

Methods for Solving the Post Correspondence Problem and Certificate Generation

Akihiro Omori¹, Yasuhiko Minamide²

¹Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan

²Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan

Abstract

Post Correspondence Problem (PCP) is a well-known undecidable problem. Solving instances with solutions is straightforward with exploration algorithms, but proving infeasibility is challenging. This research introduces two methods to demonstrate infeasibility, including generating formal proofs in Isabelle/HOL.

Keywords

Post's Correspondence Problem, Formal Verification, Reachability Problem, Automata

1. Introduction

The Post Correspondence Problem (PCP), proposed by Post in 1946 [1], is undecidable. PCP instances use tiles with two strings on top and bottom.

100	0	1
1	100	00

In this example, there are three kinds of tiles, each available in infinite quantities. The problem is to determine whether it is possible to arrange one or more tiles in such a way that the reading of the top and bottom strings matches. In this particular instance, a solution (indices of arrangement of tiles) is “1311322”, and this shows that both the top and bottom read “1001100100100”.

100	1	100	100	1	0	0
1	00	1	1	00	100	100

For instances that have a solution, it is possible to find the solution within finite time using an exploration algorithm. On the other hand, determining that no solution exists is challenging, and due to the undecidability of the problem, no general algorithm exists for this purpose. Previous research has proposed heuristic algorithms for finding solutions [2, 3] and Ling Zhao (2003) [2] attempted to solve all the problems in $PCP[3,4]$ and left 3,170 problems unsolved. $PCP[3,4]$ refers to a set of all instances where the number of tiles is 3, and the maximum length of the written strings is 4.

This research makes the following three main contributions.

SCSS 2024: 10th International Symposium on Symbolic Computation in Software Science, August 28–30, 2024, Tokyo, Japan

✉ omori.a.ab@m.titech.ac.jp (A. Omori); minamide@c.titech.ac.jp (Y. Minamide)



© 2024 Copyright 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)

- Propose two novel algorithms to demonstrate that a PCP instance has no solution.
- Solve all problems of $PCP[3,4]$ except for 13 problems.
- Show an example of automatic proof generation for concrete problems.

2. The First Method: String Constraint Formulation

We formulate PCP as a string constraint problem.

Example 2.1 (Example of T_g and T_h). Let PCP instance $I = ((1111, 1110), (1101, 1), (11, 1111))$. We denote the top and bottom strings on the i -th tile by g_i and h_i , respectively. Let T_g and T_h be transducers as defined below. Intuitively, the transducer T_g outputs g_1 for ‘1’, g_2 for ‘2’, and does not accept the empty string. The string w is a solution to the PCP if and only if $T_g(w) = T_h(w)$.

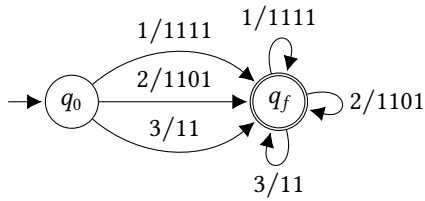


Figure 1: T_g

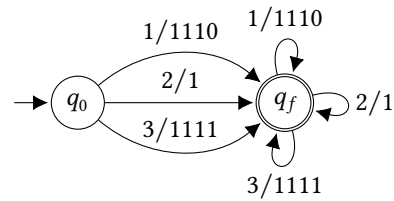


Figure 2: T_h

Definition 2.2 (String Constraint of PCP). The constraint $T_g(w) = T_h(w)$ created in this way is called the string constraint of instance I .

Regarding the string constraint ϕ of the PCP instance I , the satisfiability is undecidable. We consider ϕ' such that $\phi(w) \implies \phi'(w)$ and is efficiently decidable. Such ϕ' is referred to as a relaxation problem (simply relaxation) of ϕ . By showing that ϕ' is unsatisfiable, we would like to show that ϕ is also unsatisfiable. For example, considering only the number of characters $|T_g(w)| = |T_h(w)|$ is suitable as ϕ' . Additionally, matching Parikh images or the number of specific words is another example of ϕ' . Generalizing these examples, we have the following proposition.

Proposition 2.3. Let W be an arbitrary total integer-vector-output transducer. Consider

$$W(T_g(w)) \cap W(T_h(w)) \neq \emptyset \quad (1)$$

This is a relaxation problem of ϕ and is decidable. We set some W and hope it is infeasible.

