University of Alberta

Library Release Form

Name of Author: Ling Zhao

Title of Thesis: Solving and Creating Difficult Instances of Post's Correspondence Problem

Degree: Master of Science

Year this Degree Granted: 2002

Ling Zhao

Department of Computer Science
Edmonton, Alberta
Canada, T6G 2E8

Date: ________________

University of Alberta

Solving and Creating Difficult Instances of Post's Correspondence Problem

by

Ling Zhao

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Solving and Creating Difficult Instances of Post's Correspondence Problem** submitted by Ling Zhao in partial fulfillment of the requirements for the degree of **Master of Science**.

Jonathan Schaeffer
Co-Supervisor

Martin Müller
Co-Supervisor

Francis Jeffry Pelletier

Bernard Linsky
External Examiner

Date: ____________________

To my parents

Abstract

Post's correspondence problem (PCP) was invented by Emil L. Post in 1946. It is an undecidable problem, which means that it is impossible to find an algorithm to solve the whole problem. This problem has been extensively discussed in the theoretical computer science literature, but only recently did some researchers begin to look into the empirical properties of this problem.

Although this problem cannot be completely solved, some of its instances can be solved and may have very long solutions. It is instructive and important to find instances that have very long shortest solutions to reveal new properties and understand the complexity of this problem.

In this thesis, several problem-specific search enhancements have been employed to find the shortest solutions of instances efficiently and effectively. New disproof methods were invented to identify instances that have no solution. All of these methods made it possible to completely solve 7 PCP subclasses and to fully scan 3 other PCP subclasses. Our effort culminated in the discovery of 199 hard instances with very long shortest solutions and in setting new records for the hardest instances known in 4 PCP subclasses. In addition, we present experimental results on several important properties of PCP and on the search and disproof methods used. These results will lead to a better understanding of the theoretical and empirical properties of this problem.

Acknowledgements

First of all, great thanks go to my co-supervisors: Dr. Jonathan Schaeffer and Dr. Martin Müller. Without their guidance, discussion, criticism and encouragement, it is impossible to achieve so many exciting results. I am also grateful to the other members in my thesis committee, Dr. Francis Jeffry Pelletier and Dr. Bernard Linsky, for their suggestions and corrections.

Many researchers helped me during my research on Post's correspondence problem. Dr. Richard J. Lorentz's kind assistance in the early stages of my work is greatly acknowledged. Thank you to Dr. Michael Buro and Dr. Lorna Stewart for their instructive advice. Thank you to Heiko Stamer and Dr. Johannes Waldmann, especially for those PCP records they put up which inspired me to break.

Financial support from the Department of Computing Science at the University of Alberta and the Alberta Informatics Circle of Research Excellence (iCORE) is gratefully acknowledged.

I would also like to thank all members in GAMES group at University of Alberta for their meaningful discussions.

I also want to thank Luqing Wang and Xiaoqin Yuan for their help with LaTeX. Thanks are also due to my friends, Peng Zhao and Xiang Wan for their long-time discussions with me and help in many aspects of my life.

Finally, I would like to thank my parents for their patience and continuous support and my girlfriend for her superb understanding and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Post's correspondence problem (PCP for short) is a classic undecidable problem and is widely used as an example to explain the concept of undecidability in the complexity hierarchy. Although the whole problem cannot be solved by computers in general, researchers have discovered many results on variances and subclasses of Post's correspondence problem. There are a few pending problems associated with it, and most of them fall into the type of complexity issues. Only recently did some researchers begin to utilize search and deduction methods to solve individual PCP instances, and propose ideas and conjectures on theoretical problems through experimental analysis.

Post's correspondence problem is essentially a single-agent search problem, so traditional search techniques invented in this area can be useful to solve individual PCP instances. As the result of PCP's simple rules and nice properties which are aesthetically appealing, we use PCP as a test bed for our research in Artificial Intelligence (AI). Some new search methods have been invented to deal with PCP instances specifically, and these ideas may also benefit other applications. Though a PCP solver cannot solve all instances in PCP, it could solve most instances it encountered in our experiments. By scrutinizing all instances in several PCP subclasses, we gathered many useful experimental results that guided us to find similarities and regularities inside PCP. This may help researchers develop insights on why this problem is theoretically unsolvable.

