Tackling Post's Correspondence Problem

Ling Zhao
Department of Computing Science, University of Alberta

Post's correspondence problem (PCP) is a classic undecidable problem. Its theoretical unbounded search space makes it hard to judge whether a PCP instance has a solution or to find solutions if they exist. In this paper, we describe new methods used to efficiently find optimal solutions to difficult PCP instances, and to identify instances with no solution. We also provide strategies to create hard PCP instances that have long optimal solutions. These methods are practical approaches to this theoretical problem, and the experimental results present new insights into PCP.

Categories and Subject Descriptors: I.2.8 [**Heuristic methods**]: Heuristic search; I.2.3 [**Deduction**]: Disproof methods

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: empirical studies, Post's correspondence problem, PCP, single-agent search

1. INTRODUCTION

Post's correspondence problem (PCP for short) was first described by Emil L. Post in 1946 [1], and soon became a highly cited example of an undecidable problem in the field of computational complexity [2]. Bounded PCP is NP-complete [3]. PCP of 2 pairs was proven decidable [4], and recently a simpler proof using a similar idea was developed [5]. PCP of 7 pairs is undecidable [6]. Currently the decidability of PCP of 3 pairs to PCP of 6 pairs is still unknown.

As the property of undecidability shows, there exists no algorithm capable of solving all instances of PCP. Therefore, PCP is mainly discussed in the theoretical computer science literature, for example, to prove the undecidability of other problems. It is typical that PCP instances given in textbooks are trivial to solve, as contrast to the impossibility of solving the whole problem. This ironic situation as well as the research in the *Busy Beaver Problem* [7] motivated researchers to build

Address: Edmonton, Alberta, Canada, T6G 2E8. WWW: `http://www.cs.ualberta.ca/~zhao`
Email: `zhao@cs.ualberta.ca`

An earlier version of this paper appeared in the 3^{rd} *International Conference on Computers and Games* (*CG2002*), August 2002. This work has been supported by the Alberta Informatics Circle of Research Excellence (iCORE).

PCP solving programs [8; 9; 10]. Due to the theoretical property of PCP, clearly any PCP solving programs cannot be complete, for they can only solve some but not all PCP instances. Richard J. Lorentz first systematically studied the search methods used to solve PCP instances and the techniques to create difficult PCP instances [8]. Our work was motivated by his paper and can be regarded as an extension and the further development of his work.

In the past 20 years, search techniques in Artificial Intelligence (AI) have progressed significantly, as exemplified in the domains of single-agent search and two-player board games. A variety of search enhancements developed in these two domains have set good examples for solving instances of PCP (it is essentially a single-agent search problem), and some of the research can be directly migrated to PCP solvers after a few application-dependent modifications. On the other hand, the distinct characteristics of PCP cause special search difficulties, which has prompted us to develop methods based on new properties of PCP we discovered, and integrated them into the solver. This work resulted in quite a few methods that are important for solving PCP instances, and yields new insights into this traditionally theoretical problem.

This paper mainly discusses three directions for tackling Post's correspondence problem. The first two directions focus on finding optimal solutions to solvable instances efficiently and identifying unsolvable instances. We successfully applied several standard AI search techniques such as *forward pruning* and *bidirectional probing*, and raised four new domain-dependent methods, namely, *mask method*, *exclusion method*, *group method*, and *pattern method*. These techniques and new methods enabled us to solve many non-trivial PCP instances.

The third direction of our work is concerned with creating hard instances that have long optimal solutions. With the help of the methods developed in the above two directions, we built a strong PCP solver that discovered 199 hard instances whose optimal solution lengths are at least 100. Currently, we are holding the records for the instances with the longest optimal solutions in 4 non-trivial PCP subclasses.

The paper is organized as follows. We begin by introducing the definitions, notation, and some simple examples in Section 2. Then a variety of properties of PCP are discussed in Section 3. In Section 4, we explain isomorphism in PCP instances. Section 5 is concerned with how to apply AI search techniques to solving instances and Section 6 describes four domain-dependent methods based on special properties of PCP. Section 7 explains how to create difficult instances. Section 8 contains the experimental results and related discussions. As our approach to solve PCP instances is similar to the Busy Beaver Problem, we devote Section 9 to compare them. Finally, Section 10 provides conclusions and suggestions for future work.

2. WHAT IS POST'S CORRESPONDENCE PROBLEM

Definition 1 An *instance* of *Post's correspondence problem* is defined as a finite set of pairs of strings (g_i, h_i) $(i \in [1, s])$ over an alphabet Σ. A *solution* to this instance is a sequence of selections $i_1 i_2 \cdots i_n$ $(n \geq 1)$ such that the strings $g_{i_1} g_{i_2} \cdots g_{i_n}$ and $h_{i_1} h_{i_2} \cdots h_{i_n}$ formed by concatenation are identical.

The number of pairs in a PCP instance,[1] s in the definition, is called its *size*, and its *width* is the length of the longest string in g_i and h_i ($i \in [1, s]$). *Pair i* represents the pair (g_i, h_i), where g_i and h_i are the *top string* and *bottom string* respectively. *Solution length* is the number of selections in the solution. For simplicity, we restrict the alphabet Σ to $\{0, 1\}$, as we can always convert other alphabets to their equivalent binary format.

If an instance has at least one solution, then it is called *solvable*; otherwise, it is *unsolvable.* If an instance can be proven either solvable or unsolvable, it is *solved*; on the other hand, if no such a proof is found, it is *unsolved.* We will give examples in the next two subsections.

As solutions can be stringed together to create longer solutions, any solvable instance has an infinite number of solutions. Hence in this paper, we are only interested in *optimal solutions*, which have the shortest length over all solutions to an instance. Please note that a solvable instance may have more than one optimal solution. The length of an optimal solution is called the *optimal length.* If an instance has a fairly large optimal length compared to its size and width, we use the adjective *hard* or *difficult* to describe it.

An instance is *trivial* if it has a pair whose top and bottom strings are the same. It is obvious that such an instance has a solution of length 1. We call an instance *redundant* if it has two identical pairs, so removing either of the pairs will not influence the solving result. For simplicity, we assume the instances discussed in this paper are all nontrivial and non-redundant.

To conveniently represent subclasses of Post's correspondence problem, we use $PCP[s]$ to denote the set of all instances of size s, and $PCP[s, w]$ for the set of all instances of size s and width w. Given natural numbers s and w, $PCP[s]$ is an infinite set and $PCP[s, w]$ is a finite set, and the following relations hold:

$$PCP[s, w] \subset PCP[s] \subset PCP$$

The *hardest* instances in a finite subclass denote the instances with the longest optimal solutions among all solvable instances in the subclass. We define a function $\Phi(s, w)$ to represent the optimal length of the hardest instances in $PCP[s, w]$. From the undecidability of PCP, it is not difficult to show $\Phi(s, w)$ is a non-computable function, i.e., it grows faster than any computable function.