Although details are omitted, the condition $W(T_g(w)) \cap W(T_h(w)) \neq \emptyset$ can be reduced to the emptiness problem of a Parikh automaton constructed from W , T_g , T_h , and their product. The Parikh automaton emptiness algorithm we use is largely similar to the one described in Section 3 of [4], so we omit the details. While not detailed here, our algorithm achieved significant speedup by applying two techniques to this algorithm: (1) delaying and dynamically adding constraints related to connectivity, and (2) reducing the problem to a natural form for Mixed Integer Programming and leveraging a cutting-edge MIP solver.

3. The Second Method: Transition System Formulation

Intuitively, arranging each tile one by one represents a transition, and “the remaining part of the string and whether it is on the top or bottom” represents a state. We call such a pair *configuration*. PCP can be formulated as a reachability problem: “Is it possible to reach the state of the empty string?”

Example 3.1. When arranging two tiles like $\begin{array}{|c|c|} \hline 100 & 10 \\ \hline 1 & 0 \\ \hline \end{array}$, the state representing it is “top, remainder 010.” If a transition is made by appending $\begin{array}{|c|} \hline 111 \\ \hline 01 \\ \hline \end{array}$, the next state will be “top, remainder 0111.”

3.1. Problem Definition

We formulate PCP as a reachability problem. First, we define the transition system of PCP.

Definition 3.2 (Transition System of PCP). Let $I = ((g_1, h_1), \dots, (g_s, h_s))$ be a PCP instance of size s over Σ . We define the transition system $Tr = (Q, T, Init, Bad)$ as follows.

- State set $Q = \{\text{top}, \text{bottom}\} \times \Sigma^*$.
- Transition function $T : Q \rightarrow 2^Q$ is defined as follows.

$$T(\text{bottom}, w) = \{(\text{bottom}, w') \mid \exists i \leq s. wh_i = g_i w'\} \cup \{(\text{top}, w') \mid \exists i \leq s. wh_i w' = g_i\}$$

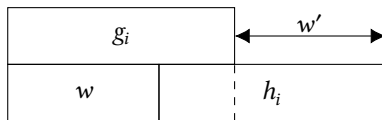
$$T(\text{top}, w) = \{(\text{top}, w') \mid \exists i \leq s. wg_i = h_i w'\} \cup \{(\text{bottom}, w') \mid \exists i \leq s. wg_i w' = h_i\}$$

- Bad state set $Bad = \{(\text{top}, \epsilon), (\text{bottom}, \epsilon)\}$.
- Initial state set $Init = T(\text{top}, \epsilon)$.

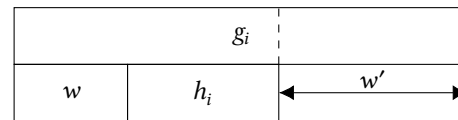
The states after arranging one tile is considered the initial state, as an empty arrangement is not valid.

In the following, T is naturally extended and used as $T : 2^Q \rightarrow 2^Q$.

The behavior of the transition $T(\text{bottom}, w)$ is illustrated below. When w is the current state, adding (g_i, h_i) results in the remaining part becoming the next state w' . There are two patterns: one where the same side as the previous state continues, and one where the side changes.



(a) Pattern where the side doesn't change



(b) Pattern where the side changes

Definition 3.3 (Reachability Problem of PCP). Does there exist n such that

$$T^n(Init) \cap Bad \neq \emptyset$$

Definition 3.4 (Inductive Invariant of PCP). A set Inv that satisfies the following three conditions is called an inductive invariant (simply invariant).

- $Init \subseteq Inv$
- Inv is closed under T : $T(Inv) \subseteq Inv$
- Inv does not include ϵ : $Bad \cap Inv = \emptyset$

Lemma 3.5. If Inv exists, then it implies that Bad is unreachable from initial states.

In the following section, we introduce algorithms to discover Inv .

3.2. Algorithm

For the Reachability Problem, many powerful algorithms like PDR (Property Directed Reachability)[5] exist. We extended PDR and achieved some success (see Section 5). We also devised a novel ad-hoc method specific to PCP, described below.

Definition 3.6 (Configuration Automaton). Let $s \in \{\text{top}, \text{bottom}\}$ and A be a finite automaton over Σ . We call the pair (s, A) the configuration automaton. The language of (s, A) is denoted as $L(s, A)$ and defined as follows. This represents a state set of the transition system.

$$L(s, A) = \{(s, w) \mid w \in L(A)\}$$

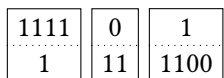
The aim of this algorithm is to discover a pair of configuration automata (for top and bottom) that represents Inv . It should be noted that not every Inv has such a pair due to the regularity of the underlying automata, which limits the scope of our consideration.

