Tackling Post's Correspondence Problem

Ling Zhao

Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2H1
zhao@cs.ualberta.ca

Abstract. 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, and to find the solutions if they exist. In this paper, we describe new application-dependent methods used to efficiently find optimal solutions to individual instances, and to identify instances with no solution. We also provide strategies to create hard PCP instances with long optimal solutions. These methods are practical approaches to this theoretical problem, and the experimental results present new insights into PCP and show the remarkable improvement achieved by the incorporation of knowledge into general search algorithms in the domain of PCP. [1]

Keywords: single-agent search, Post's correspondence problem, PCP

1 Introduction

Post's correspondence problem (PCP for short) was invented by Emil L. Post in 1946 [10], and soon became a highly cited example of an undecidable problem in the field of computational complexity [5]. Bounded PCP is NP-complete [3]. PCP of 2 pairs was proven decidable [2], and recently a simpler proof using a similar idea was developed [4]. PCP of 7 pairs is undecidable [9]. 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. Only recently did researchers begin to build PCP solving programs [8,11,13]. Richard J. Lorentz first systematically studied the search methods used to solve PCP instances and the techniques to create difficult PCP instances. These ideas were implemented and generated many good results [8]. Our work was motivated by his paper and can be regarded as an extension and the further development of his work. As his paper is frequently cited in the following sections, we use the term *Lorentz's paper* to denote it for convenience.

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

[1] Last revised on April 25, 2003

two domains have set good examples for building a strong PCP solver, and some of this research can be directly migrated to PCP solvers after a few application-dependent modifications. On the other hand, the characteristics of PCP lead to many special search difficulties, which has prompted us to develop new techniques and discover more properties of PCP that could be nicely integrated into the solver. This work clearly demonstrates the significance of combining application-dependent knowledge with general search framework in the domain of PCP, and some of the ideas may be applied to other problems.

This paper mainly discusses three directions for tackling Post's correspondence problem. The first two directions focus on solving solvable and unsolvable instances respectively, i.e., finding optimal solutions to solvable instances efficiently and effectively, and identifying unsolvable instances. Six new methods were invented for the aim of solving instances, namely, the *mask method*, *forward pruning*, *pattern method*, *exclusion method*, *group method* and *bidirectional probing*.

The third direction of our work is concerned with creating hard instances that have very 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 shortest solution lengths are at least 100. Currently, we are holding the hardest instance records in 4 PCP subclasses.

The paper is organized as follows. We begin by introducing the definition and notation, and some simple examples in Section 2. Section 3 describes six methods that are helpful to solve instances. Section 4 explains how to create difficult instances, and Section 5 contains the experimental results and related discussions. Finally, Section 6 provides conclusions and suggestions for future work.

2 What is Post's correspondence problem

An instance of Post's correspondence problem is defined as a finite set of pairs of strings (g_i, h_i) $(1 \le i \le s)$ over an alphabet Σ. A solution to this instance is a sequence of selections $i_1 i_2 \cdots i_n$ $(n \ge 1)$ 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,[2] s in the above, is called its *size*, and its *width* is the length of the longest string in g_i and h_i $(1 \le i \le s)$. *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.

Since solutions can be stringed together to create longer solutions, in this paper we are only interested in *optimal solutions*, which have the shortest length over all solutions to an instance. 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.

[2] In the following, we use the name *instance* to specifically represent *PCP instance*.

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. In this case, it will not influence the result if one of the duplicated pairs is removed. For brevity, 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. Then the following relations hold:

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

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 $PCP[3,3]$.

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

2.1 Example of solving a PCP instance

Now let's see how 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:

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

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 the above configuration, and the situation changes to:

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

Now, there are two matching choices: *pair 1* and *pair 2*. By using *the mask method* (described in Section 3.1.1), we can avoid trying *pair 2*. Then, *pair 1* is 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, and the configuration becomes empty, which shows that the sequence of selections, *1311322*, forms a solution to PCP (1). When all combinations of up to 7 selections of pairs are exhaustively searching, this solution can thus be proven to be optimal.

2.2 More examples

Some instances may have no solution. For example, the following PCP (2) is unsolvable, which can be proven through the *exclusion method* discussed in Section 3.2.1.

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

An analysis of the experimental data we gathered shows that in PCP subclasses of smaller sizes and widths, only a small portion of instances have solutions, and a much smaller portion have long optimal solutions. PCP (3) is such a difficult instance with optimal length of 206. It is elegant that this simple form embodies such a long optimal solution. If a computer performs a brute-force search by considering all possible combinations up to depth 206, the computation will be enormous. That's the reason why we utilize AI techniques and new methods related to special properties of PCP to prune hopeless nodes, and thus, accelerate search speed and improve search efficiency.