We use a matrix of 2 rows and s columns to represent an instance in $PCP[s]$, where string g_i is located at position $(i, 1)$ and h_i at $(i, 2)$. PCP (1) below is an example in $PCP[3, 3]$.

$$\begin{pmatrix} \mathsf{100} & \mathsf{0} & \mathsf{1} \\ \mathsf{1} & \mathsf{100} & \mathsf{00} \end{pmatrix} \tag{1}$$

2.1 Example of solving a PCP instance

The following describes a straightforward approach to solve PCP (1). First, we can only start at *pair 1*, since it is the only pair where one string is the other's prefix. Then we obtain this result:

[1]In the following, we use the name *instance* to represent PCP instance for brevity when no ambiguity in the context.

$$\text{Choose } \textit{pair 1}\text{:}\quad \begin{array}{l}\texttt{1}\underline{\texttt{00}}\\ \texttt{1}\end{array}$$

The portion of the top string that extends beyond the bottom one, which is underlined for emphasis, is called a *configuration.* If the top string is longer, the configuration is *in the top*; otherwise, the configuration is *in the bottom.* Therefore, a configuration consists of not only a string, but also its position information: *top* or *bottom.*

In the next step, it turns out that only *pair 3* can match this configuration, and the situation changes to:

$$\text{Choose } \textit{pair 3}\text{:}\quad \begin{array}{l}\texttt{100}\underline{\texttt{1}}\\ \texttt{100}\end{array}$$

Then there are two matching choices: *pair 1* or *pair 2.* By using the *mask method* described in Section 6.1, we can avoid trying *pair 2*, so *pair 1* becomes the only choice:

$$\text{Choose } \textit{pair 1}\text{:}\quad \begin{array}{l}\texttt{1001}\underline{\texttt{100}}\\ \texttt{1001}\end{array}$$

The selections continue until we find a solution:

$$\text{Choose } \textit{pair 1}\text{:}\quad \begin{array}{l}\texttt{10011}\underline{\texttt{00100}}\\ \texttt{10011}\end{array}\qquad \text{Choose } \textit{pair 3}\text{:}\quad \begin{array}{l}\texttt{1001100}\underline{\texttt{1001}}\\ \texttt{1001100}\end{array}$$

$$\text{Choose } \textit{pair 2}\text{:}\quad \begin{array}{l}\texttt{1001100100}\underline{\texttt{10}}\\ \texttt{1001100100}\end{array}\qquad \text{Choose } \textit{pair 2}\text{:}\quad \begin{array}{l}\texttt{1001100100100}\\ \texttt{1001100100100}\end{array}$$

After 7 steps, the top and bottom strings are exactly the same, which shows that the sequence of selections, *1311322*, forms a solution to PCP (1). By exhaustively searching all combinations of up to 7 selections of pairs, we can prove this solution is the unique optimal solution to the instance.

2.2 More examples

Some instances may have no solution. For example, the following instance can be proven unsolvable through the *exclusion method* discussed in Section 6.3.

$$\begin{pmatrix} \texttt{110} & \texttt{0} & \texttt{1} \\ \texttt{1} & \texttt{111} & \texttt{01} \end{pmatrix} \tag{2}$$

Some instances with very small sizes and widths can have amazingly long optimal solutions. PCP (3) is such an example whose optimal length is 206. Imagine if a computer performs a simple brute-force search by considering all possible combinations up to depth 206, how enormous the computation will be! This justifies why we utilize AI techniques and new methods based on special properties of PCP to prune useless search space and to improve the search efficiency.

$$\begin{pmatrix} \texttt{1000} & \texttt{01} & \texttt{1} & \texttt{00} \\ \texttt{0} & \texttt{0} & \texttt{101} & \texttt{001} \end{pmatrix} \tag{3}$$

Now let's take a look at PCP (4). It is clear that *pair 3* is the only choice in every step, and as a consequence, configurations will extend forever and the search process will never terminate if a brute-force search is employed. This example shows an unfortunate characteristic of some instances: *the search space is unbounded.* This special property suggests we should not solely rely on search to prove some instances unsolvable. Instead, clever ideas concerning proof of unsolvability are needed to deal with them.

$$\begin{pmatrix} 100 & 0 & 1 \\ 0 & 100 & 111 \end{pmatrix} \tag{4}$$

2.3 Configuration

Configuration is an important concept in solving PCP instances, and we give its relevant definitions to simplify explanation in the following sections.

Definition 2 A configuration is *empty* if its string is an empty string.

Definition 3 A configuration is *solvable* from an instance if it can lead to an empty configuration after a number of selections of pairs in the instance.

Definition 4 A configuration is *generable* from an instance if it can be generated from an empty configuration through a number of selections of pairs in the instance.

From the definitions above, PCP can be well mapped to a single-agent search problem: configurations are states in the search space and state transitions are driven by pairs chosen. The goal is to find a shortest non-empty solution path from the starting state (an empty configuration) to the goal state (an empty configuration too). It is evident that each configuration in the solution path must be generable and solvable.

Definition 5 Let c be a configuration, then its *reversal*, denoted as c^R, is generated by reversing its string, and its *turnover*, denoted as $\bar{c}$, is generated by flipping its position (from the top to bottom or vice versa).

3. PROPERTIES OF PCP

We have already mentioned three properties of PCP above: *undecidability*, *infinite number of solutions* and *unbounded search space.* In this section, more properties will be discussed.

3.1 Reversal properties

Definition 6 Let S be a string, then its *reversal*, denoted as S^R, is S written backwards.

Definition 7 Let P: (g_i, h_i) $(i \in [1, s])$ be an instance, then its *reversal*, denoted as P^R, is (g_i^R, h_i^R) $(i \in [1, s])$.

Suppose we have a solution $i_1 i_2 \cdots i_n$ to instance P. It is easy to see that $i_n i_{n-1} \cdots i_1$ is a solution to P^R. Essentially, P and P^R are equivalent, as clarified in the following lemma:

Lemma 1 *Let P be an instance. P has the same solvability as P^R in the sense that it has a solution if and only if P^R has, and it has the same number of optimal solutions and the same optimal length as those of P^R.*

We can go one step further: if configuration c is generable from instance P through a sequence of selections $i_1 i_2 \cdots i_j$, then configuration $\bar{c}^R$ is solvable from instance P^R through selections $i_j i_{j-1} \cdots i_1$. So the following lemma is obtained:

Lemma 2 *Let P be an instance and c a configuration. c is generable from P if and only if $\bar{c}^R$ is solvable from P^R.*

This simple lemma is an important property underlying the *mask method* and *exclusion method.*

3.2 Unsolvability properties

The following lemmas can be used to easily identify some types of unsolvable instances:

Lemma 3 *A solvable instance must have one pair where one string is the other's proper prefix and another pair where one string is the other's proper postfix.*

Lemma 4 *A solvable instance must have one pair whose top string is longer than the bottom one and another pair whose bottom string is longer.*

Lemma 5 *Let $x \in \{0,1\}$. If in an instance, the top string in every pair has no fewer x's than the bottom string, then all pairs whose top strings contain strictly more x's than their counterparts can be safely removed without changing the solvability of the instance. The same rule applies when the roles of the top and bottom strings are reversed.*

Lemma 3 ensures that the selections can start and end somewhere in a solvable instance. Lemma 4 is on the length balance, and Lemma 5 is on the element balance. The latter lemma can help remove useless pairs in an instance, and to an extreme where all pairs are removed, the instance can thus be proven unsolvable.