1.1 An informal introduction to Post's correspondence problem

A PCP instance is composed of several pairs with each pair having a top and a bottom string. The goal is to select a sequence of pairs (may not be distinct) such that the concatenated top string and bottom string are identical. The following PCP (1.1) is an example of a PCP instance.

$$\begin{array}{ccc} \textit{pair 1} & \textit{pair 2} & \textit{pair 3} \end{array}$$
$$\begin{pmatrix} \texttt{100} & \texttt{0} & \texttt{1} \\ \texttt{1} & \texttt{100} & \texttt{00} \end{pmatrix} \tag{1.1}$$

Let's consider the selections of pairs according to the sequence *1311322*.

1	*3*	*1*	*1*	*3*	*2*	*2*
100	1	100	100	1	0	0
1	00	1	1	00	100	100

The above may not be clear enough, but if all blank spaces between strings are removed, we will get the result below:

1 3 1 1 3 2 2
1001100100100
1001100100100

Now the top string and the bottom string are exactly the same, showing that the sequence *1311322* is a solution to PCP (1.1). Although verifying a solution can be easily done in polynomial time, finding a right sequence of pairs may be very hard.

The number of pairs in a PCP instance is called its *size*. We can further divide the whole Post's correspondence problem into an infinite number of subclasses according to their sizes. For example, $PCP[3]$ is the PCP subclass of all instances with size 3.

1.2 A brief history of Post's correspondence problem

Emil L. Post invented Post's correspondence problem[1] in 1946. He also proved it was undecidable [1], which means there exists no algorithm capable of solving all its instances (the number of instances is infinite). Many textbooks on computational complexity use PCP as an example in the introduction of undecidability (e.g. [2]). Post's correspondence problem is one of the standard problems to prove the undecidability of other problems, similar to the place of SAT for proving NP-completeness. *Bounded PCP* (the bound is on the solution length) is NP-complete [3]. The PCP subclass of size 2 has been proven decidable [4], while a simpler proof using the same idea was developed recently [5]. The PCP subclass of size 7 is undecidable [6], and therefore, any PCP subclass of size greater than 7 is also undecidable. However, decidability of PCP subclasses of size between 3 and 6 is still unknown.

During more than a half century since its appearance, Post's correspondence problem has been studied by many researchers, but mainly on its theoretical properties. Few tried to investigate how to solve individual instances of this problem. This sounds quite reasonable, because PCP is unsolvable in general and nobody could build a program capable of conquering all instances. As a result, the examples in textbooks normally are either solvable instances whose solutions can be easily found, or unsolvable instances which can be recognized through very simple rules. This causes some confusion between the extreme hardness of the problem and those easy examples.

Probably inspired by the *busy beaver problem* [7], Richard J. Lorentz first considered the methods for creating difficult PCP instances which have small sizes and widths, yet quite long shortest solutions [8]. We gratefully acknowledge that our work was motivated by Lorentz's paper, and can be regarded as an extension and the further development in this direction. M. Schmidt *et al.* developed some ideas for solving PCP instances and discussed the issues of the

[1] Actually Post called it *correspondence decision problem*. In the literature, the name *Post's correspondence problem* and *Post correspondence problem* both are commonly used. In this thesis, we use the former one, or its acronym *PCP*, to denote this problem.

possible connection between solving concrete instances and open theoretical issues [9]. For example, experimental data seem to indicate a quadratic bound on the length of the shortest solutions of solvable instances in $PCP[3]$. If such a bound could be proven, the decidability of $PCP[3]$ would be settled.

1.3 Motivation

The search techniques in Artificial Intelligence (AI) have progressed significantly in the past 20 years. This is exemplified in single-agent search by the work on the sliding-tile puzzles and Rubik's cube, and in the domain of two-player board games, such as Checkers and Chess, where computer programs have defeated the human world champions. A variety of search enhancements developed in these domains have set good examples for us to build a strong PCP solver. Some of this research can even be directly migrated to solve PCP instances simply after a few application-dependent modifications. On the other hand, the distinct characteristics of PCP such as the unbounded search space lead to special search difficulties, which has prompted us to develop new search methods and find more application-related properties that could be nicely integrated into a PCP solver.