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

The optimal solution to an instance may not be unique. For instance, PCP (4) below has 2 different optimal solutions of length 75.

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

Now let's take a look at PCP (5). 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 end. This example shows an unfortunate characteristic of some instances: *the search space is unbounded*. This special property suggests that we cannot rely on search to prove some instances unsolvable. Several new methods such as the *exclusion method*, which helps to prove the unsolvability of PCP (5), have been invented and will be presented in the next section.

$$\begin{pmatrix} \mathtt{100} & \mathtt{0} & \mathtt{1} \\ \mathtt{0} & \mathtt{100} & \mathtt{111} \end{pmatrix} \tag{5}$$

3 Solving PCP instances

Intuitively, solvable and unsolvable instances should be treated separately, but for PCP instances, many methods are valuable to both types of tasks. Therefore, we do not divide these methods into two sections, but discuss them together in this section instead.

Lorentz's paper introduced some general search techniques that can be used in the PCP solver:

1. Depth-first iterative-deepening search [7] works well on problems with exponential search space and provides a satisfactory trade-off between time and space.
2. A cache table can be used to prune revisited nodes (similar to what transposition tables do in game playing programs).
3. System-level programming techniques such as tail recursion removal result in satisfactory improvement of the running time.

Lorentz's paper also introduced the concept of *filters*, which consist of simple rules to identify unsolvable instances. In addition, configurations of an instance are not of fixed size, thus it will be very inefficient to use the standard memory allocation functions provided by an operating system. Therefore, we implemented specialized routines for configuration allocation, and this work resulted in about 15% improvement in speed. Please refer to [8,13] for details of these ideas.

In the following part, we will describe six new methods to tackle PCP instances, which can be roughly categorized into three classes:

1. Pruning configurations: the mask method, forward pruning, and pattern method.
2. Simplifying instances: the exclusion method and group method.
3. Choosing the easier direction: bidirectional probing.

These methods are all closely related to the properties of PCP, and four of them including the mask method, pattern method, exclusion method and group method are very specific and cannot work for any given instance. However, they do improve the performance of the solver dramatically and enable it to answer some unresolved questions in Lorentz's paper.

3.1 Pruning configurations

Configurations are a vital feature for solving PCP instances. The difficulty encountered in solving an instance is often presented by its vast number of configurations that need to be examined. But in many cases, a great portion of configurations can be pruned without examining their descendants, and thus, a significant percentage of search effort can be saved. Especially, the unbounded search space may be reduced to a finite one, making it possible to prove the unsolvability of some instances.

In order to make it easy to explain, we first introduce some definitions concerning configurations. The *reversal* of a configuration is generated by reversing its string. The *turnover* of a configuration is generated by flipping its position from top to bottom or vice versa. A configuration is *generable* to an instance if it can be generated from the empty configuration by a sequence of selections of pairs in the instance. Similarly, A configuration is *solvable* to an instance if it can lead to the empty configuration by a sequence of selections.

3.1.1 Mask method

The mask method deals with pruning all configurations in the top or in the bottom. Before going into detail of the powerful and seemingly magical method, we first introduce the concepts of critical configuration and valid configuration.

Definition 1: A *critical configuration* in an instance is a non-empty configuration that can be *fully matched* by a pair: the resulting configuration is empty or can be *turned upside-down* by a pair: the resulting configuration changes its position from that of the previous one.

Definition 2: A configuration is *valid* to an instance if it is generable and solvable.

Critical configurations are critical because they constitute an indispensable step for a configuration in general to reach a solution. For any configuration in the top, a necessary condition for it to lead to a solution is that there must exist a critical configuration in the top in the solution path. This property can be justified by the fact that a solvable configuration must change its position somewhere (we define the position of the empty configuration to be neither top nor bottom). A configuration being valid also means it occurs in a solution path. As a result, if no valid critical configuration in the top exists in an instance, then all configurations in the top can be 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 possible critical configurations, and then test their validity. The first step can be simply done by enumeration. The second step, validity testing, can also be automated, because there is a nice relation between an instance and its *reversal*, which is generated by reversing all the strings in the original instance: the question as to whether one configuration is generable to an instance is equivalent to the question as to whether the turnover of the reversal of the configuration is solvable to the reversal of this instance. This property can be easily proven by using the definition of PCP. It needs to be clarified that though the process of checking configurations solvable can be automated, it cannot always terminate without other stopping conditions imposed. In fact, this process is undecidable in general.