Three types of filters, namely *prefix/postfix filter*, *length balance filter*, and *element balance filter*, which were first mentioned in [8], are the direct applications of the above three lemmas respectively, and can be used to filter out unsolvable instances in our experiments. It is amazing that during the experiments on several PCP subclasses with small sizes and widths, most of instances can be filtered out using this method (see Tables 4, 5 and 6).

4. ISOMORPHISMS AMONG PCP INSTANCES

Consider the following four types of transformations on an instance:

Reversal:	change every string to its reversal.
Upsidedown:	interchange top and bottom strings in every pair.
Complement:	replace all 0's with 1's and vice versa in every string.
Pair Reordering:	reorder pairs in the instance.

It is not hard to see that through any combination of the above four transformations on an instance, we obtain a new instance that is equivalent to the original one in the sense that a solution of one instance can be easily used to deduce a solution to the other and vice versa. We call them *isomorphic* instances.

s ^{w}	1	2	3	4	5	6
1	2	28	688	870	2,912	11,968
2	2	868	32,072	723,088	13,543,712	233,747,008
3	0	24,360	5,905,200	605,304,480	5.3×10^{10}	3.9×10^{12}
4	0	657,720	1,060,733,520	5.7×10^{11}	2.0×10^{14}	6.1×10^{16}

Table 1. Cardinality of 24 PCP subclasses (s is size and w is width)

For an instance in $PCP[s]$, there will be up to $2^3 \cdot s!$ isomorphic instances (including itself) in the worst case. So an instance in $PCP[3]$ may have as many as 48 different isomorphisms! For a symmetric instance such as PCP (5), there are fewer distinct isomorphic instances.

$$\begin{pmatrix} 111 & 0 & 00 \\ 1 & 101 & 1 \end{pmatrix} \tag{5}$$

4.1 Cardinality of PCP subclasses

This subsection deals with calculating the cardinality of PCP subclasses. First, we need to define the cardinality of three types of sets.

Definition 8 *$Str(w)$ is the set of any string whose length is in $[1, w]$. $|Str(w)|$ is the cardinality of $Str(w)$.*

Definition 9 *$Pair(w)$ is the set of any string pair whose both strings are in $Str(w)$ but not identical. $|Pair(w)|$ is the cardinality of $Pair(w)$.*

Definition 10 *The cardinality of $PCP[s, w]$, denoted as $|PCP[s, w]|$, is the number of all nontrivial and non-redundant instances in $PCP[s, w]$.*

The following equations compute $|PCP[s, w]|$ for any given s and w.

$$\begin{aligned}
|Str(w)| &= 2^1 + 2^2 + \cdots + 2^w = 2^{w+1} - 2 \\
|Pair(w)| &= |Str(w)| \times (|Str(w)| - 1) \\
|PCP(s, w)| &= P^s_{|Pair(w)|} - P^s_{|Pair(w-1)|} \\
\text{where } P^s_n &= n \times (n-1) \times \cdots \times (n-s+1)
\end{aligned}$$

The cardinality of 24 PCP subclasses is given in Table 1 where exact values are given for numbers up to 10^{10} and approximate values for larger numbers.

4.2 Normalization

In order to avoid excessive redundant work, it is important to eliminate isomorphic instances when scanning all instances in finite PCP subclasses. We use a normalization process to remove isomorphic instances.

Definition 11 *Let $S = s_1 s_2 \cdots s_n$ be a string over the alphabet $\{0, 1\}$. Its score, denoted as $Score(S)$, is $\sum_{i=1}^{n}(3^i \cdot (s_i + 1))$.*

Definition 12 *Let $P = (S_1, S_2)$ be a string pair. Its score, denoted as $Score(P)$, is $(Score(S_1), Score(S_2))$.*

We order $Score(P)$ lexicographically, that is, given two string pairs $P_1 = (S_1, S_2)$ and $P_2 = (T_1, T_2)$, $Score(P_1) > Score(P_2)$ if and only if $Score(S_1) > Score(T_1)$ or $Score(S_1) = Score(T_1)$ and $Score(S_2) > Score(T_2)$. Similarly, we can also order PCP instances of the same size lexicographically.

Definition 11 ensures that the mapping function is injective, so that any different strings must have different scores. Note that other mapping functions can work here too, and we choose a simple one to illustrate the idea. Given an instance, it is always possible to find the unique instance from all of its isomorphisms that has the highest score based on the lexicographic order, and this one is called the *standard form*. When scanning all instances of a PCP subclass, any instance different from its standard form will be eliminated. Thus, we can generate all instances of a finite PCP subclass with no isomorphic instances contained.

4.3 Results of removing isomorphic instances

We conducted experiments to determine the exact number of non-isomorphic instances in 11 PCP subclasses that are manageable. The results are presented in Table 2. In the table, the value of *ratio* is computed as the total number of instances in a subclass divided by the number of non-isomorphic instances in it; s denotes the size of the corresponding PCP subclass.

	total number	non-isomorphic	*ratio*	*ratio*/s!
$PCP[2,1]$	4	1	4.000	2.000
$PCP[2,2]$	868	76	11.421	5.711
$PCP[2,3]$	32,072	2,270	14.129	7.064
$PCP[2,4]$	723,088	46,514	15.546	7.773
$PCP[2,5]$	13,543,712	856,084	15.821	7.910
$PCP[2,6]$	233,747,008	14,644,876	15.961	7.981
$PCP[3,2]$	24,360	574	42.439	7.073
$PCP[3,3]$	5,905,200	127,303	46.386	7.731
$PCP[3,4]$	650,304,480	13,603,334	47.805	7.967
$PCP[4,2]$	657,720	3,671	179.166	7.465
$PCP[4,3]$	1,060,733,520	5,587,598	189.837	7.910

Table 2. Number of non-isomorphic instances in 11 PCP subclasses

As the table shows, for PCP subclasses of $PCP[s]$, the value *ratio* quickly approaches to $8 \cdot s!$ as the width increases. Intuitively, when the width becomes larger, the chance for an instance being identical to one of its isomorphisms becomes smaller. For a similar reason, when the width is fixed, the value *ratio*/s! approaches 8 gradually as the size increases, as shown experimentally from the results of subclass $PCP[2,3]$, $PCP[3,3]$ and $PCP[4,3]$. The total number of instances in a finite subclass can be computed from the formulas given in Section 4.1, while no analytical method is known yet to calculate the exact number of non-isomorphic instances in a subclass $PCP[s, w]$. Thus the ratio $8 \cdot s!$ can be used to estimate the number.

All instances in the 11 PCP subclasses were fully scanned for their solvability, and the details are given in Sections 8.2 and 8.3.

5. APPLYING AI SEARCH TECHNIQUES

As PCP can be formalized as a search problem, we can utilize those existing AI search techniques that have been successfully applied to many theoretically decidable problems such as two-player board games and puzzles.