This algorithm manages a graph $G = (V, E)$ where each node is a configuration automaton. Specifically, each node v is associated with a set of states of a transition system. The algorithm proceeds by expanding the overall union $L(V) = \bigcup_v L(v)$ until it becomes an invariant.

Intuitively, the edge (u, v) in this graph represents a dependency relationship. This relationship means “if v cannot reach Bad , then u cannot reach Bad either”. If we can construct a graph where every node has such dependencies and does not contain any bad state, then $L(V)$ is an invariant. There are two types of this relation, as follows.

1. Inclusion relation: $L(u) \subseteq L(v)$
2. Transition relation: $T(L(u)) = L(v)$

The algorithm is essentially a breadth-first search (BFS). When considering only the transition relation, the process operates similarly to BFS. A distinctive feature of this algorithm is that it proactively *abstracts* nodes. For example, when a node such as $(\text{top}, 0011101)$ appears, the algorithm attempts to create a node like $(\text{top}, .*110.*)$ (we use a regex to represent an automaton) and draw an edge to it. If this abstracted node can reach Bad , it is removed and backtracking is performed.

Figure 4 shows a successful execution example for . The square nodes represent nodes with singleton languages, and the round nodes are abstracted nodes with regular expressions appearing in their labels. The dotted lines represent inclusion relations, and the solid lines represent transition relations. Note that in this figure, the transition relations are extended to n (where $n \geq 1$) steps, with intermediate steps omitted.

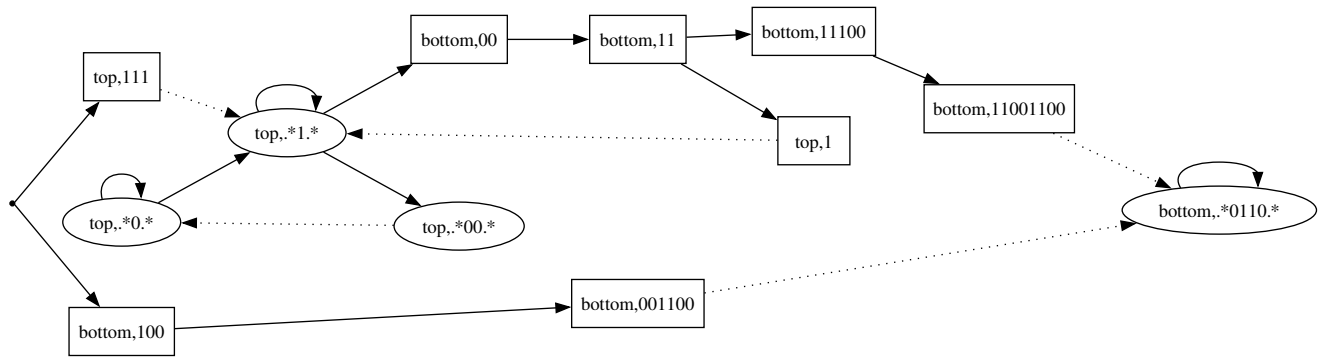


Figure 4: Example of Graph of Invariant

4. Certificate Generation

So far, we have presented two methods and complicated algorithms. However, there is a significant possibility that my implementations for these algorithms may contain bugs. Even if we successfully solve all instances of $PCP[3,4]$, our results would still be far from being considered trusted facts. Therefore, we decided to have our algorithm output proofs in the form of Isabelle/HOL code.

Another possible approach is to use Isabelle/HOL or similar tools to verify the correctness of the algorithm's implementation. However, this makes it difficult to optimize the algorithm for speed. For instance, The first method relies on an external MIP solver for its efficiency, making it challenging. Additionally, for others to quickly trust our results, it is crucial that all instances of $PCP[3, 4]$ and their proofs are organized and verified within some proof assistant such as Isabelle/HOL.

Currently, only the second method is capable of outputting a certificate. The first method will be addressed as future work (see Section 6).

4.1. Certificate: Pair of Automata

Consider the transition system of a PCP instance. By defining the invariant concretely in Isabelle/HOL and proving each of the invariant conditions (see Definition 3.4), we can validate it. This method is independent of the implementation details used in the second method and can be utilized by various algorithms discovering invariants.

Our implementation of the second method generates the following code.