As discussed later, it is very difficult to prove that some PCP instances have no solution, and therefore, a few instances remain unsolved and are likely unsolvable. Our current search methods are not suitable to prove the unsolvability of these instances, and we have to seek assistance from mathematical deduction as well as disproof methods derived from PCP's properties and characteristics.

Since PCP is a purely theoretical problem, the obvious question to ask is: why bother solving PCP instances? Here are some answers.

1. Some PCP instances pose special difficulties for current search methods, so PCP is an interesting test bed for research in Artificial Intelligence. We have not been able to directly link it to real-world applications, but this may be because PCP is unsolvable, and hence no one can use it in

reality. However, the methods and ideas proposed to solve small PCP instances may transfer to future applications.

2. By using experimental approaches to tackle PCP, we have collected a large amount of empirical data that present a statistical point of view of this problem. This quantitative approach helped us to find new phenomena and properties, and led to a clearer insight on how hard this problem is. We expect that this work could help researchers to solve theoretical problems related to PCP and unveil the border between its hard and easy instances.

1.4 Structure of the thesis

The thesis mainly addresses how to tackle Post's correspondence problem, including solving PCP instances and looking for difficult PCP instances that have very long optimal solutions. The thesis is organized as follows. In Chapter 2, we show the definition and a variety of properties of Post's correspondence problem, for example, the symmetry of PCP instances and relations among different PCP subclasses. Then in the next three chapters, we present our work on the three directions concerning this problem.

Chapter 3 is on *searching for solutions to PCP instances efficiently and quickly.* A few known search algorithms implemented in our PCP solver are discussed and new methods including the mask method, GCD enhancement and forward pruning are presented.

Chapter 4 deals with *proving instances unsolvable.* This is the first time that this research direction has been seriously investigated. The mask method, exclusion method, symmetry method, group method and pattern method were invented to serve this aim.

Chapter 5 is concerned with *finding difficult PCP instances.* With the help of the methods developed in Chapter 3 and Chapter 4, a strong PCP solver significantly increases the chance of finding difficult instances, which resulted in the discovery of 199 hard instances whose shortest solution lengths are greater than 100. There are two different efforts involved here, i.e, examining randomly generated PCP instances and systematically scrutinizing all instances in several PCP subclasses. Currently we are holding the hardest instance records in four PCP subclasses (see Section 5.4).

In Chapter 6, the experimental data and their analysis are shown. Finally, conclusions and future work are given in Chapter 7.

Chapter 2

The magic behind Post's correspondence problem

2.1 Definitions and notation

Definition 2.1.1 *Given an alphabet Σ, an instance*[1] *of Post's correspondence problem is a finite set of pairs of strings (g_i, h_i) $(1 \leq i \leq s)$ over 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 an instance, s in the above, is called its *size*, and its *width* is the length of the longest string in g_i and h_i $(1 \leq i \leq s)$. *Pair i* stands for 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 $\{0, 1\}$ since we can always transform 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 was proven either solvable or unsolvable, it has been *solved*; on the other hand, if no such proof is found, it is *unsolved* yet. We will give examples in the next two sections.

Lemma 2.1.1 *If sequence I and sequence J both are solutions to instance P, then sequence IJ is also a solution to P.*

Corollary 2.1.1 *Any solvable instance has an infinite number of solutions.*

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

Lemma 2.1.1 is evident, disclosing that the sequence formed by concatenation of solutions is a solution too. This naturally leads to Corollary 2.1.1. Since it is unnecessary to find all solutions to an instance, in this thesis, 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.* Note that there may be more than one optimal solution to an instance. 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.

To conveniently represent subclasses of Post's Correspondence Problem, we use $PCP[s]$ to denote the set of all instances with size s, and $PCP[s,w]$ for the set of all instances with size s and width w. So we can find the following relations:

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

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 $(i,2)$. The following PCP (2.1) is an example in $PCP[3,3]$, which has been shown in the previous chapter.

$$\begin{pmatrix} \texttt{100} & \texttt{0} & \texttt{1} \\ \texttt{1} & \texttt{100} & \texttt{00} \end{pmatrix} \tag{2.1}$$

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 solving result if one of the duplicated pairs is removed. For brevity, we assume the instances discussed in this thesis are all nontrivial and non-redundant.