Depth-first iterative-deepening and cache table were first used to deal with PCP instances by Richard J. Lorentz [8]. We refined the ideas and they are summarized in the following. In addition, we first applied forward pruning and bidirectional probing to improve the search efficiency of solving PCP instances.

5.1 Depth-first iterative-deepening

A depth-first iterative-deepening algorithm based on A^* algorithm is the first known algorithm able to effectively solve the 15-puzzle, which has been proven asymptotically optimal in time and space complexity for a class of tree search problems [12]. This algorithm can be briefly described as follows: first, a starting *depth threshold* is set and the state space is searched to the threshold. If no solution is found, the threshold is augmented by a fixed *depth increment* and a new search is performed. The above process is repeated until solutions are discovered or the depth threshold reaches a *final threshold*. When the whole search process finishes, either optimal solutions have been found, or it is proven that no solution exists up to the final depth threshold.

The depth increment is an important parameter. If the increment is too small, the solver may do too much redundant work as all nodes visited in an iteration will be revisited in the next iteration. If the increment is too large, the search process may fall into the space where most nodes have larger depths than the optimal length, which can result in a significant loss of efficiency. In our experiments, we use 20 as the depth increment.

5.2 Cache table

During the search process, it is possible to generate a configuration that has been encountered before, where the depth of this newly generated configuration is longer than that of the old one. Since we focus on optimal solutions, whenever this case emerges, we can simply prune the new configuration.

To check revisited configurations, we employ a cache table which is very similar to the transposition tables widely used in game-playing programs. A cache table is a block of specially allocated memory space used to store a number of configurations that have been visited before. When a newly generated configuration hits the cache, we can determine whether it can be pruned by comparing its depth with the depth of the one in the cache table. If it cannot be pruned, the cache should be added or updated if applicable. To facilitate the identification process, we use a hash function to map configurations to entries in the cache table. The size of the cache table and the hash function need to be tuned for satisfactory performance.

5.3 Forward pruning

Similar to heuristic search algorithms such as A^* and IDA^* [13; 12], a heuristic function of a configuration can be calculated and used as a lower bound for the solution length (for an unsolvable instance, its solution length is defined to be infinity). A heuristic value of a configuration is an estimate of the number of additional selections needed to reach a solution. When the heuristic value of one configuration added to its depth exceeds the current depth threshold in the iterative deepening search, this configuration definitely has no hope of reaching a solution within that threshold. Hence we can reject it even if its depth is still far away from the threshold. Since the heuristic function never overestimates, the pruning is safe and does not affect the optimality of solutions.

One simple heuristic value of a top (bottom) configuration can be calculated by its length divided by the maximum length difference of all pairs whose bottom (top) string is longer. This heuristic is based on the length balance, and similarly, we can calculate heuristics on the balance of elements 0 or 1.

More complex heuristics can be developed analogous to the pattern databases used to efficiently solve instances of the 15-puzzle [14]. Before a search is performed on an instance, we can pre-compute matching results for some strings as the prefixes of configurations, and use them to calculate a more accurate estimate of the solution length than the simple heuristics during the search.

5.4 Bidirectional probing

As shown before, an instance is equivalent to its reversal in terms of solvability. But the search difficulty to solve these two forms may be amazingly different, as shown experimentally in Section 8.1. Hence, we use a probing scheme to decide which search direction is more promising. Initially we set a *comparison depth* (40 in the implementation), and two search processes are performed for the original instance and its reversal to the comparison depth respectively. A comparison of the number of visited nodes in both searches gives a good indication about which direction is easier to explore. The solver then chooses to solve the one with the smaller number of visited nodes. As the branching factors in most instances are quite stable, this scheme worked very well in our experiments.

As a complement, we would like to point out that in the four types of transformations to generate isomorphic instances (see Section 4), only reversal can make a big difference during the solving process.

6. NEW DOMAIN-DEPENDENT METHODS

Although AI search techniques help the PCP solver find solutions to many difficult instances, domain-dependent methods are required to tackle solvable instances more effectively and to prove the unsolvability of hard instances.

Intuitively, solvable and unsolvable instances should be treated separately, but for PCP instances, these methods are valuable to both types of tasks, and we hence discuss them together in this section. These new methods are all specifically based on the characteristics of PCP, and can be categorized into two classes.

The first class is concerned with pruning configurations. If a configuration can be proven unsolvable through simple rules, it can be safely pruned without generating its descendants, and by this means, a great portion of search space can be saved. This may result in a significant improvement of the solving time for a solvable instance, and lead to an unsolvability proof for an unsolvable instance by reducing its unbounded search space to a finite one. We formalized two methods namely *mask method* and *pattern method* to find such rules that any configuration satisfying them can be pruned.

The second class is for simplifying instances. By analyzing a specific instance, we may find special structures showing the instance can be simplified by removing some of its pairs or replacing a substring with a simpler one. This simplification significantly reduces the search work, and makes it possible to prove some instances unsolvable. We invented *exclusion method* and *group method* to realize this idea.

6.1 Mask method

The mask method deals with pruning all configurations in the top or in the bottom by proving all configurations in one position are unsolvable. At the beginning, we need to introduce the important concept of critical configuration:

Definition 13 A *critical configuration* in an instance is a non-empty configuration that can be fully matched by a pair of the instance such that the configuration becomes empty, or be turned upside-down such that the position is flipped from top to bottom or vice versa.

Critical configurations are critical because they constitute an indispensable step for a configuration in general to reach a solution. If a top configuration is solvable, it must reach a top critical configuration before transferring to an empty configuration (being solved). Conversely, if no top critical configurations can occur in any solution path, all top configurations generated are unsolvable and can be safely pruned: the instance has a *top mask*. Similarly, a *bottom mask* means there is no hope of reaching a solution once the configuration is in the bottom.

To check if an instance has masks, we need to find all critical configurations, and then test if they are possible in a solution path or equivalently, if they are generable and solvable. The first step to obtain all critical configurations can be simply done by enumeration. To test if a configuration is generable can be converted to test if the turnover of its reversal is solvable according to Lemma 2. Note that the process to judge whether any given configuration is generable or solvable is undecidable, and in our implementation, a fixed depth was set to limit the search. Because of the limitation, it is possible that an instance has masks while we could not find them out.

The following explains how these steps work to discover the top mask in PCP (6), whose reversal is PCP (7).

$$\begin{pmatrix} \texttt{01} & \texttt{00} & \texttt{1} & \texttt{001} \\ \texttt{0} & \texttt{011} & \texttt{101} & \texttt{1} \end{pmatrix} \tag{6}$$

$$\begin{pmatrix} \texttt{10} & \texttt{00} & \texttt{1} & \texttt{100} \\ \texttt{0} & \texttt{110} & \texttt{101} & \texttt{1} \end{pmatrix} \tag{7}$$

At first, we need to find all top critical configurations. Since they could either be fully matched or turned over by a pair, any matched pair must have a longer bottom string. In this instance, the matched pair could only be *pair 2* or *pair 3*. With a little computation, we can find that only one top critical configuration exists in PCP (6), i.e. 10, which can be fully matched by *pair 3*. Since this configuration can be fully matched, it is solvable. In the next step, it is needed to check whether 10 in the top if generable from PCP (6), or equivalently, whether 01 in the bottom is solvable from PCP (7). But in PCP (7), this configuration cannot choose any pair to match. Thus, no top critical configurations that are both generable and solvable exist in PCP (6), and the instance has a top mask.