The following will explain 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 critical configurations in the top. Since they could either be fully matched or turned over by at least one pair, any matched pair must have a longer bottom string. In this instance, the matched pair could only be *pair 2* or *pair 3*. It is not hard to find that only one critical configuration in the top exists, i.e. 10 in the top, which can be fully matched by *pair 3*. Secondly, we check whether the 10 in the top can be generated by PCP (6), or equivalently, whether 01 in the bottom in PCP (7) can lead to a solution. However, in PCP (7), the configuration of 01 in the bottom cannot use any pair to match. Thus, PCP (6) 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 this fact was not known, *pair 1* would be chosen as the starting pair and a huge useless search space would have to be explored before concluding that its optimal length is 160. That is the reason why Lorentz's solver successfully found solutions to two instances but could not prove their optimality [8]. Both can be decided with the assistance of the mask method.

The mask method can also prove the unsolvability of many instances. For example, if an instance has some masks that forbid all possible starting pairs, it has no solution.

GCD rule
The step to prove critical configurations invalid can be strengthened by the *GCD* (Greatest Common Divisor) *rule*: let d be the greatest common divisor of the length differences of all pairs, then the length of every valid configuration must be a multiple of d.[3] 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 critical configuration of 0 in the bottom, which can be turned upside-down by *pair 2*, it is invalid because its length is not a multiple of 2. As a result, this instance has a bottom mask, revealing that the starting selection must be *pair 2*. Finally, with a few steps of enumeration, we can prove that PCP (8) has no solution.

The above example uses the difference of length, and similarly, we can use the difference of the number of elements 0 or 1. For example, if the difference of the number of occurrences of element 0 in the two strings of any pair has a GCD d, then the number of element 0's occurred in any valid configuration must be a multiple of d.

3.1.2 Forward pruning

Similar to heuristic search algorithms such as A^* and IDA^* [6,7], heuristic functions can be used to calculate lower bounds of the solution length for a configuration (for an unsolvable instance, its solution length is defined to be infinity). A heuristic value of a configuration is an estimate of how many more

[3] This idea was separately mentioned by R. Lorentz and J. Waldmann in private communications.

selections are needed before reaching 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 is admissible, the pruning is safe and does not affect the optimality of solutions.

One simple heuristic value of a configuration in the top (bottom) can be calculated by its length divided by the maximum length difference of all pairs that have their bottom (top) strings longer. This heuristic is based on the balance of length, and similarly, we can calculate heuristics on the balances 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 [1]. 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.

3.1.3 Pattern method

If a configuration has a prefix α and any possible path starting from it will always generate a configuration with the prefix α after some steps, and no empty configuration occurs in the middle, then this prefix cannot be completely removed whatever selections are made. The significance of this property is that any configuration having such a prefix never leads to a solution. This observation essentially comes from the goal to shrink configurations to the empty string. If there is a substring that will unavoidably occur, it is impossible for configurations to transfer to the empty string.

The pattern method is a very powerful tool to prove instances unsolvable. The following example illustrates how this method is effective to prove that PCP (9) has a prefix pattern of 11 in the top, and as a result, PCP (9) is proven unsolvable.

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

For a configuration of $11A$ in the top, where A can be any string, the next selection can only be *pair 1*. Thus a new configuration $1A011$ in the top 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 configuration $11A$ in the top will definitely transfer to another configuration $11B$ in the top after a number of steps. The prefix cannot be removed, showing that any configuration in the top with a prefix of 11 will never lead to a solution. The procedure to find a prefix in PCP (9) is presented in Fig. 1.

The dotted vertical line in the figure partitions configurations into two parts: left part and right part. If a configuration still has the chance to lead to a

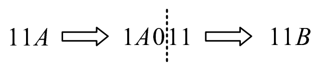

Fig. 1. Deduction of a prefix pattern in PCP (9)

solution, 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 is matched across the border.

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, prune any configuration in the top with a prefix of 11. When this process terminates, the unsolvability of PCP (9) is proven.

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.

