19: **end if**

Algorithm 3. Checking solvability

Algorithm 3 shows the procedure for checking the solvability of N puzzles. At lines 2 to 9, the number of inversions of each slot is counted. These figures are counted up at lines 11 to 14. Finally, the sum is checked if it is an even or odd number at lines 15 to 19.

VI. EXPERIMENTAL RESULTS

A. Generating puzzles

In the experiment, we have generated 500 sliding puzzles with size 8 * 8.All generated configurations of 8 puzzles are solvable. For each puzzle, we have measured the number of generated clauses with the procedures shown in Algorithm 2. For simplicity, we have generated the configuration of the first 8 slots with random integers ranging from 1 to 8 and fixed 9th slot to hole.

B. Counting clauses

Algorithm 4 shows the brief description of the modified given clause algorithm for counting the generated clauses.

Algorithm 4 Incrementing the number of generated clauses

```
1: while given clause is NOT NULL do
2:    index_lits_clash(giv_cl);
3:    append_cl(Usable, giv_cl);
4:    if splitting() then
5:       possible_given_split(giv_cl);
6:    end if
7:    infer_and_process(giv_cl);
8:    giv_cl = extract_given_clause();
9:    track(the_number_of_generated_clauses);
10: end while
```

Algorithm 4. Counting clauses generated

At line 9, the number of generated clauses is incremented.
After line 8 of picking up the clause from a set of support, we can record the current size of the set of support.
By doing this, we can obtain the plot with \# puzzles and the number of generated clauses of the Y-axis, as shown in the next section.

Table I shows the numerical results of solving 500 puzzles randomly generated. The number of generated clauses with paramodulation ranges from 510 (1,3,5,4,6,8,7,2,hole - easiest) to 188,610 (6,2,7,3,4,5,8,1,hole - the most difficult). In the view of complexity of reasoning process, the configuration [#295 1,3,5,4,6,8,7,2,hole)] is 369.82 times harder to solve than the configuration [#124 (6,2,7,3,4,5,8,1,hole)].

board No	Initial state	clauses generated
#295	1,3,5,4,6,8,7,2,hole	510 (easiest)
#340	8,1,3,4,2,5,7,6,hole	918
#294	2,3,5,1,6,8,7,4,hole	1,362
#86	8,7,6,4,5,2,3,1,hole	188,475
#124	6,2,7,3,4,5,8,1,hole	188,610 (the most difficult)

Table 1. Initial board states and the complexities of paramodulation

Board No	Initial state	vertical	horizontal
#295	1,3,5,4,6,8,7,2,hole	254	388
#340	8,1,3,4,2,5,7,6,hole	502	725
#294	2,3,5,1,6,8,7,4,hole	1,290	1,964
#86	8,7,6,4,5,2,3,1,hole	110,423	100,803
#124	6,2,7,3,4,5,8,1,hole	96,809	100,824

Table 2. The number of clauses generated by vertical/horizontal heat flow

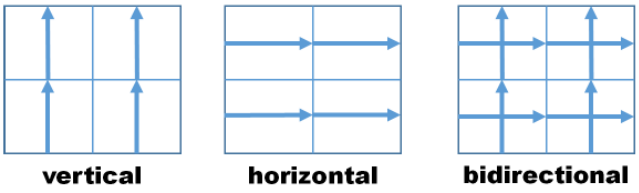

Figure 3. Heat flow in paramodulation.

C. Heat flow

In nature, sliding puzzles are two-dimensional, even if the sliding is facilitated by encaged marbles or three-dimensional tokens. We define the heat flow in paramodulation as follows.

Definition of heat flow. Heat flow makes the reasoning program consider the hot list immediately with vertical and horizontal permutation.

Horizontal heat flow is set by the hot clause as follows:

```
list(hot).
EQUAL(l(hole,l(n(x),y)),l(n(x),l(hole,y))).
```

Also, vertical heat flow is set by the hot clause as follows:

```
list(hot).
EQUAL(l(hole,l(x,l(y,l(z,l(u,l(n(w),v)))))),
l(n(w),l(x,l(y,l(z,l(u,l(hole,v))))))).
end_of_list.
```

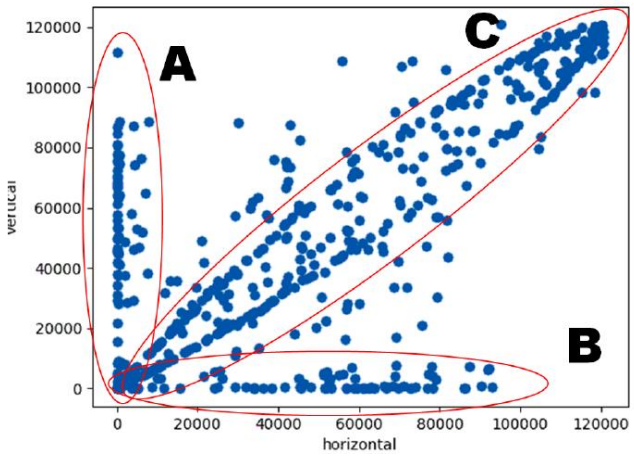

Figure 4. 2D scatter diagram of the number of clauses geerated by solving 500 puzzles

Figure 4 shows the number of clauses generated in solving 500 puzzles. Figure 4 has 500 points of the initial state of the board. The X-axis is the number of clauses generated with vertical heat flow. Y-axis is the number of clauses generated with horizontal heat flow. That is, there are 500 points of boards with point (x,y) where x is the number of clauses generated with horizontal heat flow and y is the number of clauses in vertical heat flow. For example, the points \#295 have the values (388,254) as shown in Table II.