For some instances, the mask method is an effective tool to find their optimal solutions. Take PCP (6) for example. It has a top mask, so we forbid the use of *pair 1* at the beginning, and can only choose *pair 3*, which helps quickly find the unique optimal solution of length 160. If we did not know this fact, *pair 1* could be chosen as the starting pair and a huge useless search space would have to be explored before ascertaining that its optimal length is 160.

The mask method can also prove the unsolvability of many instances. For example, if an instance has both top and bottom masks, it certainly has no solution.

The step to prove critical configurations not generable can be strengthened by the *GCD* (Greatest Common Divisor) *rule*: if the length differences of all pairs have a greatest common divisor d, then the length of any generable configuration must be a multiple of d. Consider the following PCP (8) as an example. The GCD of all length differences is 2.

$$\begin{pmatrix} \texttt{111} & \texttt{001} & \texttt{1} \\ \texttt{001} & \texttt{0} & \texttt{111} \end{pmatrix} \tag{8}$$

Although in PCP (8), we can find a bottom critical configuration of 0 which can be turned upside-down by *pair 2*, it is not generable because its length is not a multiple of 2. As a result, this instance has a bottom mask, revealing the starting selection must be *pair 2*. Finally, with a few steps of enumeration, we can prove PCP (8) has no solution.

Note that the above example uses the difference of length, and similarly we can use the difference of the number of elements 0 or 1.

6.2 Pattern method

If all possible paths a configuration with a prefix α generates always lead to configurations with prefix α (no empty configuration occurs in the paths), then clearly any configuration having such a prefix is unsolvable. This observation essentially comes from the goal to shrink configurations to an empty configuration. If there is a substring that will unavoidably occur, it is impossible for configurations to transfer to an empty configuration. The following example illustrates how this method is effective to prove PCP (9) has a prefix pattern of 11 in the top, and as a result, PCP (9) is unsolvable.

$$\begin{pmatrix} \texttt{011} & \texttt{01} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{100} \end{pmatrix} \tag{9}$$

Given a top configuration of $11A$, where A can be any string, the next selection can only be *pair 1*. Thus a new top configuration $1A011$ is obtained. Now let's focus on how the substring 0 right after the A is matched. The matched 0 can be supplied either by the only 0 in the bottom string of *pair 2*, or by the last 0 in the bottom string of *pair 3*. Whichever it is, after 0 is matched the substring 11 right behind it will inevitably become the prefix of a new configuration. So a top configuration $11A$ will definitely transfer to another top configuration $11B$ after a number of steps. The prefix cannot be removed, showing any top configuration with a prefix of 11 is unsolvable. The procedure to find a prefix in PCP (9) is illustrated in Fig. 1.

$$11A \Longrightarrow 1A0 \,\vdots\, 11 \Longrightarrow 11B$$

Fig. 1. Deduction of a prefix pattern in PCP (9)

The dotted vertical line in the figure partitions configurations into left and right parts. According to the matching rules of PCP, its left part from the line must be matched exactly during a number of selections. Therefore, the dotted vertical line works as a border: matching must stop at one side of it and continue on the other side; no substring can be matched across the line.

It can be proven that PCP (9) has a bottom mask. So with the help of the prefix pattern derived above, we can exhaustively try all possible selections and prove PCP (9) is unsolvable.

It is quite intuitive to discover the pattern in PCP (9), yet to find similar patterns in other instances may not be simple. For example, Fig. 2 illustrates the procedure to detect the prefix pattern of 000 in the top in PCP (10), which is indispensable for the unsolvability proof of this instance.

$$\begin{pmatrix} 01 & 0 & 00 \\ 0 & 100 & 10 \end{pmatrix} \tag{10}$$

$$000A \Longrightarrow A0 \,\vdots\, 10101 \Longrightarrow \begin{cases} 1010 \,\vdots\, 1\ B_1 01 \Longrightarrow 1\ B_1 0 \,\vdots\, 10000 \Longrightarrow 100 \,\vdots\, 000C_1 \Longrightarrow 000D_1 \\ 1010 \,\vdots\, 1\ B_2\, 0 \Longrightarrow 1\ B_2\, 00 \,\vdots\, 000 \Longrightarrow 000C_2 \\ 1010 \,\vdots\, 1\ B_3 00 \Longrightarrow 1\ B_3 000 \,\vdots\, 000 \Longrightarrow 000C_3 \end{cases}$$

Fig. 2. Deduction of a prefix pattern in PCP (10)

6.3 Exclusion method

The exclusion method is utilized to detect pairs that will never be used when the selections start at some pair. The exclusion comes from the fact that if any combination of certain pairs cannot generate a configuration that can be matched by a specific pair, then that pair is useless and can be safely removed. PCP (11) is such an example.

$$\begin{pmatrix} 1 & 0 & 101 \\ 0 & 001 & 1 \end{pmatrix} \tag{11}$$

If we start from *pair 2*, then the following selections can only be *pair 1* or *pair 2*. The proof can be separated into three steps.

(1) If only *pair 1* and *pair 2* are allowed, all generated configurations stay in the bottom.

(2) Any combination of the bottom strings of these two pairs cannot have a substring of 101, the top string of *pair 3*. Thus when a configuration generated by these two pairs has its length of at least 3, *pair 3* has no chance to be selected.

(3) The only bottom configuration with length less than 3 that can be matched by *pair 3* is 10, yet it cannot be generated by selections of *pair 1* and *pair 2*.

Therefore, we only need to deal with an instance consisting of *pair 1* and *pair 2* after selections start at *pair 2*. This new instance never leads to a solution since the length of configurations it generates never decreases. Hence starting at *pair 2* is hopeless. Similarly, we can prove starting at *pair 3* is also unsolvable by excluding *pair 2*. As we can find no other starting pairs, PCP (11) is proven unsolvable.

6.4 Group method

If any occurrence of a substring in configurations can only be entirely matched during one selection of pairs, instead of being matched through several selections, we can consider the substring as an undivided entity, or a *group*. In other words, if the first character in a group is matched during one selection, all others in the group have to be matched in the same selection. The group method is utilized to detect such groups and help to simplify instances. Consider PCP (12), where the substring 10 forms a group.

$$\begin{pmatrix} 011 & \mathbf{10} & 0 \\ 1 & 0 & 0\mathbf{10} \end{pmatrix} \tag{12}$$

$$\begin{pmatrix} 011 & g & 0 \\ 1 & 0 & 0g \end{pmatrix} \tag{13}$$

$$g = 01$$

The substring 10 can be inserted into configurations through the bottom string in *pair 3*, and then can be matched by 10 in the top string in *pair 2*. If we consider an instance consisting of only *pair 2* and *pair 3*, then it is not difficult to find out that whenever there is a substring 10 occurring in a configuration, this substring must be entirely supplied by a selection of *pair 3* and can only be matched by *pair 2*. Therefore, we can use a new symbol g to represent the group 10, and the instance will be simplified to PCP (13).