1. Definition of the PCP instance
2. Definition of Inv
 - a) The top-side Automaton
 - b) The bottom-side Automaton
3. Proof of the closedness of Inv
 - a) Definition of $T(Inv)$ (in the form of a specific pair of deterministic automata)

- b) Concrete definition of the automaton for $Inv \cap \overline{T(Inv)}$
- c) Proof of $Inv \cap \overline{T(Inv)} = \emptyset$
- d) Proof of $\overline{Inv} \cap T(Inv)$ similarly, and show that $T(Inv) \subseteq Inv$

Proofs such as “the existence of Inv implies that the PCP has no solution” were conducted manually in advance. Examples of complete proofs are found on the author’s GitHub repository [6].

5. Application to $PCP[3,4]$

In this research, we address the instances of $PCP[3,4]$. Ling Zhao (2003) [2] attempted to solve all these instances but left 3,170 unsolved. The list of these instances is available on his website [7]. Our goal was to solve all instances of $PCP[3,4]$, gradually reducing the number of unsolved problems. As shown in Figure 5, the initial 3,170 unsolved problems were reduced to 127 using the first method. After several additional methods, only 13 problems remained unsolved. These remaining problems are listed on the author’s website [8].

PDR, SAT, Method2(1), and Method2(2) are techniques for discovering Inv . Certificate generation is implemented for those methods. The method SAT uses a SAT solver to discover Inv , while Method2(1) and Method2(2) differ in their abstraction methods.

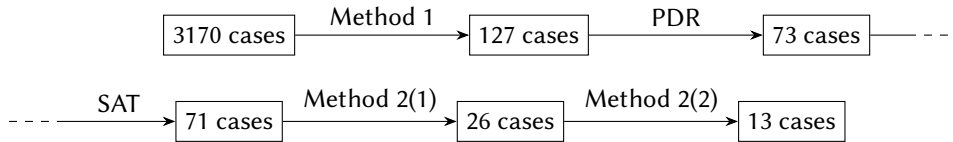


Figure 5: Journey of Solving $PCP[3,4]$

6. Conclusion and Future Work

We have been working for a complete resolution of $PCP[3,4]$ and came close, with only 13 instances remaining unsolved. To have these results accepted as trusted facts, we also aim to provide formal proofs using Isabelle/HOL for each instance, which has been achieved for the second method. Although both goals are yet to be fully achieved, we believe they are attainable as outlined below.

To solve the remaining 13 problems, we consider two possibilities. One is to solve these instances manually. We predict that most of the 13 problems do not have solutions, and providing ad-hoc proofs by humans might be the quickest way. The other possibility involves devising new variants of the methods in this paper or investing additional computational resources. Since the manual approach can also help gain deeper insights into individual instances and PCP itself, we would like to first aim for manual resolution.

Generating certificates for the first method is challenging because it uses an external Mixed Integer Programming (MIP) solver as a subroutine. Generating a certificate for the feasibility of an MIP is straightforward, as it merely requires providing a specific solution. However,

generating a certificate for infeasibility is more difficult. Cheung et al. (2017) [9] extended the existing MIP solver SCIP to output easily verifiable certificates in their own format. We believe that we can overcome this difficulty by converting these certificates into Isabelle/HOL code.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 19K11899 and 24K14891.

References

- [1] E. L. Post, A variant of a recursively unsolvable problem, *Bulletin of the American Mathematical Society* 52 (1946) 264–268.
- [2] L. Zhao, Tackling Post’s correspondence problem, in: *Computers and Games*, Springer Berlin Heidelberg, 2003, pp. 326–344.
- [3] R. J. Lorentz, Creating difficult instances of the post correspondence problem, in: *Computers and Games*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 214–228.
- [4] N. Verma, H. Seidl, T. Schwentick, On the complexity of equational horn clauses, 2005, pp. 337–352. doi:10.1007/11532231_25.
- [5] A. R. Bradley, Sat-based model checking without unrolling, in: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI’11*, Springer-Verlag, Berlin, Heidelberg, 2011, p. 70–87.
- [6] A. Omori, pcp-proof, <https://github.com/Mojashi/pcp-proof>, 2024.
- [7] L. Zhao, Pcp documents, 2002. URL: <https://webdocs.cs.ualberta.ca/~games/PCP>.
- [8] A. Omori, Unresolved problems, 2024. URL: <https://pcp-vis.pages.dev/gallery>.
- [9] K. K. H. Cheung, A. Gleixner, D. E. Steffy, Verifying integer programming results, in: F. Eisenbrand, J. Koenemann (Eds.), *Integer Programming and Combinatorial Optimization*, Springer International Publishing, Cham, 2017, pp. 148–160.