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

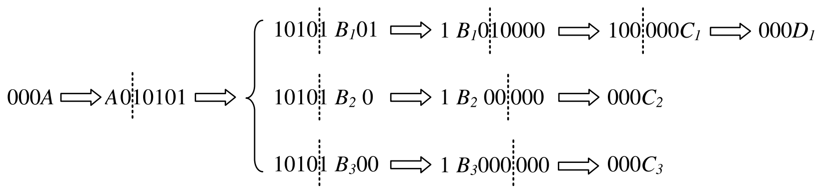

Fig. 2. Deduction of a prefix pattern in PCP (10)

3.2 Simplifying instances

Through the analysis of specific instances, it is possible to simplify them by removing some pairs or replacing substrings with simpler ones. This simplification significantly reduces the search work, and makes it possible to solve some instances.

3.2.1 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 all other pairs, then those pairs are useless and can be safely removed. PCP (11) is such an example.

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

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. Any configuration generated by *pair 1* and *pair 2* is 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 configuration in the bottom with length less than 3 that can be matched by *pair 3* is 10 in the bottom, yet it cannot be generated by selections of *pair 1* and *pair 2*.

Therefore, after selections start at *pair 2*, we only need to deal with an instance consisting of *pair 1* and *pair 2*. This new instance never leads to a solution because of the length of configurations it generates will never decrease. Hence starting at *pair 2* is hopeless. Combined with the fact that this instance has a top mask, we successfully prove it unsolvable.

3.2.2 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 will be matched in the same selection. The group method is utilized to detect such groups and help to simplify instances. Consider the following instance, where the substring 10 is undivided.

$$\begin{pmatrix} \texttt{011} & \texttt{10} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{010} \end{pmatrix} \tag{12}$$

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:

$$\begin{matrix} \begin{pmatrix} \texttt{011} & g & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{0}g \end{pmatrix} \\ g = \texttt{01} \end{matrix} \tag{13}$$

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 will 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 and safely removed when selections start at *pair 3*. Since no other possible starting pair exists, PCP (12) is unsolvable.

3.3 Bidirectional probing

An instance and its reversal are isomorphic to each other in the sense that they share the same solvability, and have the same number of optimal solutions and the same optimal length if solvable. Therefore, solving either one of them is enough. But these two forms may be amazingly different in terms of search difficulties, as shown experimentally in Section 5.1.

Hence, we use a probing scheme to decide which search direction is more promising. Initially we set a *comparison depth* (40 in the implementation). Then two search processes are performed for the original instance and its reversal to the comparison depth separately. 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 factor in most instances is quite stable, this scheme worked very well in our experiments.

4 Creating difficult instances

A strong PCP solver enhanced by the application-dependent methods discussed in the previous section is essential for creating many difficult instances. Besides, the hard instances that we found attracted us to find their solutions efficiently, and those unresolved instances were intriguing for 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.

4.1 Random search for hard instances

A random instance generator plus a PCP solver is a straightforward means of discovering interesting instances, which has been used to create difficult instances in [8]. Statistically, the generator will create a few hard instances with very long optimal lengths sooner or later. However, we can still do much work to increase the chance of finding hard instances. By using several search enhancements and various methods that help to prove instances unsolvable, the program can quickly stop searching hopeless instances and find the optimal solutions to solvable instances faster.

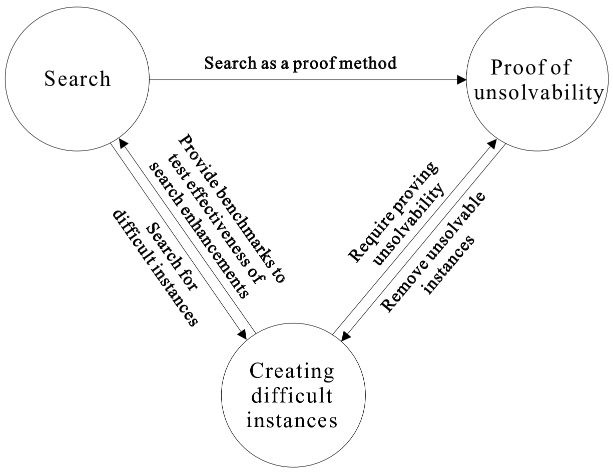
Search
Proof of unsolvability
Creating difficult instances
Search as a proof method
Provide benchmarks to test effectiveness of search enhancements
Search for difficult instances
Require proving unsolvability
Remove unsolvable instances

Fig. 3. Relations between three research directions in PCP

During the search process of an instance, we use three factors as stopping conditions if no solution is found. 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. Based on the observation of the hard instances we collected, most of those instances do not have a large number of cutoff nodes (the largest is only 28,972). Thus in the implementation, we also use the number of cutoff nodes as one of the stopping conditions.