If only *pair 2* and *pair 3* are taken into consideration, the configurations they generate will stay in the bottom and have their lengths non-decreasing. So these configurations lead to no solution. As the new symbol g cannot be matched by 0 or 1, it is easy to see that *pair 1* can be excluded when selections start at *pair 3*. Since no other possible starting pair exists, PCP (12) is unsolvable.

7. CREATING DIFFICULT INSTANCES

A strong PCP solver enhanced by the methods discussed in the previous two sections is essential for creating many difficult instances. Besides, the hard instances that we found attracted us to find their solutions efficiently, and those unsolved instances were intriguing to us to come up with new ideas. Thus, the three directions we are working on are interrelated, as shown in Fig. 3.

The task of creating difficult instances can be further categorized into two schemes: random search and systematic search.

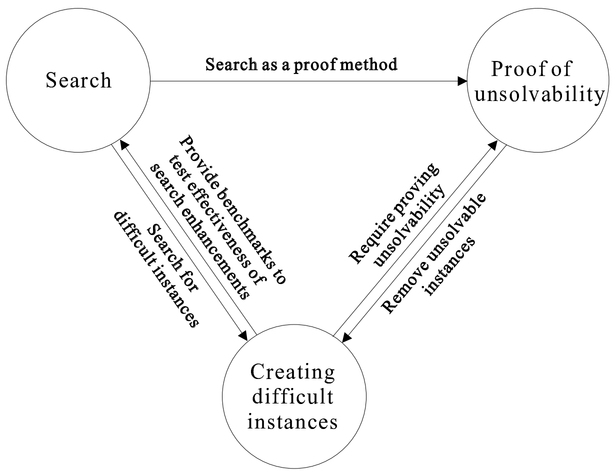
Search
Search as a proof method
Proof of unsolvability
Provide benchmarks to test effectiveness of search enhancements
Search for difficult instances
Require proving unsolvability
Remove unsolvable instances
Creating difficult instances

Fig. 3. Relations between three research directions in PCP

7.1 Random search for hard instances

A random instance generator plus a PCP solver is a straightforward means of discovering interesting instances. Implemented with several search enhancements and various methods that help to prove instances unsolvable, the program can quickly stop searching unpromising instances and find the optimal solutions to solvable instances faster.

During the search process of an instance, we use three factors as stopping conditions if no solution is found, which helps to terminate the solving process even in an infinite search space. They are the final depth threshold, the number of visited nodes, and the number of *cutoff* nodes (nodes pruned by the cache table). Using the number of visited nodes as a stopping condition forces the search process to treat every instance equally, avoiding the situation of frequently getting stuck in instances that have a large branching factor but no solution. The use of cutoff nodes as a stopping condition comes from the observation that most of the hard instances we collected do not have a large number of cutoff nodes (the largest is only 28,972).

7.2 Systematic search for hard instances

As the random search scheme randomly chooses instances to consider, the chance of finding difficult instances is still dependent on luck, so a systematic approach seems more convincing to demonstrate the strengths of a PCP solver. If all instances in a specific PCP subclass are examined, they may be completely solved and lots of hard instances including the hardest one in this subclass can be discovered. Even if we cannot solve all of them, the unsolved instances may stimulate us to find better approaches to deal with them. As all non-isomorphic instances in finite PCP subclasses can be generated through the process described in Section 4, we used our PCP solver to scan all instances in the 11 PCP subclasses listed in Table 2. The results are shown in Sections 8.2 and 8.3.

7.3 New PCP records

The random and systematic search schemes for creating difficult instances helped us set the *hardest* instance records in 4 non-trivial PCP subclasses shown in Table 3. The records in subclasses $PCP[3, 4]$ and $PCP[4, 3]$ were found by using the systematic search scheme and those in $PCP[3, 5]$ and $PCP[4, 4]$ were obtained through the random search scheme. Note the record in $PCP[3, 3]$ was independently discovered by Richard J. Lorentz and Johannes Waldmann. More information is provided on the web sites [15; 16]. These records show experimentally the difficulty of solving instances of an undecidable problem that have very small sizes and widths.

subclass	hardest instance known	optimal length	number of optimal solutions
$PCP[3,3]$	$\begin{pmatrix} 110 & 1 & 0 \\ 1 & 0 & 110 \end{pmatrix}$	75	2
$PCP[3,4]$	$\begin{pmatrix} 1101 & 0110 & 1 \\ 1 & 11 & 110 \end{pmatrix}$	252	1
$PCP[3,5]$	$\begin{pmatrix} 11101 & 1 & 110 \\ 0110 & 1011 & 1 \end{pmatrix}$	240	1
$PCP[4,3]$	$\begin{pmatrix} 111 & 011 & 10 & 0 \\ 110 & 1 & 100 & 11 \end{pmatrix}$	302	1
$PCP[4,4]$	$\begin{pmatrix} 1010 & 11 & 0 & 01 \\ 100 & 1011 & 1 & 0 \end{pmatrix}$	256	1

Table 3. Records of hardest instances in 5 PCP subclasses

8. EXPERIMENTAL RESULTS AND ANALYSIS

We implemented most of the methods we discussed in Sections 5 and 6 except the pattern method and group method, because we could not find a general way to automate them. The program was written in C++ under Linux platform.

With other normal search enhancements and programming techniques incorporated, the final version of our PCP solver achieved a search speed of 1.38×10^6 nodes per second on a machine with a Pentium III 600MHZ processor and 128M RAM.

8.1 Results of solving methods

In this subsection, all experiments were performed on 200 hard instances. 199 instances have optimal lengths at least 100 and were collected from 4 PCP subclasses through the methods described in Section 7; the remaining test case is the hardest instance known in $PCP[3, 3]$, as shown in Table 3. The average branching factor of 200 test instances is only 1.121 after all enhancements were incorporated. Such a small branching factor makes it feasible to find a solution with length even over 300.

We cannot provide a quantitative evaluation of the improvements derived from the mask method and exclusion method, since they are essential to solve certain types of instances. It is more proper to comment that these two methods would help to prune a huge useless search space in some cases, e.g., reducing an infinite search space to a finite one.

Bidirectional probing is also crucial to solve some hard instances. PCP (14) with an optimal length of 134 is such an example. Up to depth 40, searching this instance directly is more than 15,000 times harder than searching its reversal in terms of the number of visited nodes. Consider that searching to depth 40 has already made such a big difference, if searching to depth 134, the difference will explode exponentially. Thus, it becomes unrealistic to solve this instance when the wrong direction is chosen. What's more, proving an instance unsolvable directly may be much harder than proving its reversal directly. These results clearly demonstrate how important bidirectional probing is.

$$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{1} & \texttt{0} \\ \texttt{0} & \texttt{101} & \texttt{00} & \texttt{11} \end{pmatrix} \tag{14}$$

One nice strength of the above three methods is that they are all done before a deep search is performed and they introduce negligible overhead to the solver.