In Figure 4, we distiguish three areas among 500 points.

1. Area A: The boards are affected by vertical heat flow.
2. Area B: Horizontal heat flow are effective on the boards.
3. Area C: Both vertical and horizontal heat flow have effects on the boards

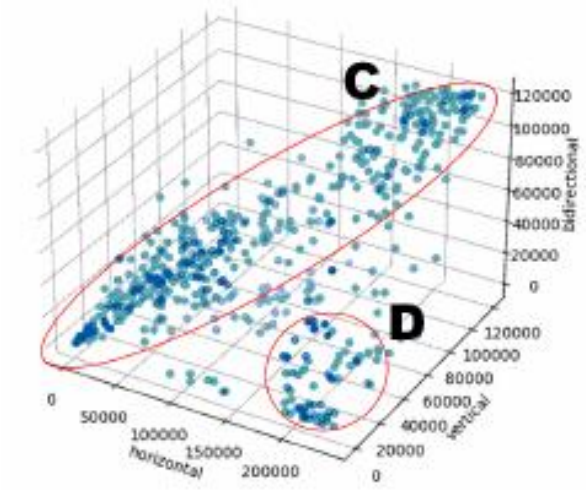

Figure 5. 3D scatter diagram of the number of generated clauses in solving 500 puzzles.

Figure 5 is a 3D scatter diagram of 500 boards. Points in Area C in Figure 4 are also plotted in Figure 5. We see the bulk of points (Area D) in the lower side of the figure (seemingly with little effect of bidirectional heat flow).

VII. RELATED WORK

Historically, Noyes Chapman invented the oldest type of sliding puzzle, which is the fifteen puzzle in 1880. Folklore tells us that in 1886, puzzle master Sam Loyd offered a one-thousand dollar prize if anyone could swap tile 14 and 15 and return the other tiles to their original slots. Archer [6] firstly discusses an algorithmic analysis of 15 puzzles. In [6], a summary of all possible permutations of slots attained by moving the black block from cell i to cell j affecting the permutation of sigma_i,j. Howe [7] proposes two approaches in the two kinds of viewpoints: the properties of permutations and graph theory. Calabro [8] proposes $ O(n^2) $ time algorithm for deciding the time when the initial configuration of the n * n puzzle game is solvable.

Paramodulation originated as development of resolution [12], one of the main computational methods in first-order logic, see [13]. For improving resolution-based methods, the study of the equality predicate has been particularly important since reasoning with equality is well-known to be of the great importance of mathematics, logic, and computer science. Ando et al. [14] propose a measurement of the complexity of sliding block puzzles using paramodulation.

VIII. CONCLUSION

In this paper, we have presented the new novel experiments of the complexity of sliding block puzzles based on the concept of heat flow in paramodulation. Heat flow is set by the hot list with vertical and horizontal permutation. In the experiment, we have generated 500 * 8 puzzles to calculate the number of clauses generated by vertical and horizontal heat flow in the board. We have obtained some curious results. To name a few, board \#295 (1,3,5,4,6,8,7,2,hole) turned out to be easiest with the 2D coordinate (388, 254). Board \#124 (6,2,7,3,4,5,8,1,hole) is the most difficult with the 2D coordinate (96,809 100,824). Also, we have distinguished three areas in 500 points. For one possible further work, we are aiming to leverage this research for the hybrid of algorithmic module

IX. REFERENCES

[1] Ewing L. Lusk, William McCune, Ross A. Overbeek: ITP at Argonne National Laboratory. CADE 1986: 697–698

[2] Larry Wos, George A. Robinson, Daniel F. Carson: Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. J. ACM 12(4): 536–541 (1965)

[3] G. Robinson and L. Wos: Paramodulation and theorem-proving in first order theories with equality. In D. Michie and R. Meltzer (eds.), Machine Intelligence, Vol. IV, pp. 135–150. Edinburgh University Press, 1969.

[4] Peter Graf: Term Indexing (Lecture Notes in Computer Science, 1053), Springer, Mar. 27, 1996.

[5] Larry Wos, Gail W. Pieper: The Hot List Strategy. J. Autom. Reason. 22(1): 1–44 (1999)

[6] A. F. Archer: A Modern Treatment of the 15 Puzzle. The American Mathematical Monthly 106, 793–799, 1999.

[7] Tom Howe: Two Approaches to Analyzing the Permutations of the 15 Puzzle. https://www.whitman.edu/Documents/Academics/Mathematics/2017/

[8] Chris Calabro (2005): Solving the 15-Puzzle.

[9] John K. Slaney, Ewing L. Lusk, William McCune: SCOTT: Semantically Constrained Otter System Description. CADE 1994: 764–768

[10] Ross A. Overbeek: An implementation of hyper-resolution. Computers & Mathematics with Applications, Vol. 1, Issue 2, June 1975, pp. 201–214.

[11] Larry Wos, Gail W. Pieper: The Hot List Strategy. J. Autom. Reason. 22(1): 1–44 (1999)

[12] J. A. Robinson: A machine-oriented logic based on the resolution principle. Journal of the Association for Computing Machinery, Vol. 12 (1965), pp. 23–41.

[13] L. Bachmair, H. Ganzinger: Resolution theorem proving. In A. Robinson, A. Voronkov (eds.), Handbook of Automated Reasoning, Vol. I, Elsevier Science, Amsterdam (2001), pp. 19–99.

[14] Ruo Ando, Yoshiyasu Takefuji: A new perspective of paramodulation complexity by solving massive 8 puzzles. CoRR abs/2012.08231 (2020)