One method suggested in the literature is the removal of instances that have the *pair purity* feature, that is, a pair consisting wholly of ones or zeroes [8]. This idea is based on the observation that many instances bearing this feature have no solution, but have quite messy search trees that are very costly to explore. However, our experimental results show that this method may cause the failure of discovering some hard instances (described in Section 5.3).

4.2 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 will be discovered. Even if we cannot solve all of them, the unresolved instances may be left as incentives for new approaches.

It is not hard to generate all instances in a finite PCP subclass, and there is a related problem about how to remove those isomorphic instances which potentially will cause lots of redundant work. We omit it for brevity, and for further information see [13].

In Lorentz's paper, the subclass $PCP[3,3]$ was explored, with more than 2000 instances unresolved (most of them have the pair purity feature). We continued the work and rescanned all instances in $PCP[3,3]$ as well as 9 other subclasses. The results are shown in Sections 5.2 and 5.3.

4.3 New PCP records

The random and systematic search schemes for creating difficult instances helped create new records in 4 PCP subclasses. These records are new instances with the longest optimal lengths known in specific PCP subclasses. The records in subclasses $PCP[3,4]$ and $PCP[4,3]$ were found by the systematic search scheme and those in $PCP[3,5]$ and $PCP[4,4]$ were obtained through the random search scheme. Table 1 gives these four records as well as the record in $PCP[3,3]$, which was discovered by R.J. Lorentz and J. Waldmann independently. More information is provided on the websites [12,14].

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

Table 1. Records of hardest instances known in 5 PCP subclasses

5 Experimental results and analysis

We implemented most of the methods we discussed in Section 3 under the depth-first iterative-deepening search framework, except the pattern method and group method, because we could not find a general way to automate them. The program was written using C++ under Linux. 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 4; the remaining test case is the hardest instance known in $PCP[3,3]$, as shown in Table 1.

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 an optimal solution with length even greater than 300.

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 600MHZ processor and a 128M RAM.

5.1 Results of solving methods

It is hard to give a quantitative evaluation of the improvements derived from the mask method and the exclusion method, since sometimes they are essential to solve instances. It is more proper to comment that these two methods would help to prune a huge search space in some cases.

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 132, the difference will explode exponentially. Thus, it becomes unrealistic to solve this instance when the wrong direction is chosen. This clearly demonstrates how important the bidirectional probing method 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 balance of length, and the second is based on the balance of elements, as Section 2.1.2 described. Three separate experiments were conducted on forward pruning, namely, only using the first type of heuristic, only using the second type and using both types. The results show that only using the heuristic on the balance of length gives the best performance.

We tried to compare the improvement achieved by heuristic pruning with the situation when no pruning is used, 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!

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

We also did experiments on some search parameters. For example, 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.

5.2 Results of scanning PCP subclasses

We scanned seven PCP subclasses that are easy to handle, and all instances in these subclasses have been completely solved. All isomorphic instances have been removed, and the results are shown in Table 2. 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 PCP subclasses is. In addition, the small largest optimal lengths in these subclasses partly explain why they can be completely explored in a small amount of computation.

PCP subclass	total number	after filter	after mask	after exclusion	solvable instances	unsolvable instances	largest optimal length
$PCP[2,2]$	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 2. Solving results of 7 PCP subclasses

One special phenomenon we observed is that the hardest instances in PCP subclasses with size 2 can all be represented in the following form:[4]

$$\begin{pmatrix} 1^n0 & 1 \\ 1 & 01^n \end{pmatrix} \tag{16}$$

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

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

We used the systematic method to further examine three PCP subclasses that are much harder to conquer. Table 3 first summarizes the results from $PCP[3,3]$.

Total number	127,303
After filter	8,428
After mask	2,089
After exclusion	2,002
Solvable instances	1,961
Unsolvable instances	40
Unsolved instances	1

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

In Tables 3 and 4, an instance removed by the exclusion method may still have solutions, but it cannot have a *non-trivial* solution.[5] Since the result of solving such an instance is identical to the combinations of the results from instances with smaller sizes, we do not give it any further processing. Similarly, instances removed by the element balance filter (see [8]) may also have trivial solutions, but these solutions are of no interest to us. In addition, 32 instances remained unsolved by our PCP solver, but were proven unsolvable by hand using the methods discussed in Section 2. The difference comes from the fact that though some methods can successfully prove several instances unsolvable, some of them are too specific to be generalized, and thus they were not incorporated into our PCP solver.