2.2 An example of solving PCP instances

Now let's see a concrete example of solving PCP (2.1) in the above. 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:

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

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*. We use c to denote a configuration, which contains not only a string, but also its position information: *top* or *bottom*. If the position of c is flipped, i.e., from top to bottom or vice versa, we will get $\bar{c}$, its *turnover*.

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

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

Now, there are two matching choices: *pair 1* and *pair 2*. By using *the mask method* (described in Section 3.3), 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:

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

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

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

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

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 (2.1). By exhaustively searching all combinations of up to 7 selections of pairs, we can prove that this solution is optimal. The search tree for solving this instance is illustrated in Figure 2.1.

In the figure, the thick directed lines in the search tree constitute a solution path. Nodes in the tree represent configurations, which are in the bottom when there are bars over them and in the top otherwise. A connection between two

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

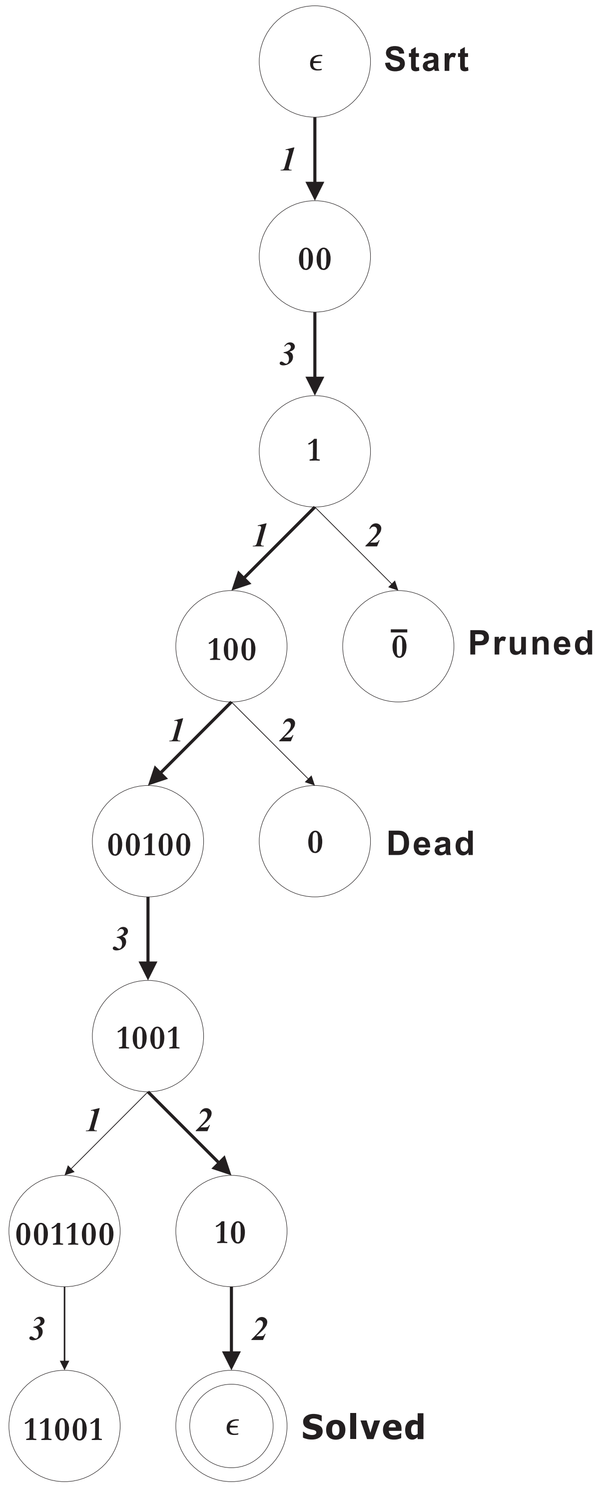

Figure 2.1: Search tree and solution path in PCP (2.1)

nodes is labelled by the pair selected.