We implemented two types of admissible heuristic functions to prune hopeless nodes in the forward pruning. The first one is based on the length balance, and the second is based on the element balance, as Section 5.3 described. Three separate experiments were conducted on forward pruning, namely, using the first type of heuristic, using the second type and using both types. The results show that only using the heuristic on the length balance achieved the best performance.

We tried to compare the improvement achieved by forward pruning with the situation when no pruning is done, but we could not finish the task since it would take too much time. PCP (15) is an illustrative example. The solver spent 14,195 seconds to solve this instance when no pruning was used, compared to merely 5.2 seconds when the length balance heuristic was employed. This is a 2730-fold speedup in solving time!

$$\begin{pmatrix} \texttt{11011} & \texttt{110} & \texttt{1} \\ \texttt{0110} & \texttt{1} & \texttt{11011} \end{pmatrix} \tag{15}$$

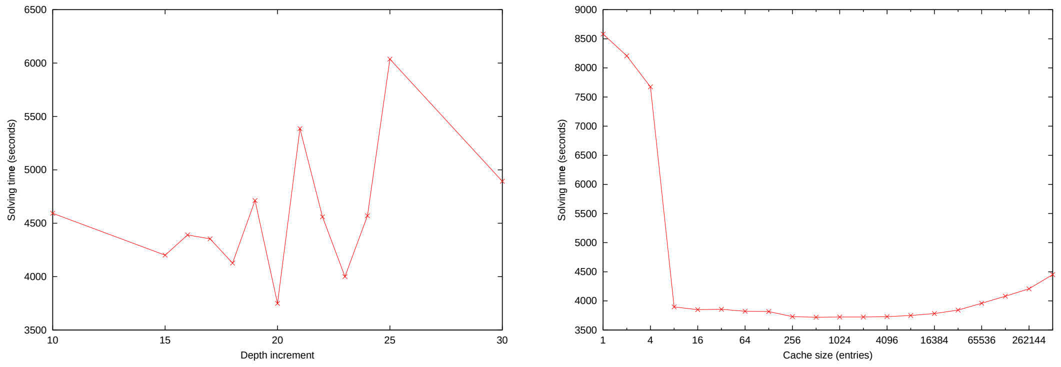

Fig. 4. Solving time with respect to different depth increments and cache sizes

We also did experiments on search parameters, as shown in Fig 4. The experimental results show that when the depth increment in the iterative deepening is

20, the solving time is minimal over our test set. The result of the experiments on the cache size, however, is a little surprising. It shows that from 8 entries to 65536 entries, the solving time is quite stable. We suspect this phenomenon is largely dependent on the test instances chosen. By default, a large table is employed because it is essential to prove some instances unsolvable.

8.2 Results of scanning PCP subclasses

We first scanned 8 PCP subclasses that are easy to handle, and all instances in these subclasses were completely solved. All isomorphic instances have been removed, and the results are shown in Table 4. This table also provides a statistical view on the effectiveness of our unsolvability proof methods, and illustrates how small the percentage of solvable instances in these subclasses is.

PCP subclass	total number	after filter	after mask	after exclusion	solvable instances	unsolvable instances	$\Phi(s,w)$
$PCP[2,1]$	1	0	0	0	0	1	-
$PCP[2,2]$	76	3	3	3	3	73	2
$PCP[2,3]$	2,270	51	31	31	31	2,239	4
$PCP[2,4]$	46,514	662	171	166	165	46,349	6
$PCP[2,5]$	856,084	9,426	795	761	761	855,323	8
$PCP[2,6]$	14,644,876	140,034	3,404	3,129	3,104	14,641,772	10
$PCP[3,2]$	574	127	67	61	61	513	5
$PCP[4,2]$	3,671	1,341	812	786	782	2,889	5

Table 4. Solving results of 8 PCP subclasses

One special phenomenon we observed is that the hardest instances of several PCP subclasses with size 2 can all be represented in the following form:[2]

$$\begin{pmatrix} 1^n0 & 1 \\ 1 & 01^n \end{pmatrix}$$

It is not hard to prove the optimal length of this kind of instances is $2n$. As conjectured in [8], instances having such a form might always be the hardest ones in $PCP[2, n+1]$ in general. If the conjecture is true, it will lead to a much simpler proof that $PCP[2]$ is decidable than the existing one [4]. Our experimental results support the conjecture in the cases from $PCP[2,2]$ to $PCP[2,6]$.

We used the systematic method to further examine three PCP subclasses that are much harder to conquer. Table 5 summarizes the results from $PCP[3,3]$, and Table 6 in the next subsection shows the results from $PCP[3,4]$ and $PCP[4,3]$.

In Tables 5 and 6, an instance removed by the exclusion method may still have solutions, but it cannot have a *valid* solution where all pairs of the instance are used. Since the result of solving such an instance is identical to the combinations of the results from instances with smaller sizes, we stop the further processing. Similarly, instances removed by the element balance filter may also have invalid solutions, but these solutions are of no interest to us.

[2]Of the three hardest instances of $PCP[2,2]$, only one instance can be represented in this form.

Total number	127,303
After filter	8,428
After mask	2,089
After exclusion	2,002
Solvable instances	1,968
Unsolvable instances	34
$\Phi(3,3)$	75

Table 5. Scanning results of subclass $PCP[3,3]$

Our PCP solver solved all but 33 instances in $PCP[3,3]$. We proved 32 instances of them unsolvable by hand using the methods discussed in Section 6. To prove these instances unsolvable requires several tricks, but they were not implemented as they are too specific to several instances and are hardly generalized to solve a considerable number of instances.

The remaining instance is PCP (16). It was solved by Mirko Rahn through a new method that generalizes the pattern method discussed in this paper [17]. Thus all instances in PCP[3,3] were settled down, and we have $\Phi(3,3) = 75$.

$$\begin{pmatrix} 110 & 1 & 0 \\ 1 & 01 & 110 \end{pmatrix} \tag{16}$$

8.3 Results of creating difficult instances

We scanned all instances in $PCP[3,4]$ and $PCP[4,3]$ to discover hard instances. The results are summarized in Table 6.

	$PCP[3,4]$	$PCP[4,3]$
Total number	13,603,334	5,587,598
After filter	902,107	1,024,909
After mask	74,881	275,389
After exclusion	65,846	266,049
Solvable instances	61,158	249,493
Unsolvable instances	1,518	2,633
Unsolved instances	3,170	13,923
Hard instances	5	72
$\Phi(s,w)$	≥ 252	≥ 302

Table 6. Scanning results of subclass $PCP[3,4]$ and $PCP[4,3]$

We used the following three conditions to limit the search:

(1) search depth ≤ 400
(2) number of visited nodes $\leq$ 180,000,000
(3) number of cutoff nodes $\leq$ 5,000,000

The scanning process took about 30 machine days to finish and resulted in the discovery of 77 hard instances whose optimal lengths are at least 100. At the same time, more than 17,000 instances remain unsolved to the solver, and it becomes impossible to check such a large quantity of instances manually. Although most of these unsolved instances may have no solution, it is still likely that they contain some extremely difficult solvable instances. Thus, these instances are left for future work, waiting for some new search and disproof methods.