The only unsolved instance in $PCP[3,3]$ is PCP (17).[6] Our solver searched to a depth of 300, but still could not find a solution to this instance. Various deduction methods were tried to prove it unsolvable, but also failed. Compared to the fact that the hardest instance in all instances of $PCP[3,3]$ except this one only has an optimal length of 75, we believe that this instance is very likely to have no solution. However, apparently new approaches are needed to deal with it.

$$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{01} & \texttt{110} \end{pmatrix} \tag{17}$$

5.3 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 4.

[5] A solution to an instance is *trivial* if in the solution not all pairs of the instance are used.

[6] It has solved by Mirko Rahn through a new method that generalizes the pattern method discussed in this paper.

	$PCP[3,4]$	$PCP[4,3]$
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

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

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.

In the 72 hard instances we collected from $PCP[4,3]$, 13 instances (18.1%) have the pair purity feature, and some of them need more than a hundred seconds to solve. This evidence suggests that even an instance with the pair purity feature may still have a very long optimal solution (the longest we found is 240), and the quantity of these instances cannot be ignored.

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 website [14].

6 Conclusions and future work

In this paper, we described some new methods and techniques to tackle PCP instances, including finding optimal solutions quickly, proving instances unsolvable and creating many interesting difficult instances. We successfully solved some instances that were unsolved in Lorentz's paper and scanned all instances in 10 PCP subclasses. Our work resulted in the discovery of 199 difficult instances with optimal length at least 100, and in setting new hardest instance records in 4 PCP subclasses. We also present the empirical results for solving PCP instances and they may serve for investigating some theoretical issues related to this problem.

Although the six new methods described in this paper are all application-dependent, the ideas behind them are not new at all. For example, bidirectional probing uses the idea of performing a shallow search to direct a deep search, and incorporating application-dependent knowledge into general search framework has worked successfully in many domains such as planning and games.

This paper further exploits the ideas discussed in Lorentz's paper, and there is no doubt that the results can be improved. PCP (17) is the only unsolved instance in $PCP[3, 3]$, and more than 17,000 instances in $PCP[3, 4]$ and $PCP[4, 3]$ are unsolved, waiting for 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. We believe that if these methods could be successfully incorporated into our PCP solver, a great portion of 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 evaluate the benefits provided by the complex heuristics mentioned in Section 2.1.2. 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 PCP subclasses. If the hardest instances in these subclasses could be found, it may be possible to find some similarities between them and link them to theoretical issues. Nevertheless, identifying more hard instances can provide more a better understanding of the complexity of PCP and pave the road for improvement of solving methods.

Acknowledgement

I would like to thank Dr. Michael Buro and Dr. Lorna Stewart for their instructive advice, and thank Dr. Richard J. Lorentz, Heiko Stamer and Dr. Johannes Waldmann for their generous help. Special thanks are due to my supervisors, Dr. Jonathan Schaeffer and Dr. Martin Müller for their encouragement, patience and meaningful discussions. I am also grateful to the anonymous referees for the valuable comments. This work has been supported by the Alberta Informatics Circle of Research Excellence (iCORE).

References

1. J.C. Culberson and J. Schaeffer. Searching with pattern databasess. CSCSI '96 (Canadian AI Conference), Advances in Artificial Intelligence, Springer Verlag, 402-416, 1996.
2. 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.
3. M.R. Garey, and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.
4. V. Halava, T. Harju and M. Hirvensalo. Binary (generalized) post correspondence problem, TUCS Technical Report, No. 357, August 2000.
5. J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory and Computation, Addison-Wesley, 1979.

6. 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.
7. R.E. Korf. Depth-First Iterative-Deepening: an optimal admissible tree search, Artificial Intelligence, 27, 97-109, 1985.
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. 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.
10. E.L. Post. A variant of a recursively unsolvable problem, Bulletin of the American Mathematical Society, 52, 264-268, 1946.
11. M. Schmidt, H. Stamer and J. Waldmann. Busy beaver PCPs, Fifth international workshop on termination (WST '01), Utrecht, The Netherlands, 2001.
12. H. Stamer. PCP at home, `http://www.informatik.uni-leipzig.de/~pcp`, 2000-2002.
13. L. Zhao. MSc thesis: Solving and creating difficult instances of Post's correspondence problem, Department of Computing Science, University of Alberta, 2002.
14. L. Zhao. PCP homepage, `http://www.cs.ualberta.ca/~zhao/PCP`, 2001-2002.