2.3 More examples

Some PCP instances may have no solution. For example, the following PCP (2.2) is unsolvable, which can be proven through the *exclusion method* (discussed in Section 4.3).

$$\begin{pmatrix} \texttt{110} & \texttt{0} & \texttt{1} \\ \texttt{1} & \texttt{111} & \texttt{01} \end{pmatrix} \tag{2.2}$$

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 has very long optimal solutions. PCP (2.3) is such a difficult instance whose optimal length is 206. It is elegant that this simple form embodies such an incredibly long optimal solution. If a computer performs a brute-force search, i.e., 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. These techniques will be introduced in Chapter 3.

$$\begin{pmatrix} \texttt{1000} & \texttt{01} & \texttt{1} & \texttt{00} \\ \texttt{0} & \texttt{0} & \texttt{101} & \texttt{001} \end{pmatrix} \tag{2.3}$$

The optimal solution to an instance may not be unique. In PCP (2.4) below, there are 2 different optimal solutions of length 75.

$$\begin{pmatrix} \texttt{100} & \texttt{0} & \texttt{1} \\ \texttt{1} & \texttt{100} & \texttt{0} \end{pmatrix} \tag{2.4}$$

Now let's take a look at PCP (2.5). 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 PCP instances: *the search space is unbounded.* Take the game of Go for example. Although its complexity goes well beyond the computer's capacities, its number of states remains finite, which implies that a fast enough supercomputer with a large enough memory could in principle solve this game. However, even if such a supercomputer could be constructed, it still could not solve PCP (2.5) simply by going through all possible states, since there are infinitely many. This lesson suggests the need for clever ideas to prove instances unsolvable. Several new methods such as the *exclusion method*, which helps to prove the unsolvability of PCP (2.5), have been invented and will be presented in Chapter 4.

$$\begin{pmatrix} \texttt{100} & \texttt{0} & \texttt{1} \\ \texttt{0} & \texttt{100} & \texttt{111} \end{pmatrix} \tag{2.5}$$

2.4 Properties of Post's correspondence problem

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

2.4.1 Reversal properties

We first introduce an important concept: the *reversal* of an instance.

Definition 2.4.1 *Let S be a string, then its reversal, denoted by S^R, is S written backwards.*

Definition 2.4.2 *Let P: (g_i, h_i) $(1 \leq i \leq s)$ be an instance, then its reversal, denoted by P^R, is (g_i^R, h_i^R) $(1 \leq i \leq s)$.*

Suppose we have a solution $i_1 i_2 \cdots i_n$ to instance P, then 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 2.4.1 *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 even go further: suppose that through a sequence of selections, $i_1 i_2 \cdots i_j$, the corresponding configurations generated during these steps are $c_1, c_2, \cdots, c_j$, then configuration $\bar{c}_j^R$ in instance P^R can be solved by selections $i_j i_{j-1} \cdots i_1$.

Lemma 2.4.2 *Let P be an instance, and c a configuration, then c can be generated by P if and only if $\bar{c}^R$ can lead to a solution in P^R.*

This seems trivial, but it is a very important property underlying the *mask method* and the *exclusion method* discussed in the next two chapters.

2.4.2 Unsolvability properties

The following lemmas can be used to easily identify some types of instances that have no solution:

Lemma 2.4.3 *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 2.4.4 *A solvable instance must have one pair whose top string is longer than the bottom one and another one with its bottom string longer than the top one.*

Lemma 2.4.5 *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 2.4.3 ensures that the selections can start and end somewhere in a solvable instance. Lemma 2.4.4 focuses on the balance of length, and Lemma 2.4.5 deals with the balance of elements. Although the latter lemma cannot be directly used to prove instances unsolvable, it does help to remove useless pairs in an instance which will lead to no solution whenever selected, and thus, the instance is simplified.

2.4.3 Balance effects

Balance effects were first mentioned in [10]. They are based on the observation that the final concatenated top and bottom strings should be identical, hence their lengths must be equal (*balance of length*). For a similar reason, the number of 0's and 1's in both strings should be the same too (*balance of element*). These properties seem quite obvious, but prove themselves very valuable by disclosing important properties of solutions.