Using the random approach to search for difficult instances, we successfully discovered 21 instances in $PCP[3,5]$ and 101 instances in $PCP[4,4]$. Their optimal lengths are all at least 100. The whole process took more than 200 machine days to finish. All of those hard instances and unsolved instances can be found on the web site [16].

9. A COMPARISON BETWEEN PCP AND BUSY BEAVER PROBLEM

Tibor Rado invented the Busy Beaver Problem in 1962, which is to find the simple deterministic Turing machine that produces a maximum number of 1's on the tape when halting [7]. These deterministic Turing machines have n states (excluding the halting state), one infinite tape initially filled with blank symbols, and they are only allowed to write 1's on the tape. $\Sigma(n)$ is defined as the maximum number of 1's such a Turing machine with n states produces when it halts. Tibor Rado also proved $\Sigma(n)$ is a non-computable function and solved $\Sigma(1)$ and $\Sigma(2)$. The continuous research on this logical game produced the exact values for $\Sigma(3)$ and $\Sigma(4)$ [18; 19], as well as the lower bounds for $\Sigma(5)$ and $\Sigma(6)$ [20; 21]. Currently, $\Sigma(6) > 1.29 \cdot 10^{865}$.

Our experimental approach to PCP shares many similarities with the approach to compute the function $\Sigma(n)$ in the Busy Beaver Problem. $\Phi(s, w)$ and $\Sigma(n)$ are both non-computable functions, and computing $\Phi(s, w)$ requires the techniques to prove PCP instances unsolvable while computing $\Sigma(n)$ needs the methods to prove Turing machines non-halting. Both approaches discard isomorphic instances or Turing machines to avoid redundant work. The pattern method raised in this paper is quite similar to the method to find the *partial recurrent pattern* in Turing machines [18].

However, since the Turing machines are deterministic, they can choose at most one transition function in each move, as contrast to multiple choices during pair matching in PCP instances. Besides, the Turing machines are amenable to move compression [20]. Therefore, it is not surprising that some Turing machine with 6 states can be found halt at step more than $3 \cdot 10^{1730}$ [21], while currently the hardest PCP instance with size and width at most 4 is merely longer than 300.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we described many new methods and techniques to tackle PCP instances, including finding optimal solutions quickly, proving instances unsolvable and creating interesting difficult instances. We successfully applied these methods to our PCP solver, and used it to scan all instances in 11 PCP subclasses and to search randomly in 2 much harder subclasses. Our work resulted in the discovery of 199 difficult instances with optimal length at least 100, and in setting the hardest instance records in 4 non-trivial PCP subclasses.

We have discovered many new characteristics and properties of PCP, and provided empirical results for solving PCP instances. These results can be helpful to investigate some theoretical issues related to this problem.

However, there is still lots of room for further improvements. There are more than 17,000 unsolved instances in $PCP[3,5]$ and $PCP[4,4]$ waiting for some new methods to conquer them. Although we implemented most of the methods and techniques discussed in this paper, the group method and pattern method have only been applied by hand to solve some hard instances. If these methods could be successfully incorporated into our PCP solver, a great portion of the unsolved instances would be proven unsolvable.

As PCP instances are closely related to their reversals, bidirectional search can also be applied to solve them. It is also very interesting to investigate the benefits brought by the complex heuristics mentioned in Section 5.3. In this way, PCP can act as a special test bed for general search enhancements.

We anticipate the work to continue to tackle $PCP[3,5]$, $PCP[4,4]$ and more difficult PCP subclasses. If the hardest instances in these subclasses could be found, it may be possible to find some similarities and link them to theoretical issues. Nevertheless, identifying more hard instances can provide a better understanding of the complexity of PCP and pave the road for improvement of solving methods.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Jonathan Schaeffer and Dr. Martin Müller for their numerous comments, and Heiko Stamer and Dr. Johannes Waldmann for their help. The author is also grateful to Dr. Richard J. Lorentz for the valuable discussions.

REFERENCES

[1] E.L. Post. A variant of a recursively unsolvable problem, Bulletin of the American Mathematical Society, 52, 264-268, 1946.

[2] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory and Computation, Addison-Wesley, 1979.

[3] M.R. Garey, and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.

[4] A. Ehrenfeucht, J. Karhumaki and G. Rozenberg. The (generalized) post correspondence problem with lists consisting of two words is decidable, Theoretical Computing Science, 119-144, 21, 2, 1982.

[5] V. Halava, T. Harju and M. Hirvensalo. Binary (generalized) post correspondence problem, TUCS Technical Report, No. 357, August 2000.

[6] Y. Matiyasevich and G. Senizergues. Decision problems for semi-Thue systems with a few rules, 11th Annual IEEE Symposium on Logic in Computer Science, 1996.

[7] T. Rado. On non-computable functions. The Bell System Technical Journal, 41:877-884, 1962.

[8] R.J. Lorentz. Creating difficult instances of the post correspondence problem, The Second International Conference on Computers and Games (CG'2000), Hamamatsu, Japan, 145-159, 2000.

[9] M. Schmidt, H. Stamer and J. Waldmann. Busy beaver PCPs, Fifth international workshop on termination (WST '01), Utrecht, The Netherlands, 2001.

[10] L. Zhao. MSc thesis: Solving and creating difficult instances of Post's correspondence problem, Department of Computing Science, University of Alberta, 2002.

[11] J. Waldmann and H. Stamer. Neuigkeiten zum PCP, presentation slides (in German), 2000.

[12] R.E. Korf. Depth-First Iterative-Deepening: an optimal admissible tree search, Artificial Intelligence, 27, 97-109, 1985.

[13] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107, 1968.

[14] J.C. Culberson and J. Schaeffer. Searching with pattern databasess. CSCSI '96 (Canadian AI Conference), Advances in Artificial Intelligence, Springer Verlag, 402-416, 1996.

[15] H. Stamer. PCP at home, `http://www.informatik.uni-leipzig.de/~pcp`, 2000-2002.

[16] L. Zhao. PCP homepage, `http://www.cs.ualberta.ca/~zhao/PCP`, 2001-2002.

[17] M. Rahn. The last instance in $PCP[3,3]$ has no solution. Unpublished manuscript. 2002.

[18] S. Lin and T. Rado. Computer studies of Turing machine problems. Journal of the ACM, 12(2):196-212, April 1965.

[19] A.H. Brady. The determination of the value of Rado's noncomputable function Sigma(k) for four-state Turing machines, Mathematics of Computation, vol. 40, no. 162, 647-665, April 1983.

[20] H. Marxen, J. Buntrock. Attacking the Busy Beaver 5, Bulletin of the EATCS, Number 40, 247-251, February 1990.

[21] H. Marxen. Busy Beaver, `http://www.drb.insel.de/~heiner/BB/`, 2002.