Definition 2.4.3 *Let S be a string over the alphabet $\{0,1\}$ and $|S|$ be the length of S. $Count(S,0)$ and $Count(S,1)$ are the number of zeroes and ones in S respectively.*

Let P: (g_i, h_i) $(1 \leq i \leq s)$ be an instance. If we define f_i as the number of occurrences of *pair* i in the sequence $i_1 i_2 \cdots i_n$, we can construct three equations based on the balance of length, element zero, and element one, respectively:

$$\sum_{i=1}^{s}(f_i \times |g_i|) = \sum_{i=1}^{s}(f_i \times |h_i|)$$
$$\sum_{i=1}^{s}\big(f_i \times Count(g_i,0)\big) = \sum_{i=1}^{s}\big(f_i \times Count(h_i,0)\big)$$
$$\sum_{i=1}^{s}\big(f_i \times Count(g_i,1)\big) = \sum_{i=1}^{s}\big(f_i \times Count(h_i,1)\big)$$

The last two equations subsume the first one and can be used to prove instances unsolvable (when the two equations put together have no solution), or to get information about the structure of possible solutions. For example in PCP (2.6):

$$\begin{pmatrix} \texttt{11111} & \texttt{1} & \texttt{0011} \\ \texttt{0} & \texttt{1100} & \texttt{11} \end{pmatrix} \tag{2.6}$$

we can create two equations on the balance of element zero and element one:

$$\begin{cases} 2f_3 &= f_1 + 2f_2 \\ 5f_1 + f_2 + 2f_3 &= 2f_2 + 2f_3 \end{cases}$$

they can be simplified to:

$$\begin{cases} -f_1 - 2f_2 + 2f_3 = 0 \\ 5f_1 - f_2 = 0 \end{cases}$$

then we obtain the following result:

$$\begin{cases} f_1 = 2x \\ f_2 = 10x \qquad x \in \mathbb{N} \\ f_3 = 11x \end{cases}$$

If this instance is solvable, the length of its solution is

$$f_1 + f_2 + f_3 = 23x$$

which must be a multiple of 23. Actually, PCP (2.6) has an optimal solution of length 115.

A factor of solution length deduced by this method is a suitable candidate for the depth increment in an iterative-deepening search (see Section 3.1), and the proportions between the frequency of pairs can be used to prune hopeless nodes in the search (see Section 3.4). However, only in a few instances of $PCP[3]$ can we deduce meaningful factors of solution lengths in this way.

2.4.4 Isomorphisms among PCP instances

Isomorphism exists in PCP instances, and it is important to detect them to avoid redundant work. In the following, we introduce the way to generate isomorphic instances, and show how to normalize instances to a standard form, which makes it possible to remove all isomorphic instances.

Types of transformations

There are four types of transformations to get the equivalent variations of an instance:

1. **Pair Reordering**: reorder pairs in the instance.

2. **Upsidedown**: interchange the top and bottom strings in every pair.

3. **Reversal**: change every string to its reversal.

4. **Complement**: replace all 0's with 1's and vice versa in every string.

It is not hard to see that through any combination of the above four transformations on the original instance, we obtain an instance that *shares the same solvability as the original one*, and these two instances are *isomorphic* to each other. Solving any of the isomorphic instances means solving the instance.

For an instance in $PCP[s]$, there will be up to $s! \cdot 2^3$ isomorphic instances (including itself) in the worst case. So an instance in $PCP[3]$ may have as many as 48 isomorphisms! For a symmetric instance such as PCP (2.7), there will be fewer isomorphic instances.

$$\begin{pmatrix} \texttt{111} & \texttt{0} & \texttt{00} \\ \texttt{1} & \texttt{101} & \texttt{1} \end{pmatrix} \tag{2.7}$$

Cardinality of PCP subclasses

Definition 2.4.4 *$Str(w)$ is the set of all strings whose lengths are greater than 0 but no longer than w. $|Str(w)|$ is the cardinality of $Str(w)$.*

Definition 2.4.5 *$Pair(w)$ is the set of all string pairs whose strings are both in $Str(w)$ but not identical. $|Pair(w)|$ is the cardinality of $Pair(w)$.*

Definition 2.4.6 *The cardinality of $PCP[s, w]$, denoted by $|PCP[s, w]|$, is the number of nontrivial and non-redundant instances in it.*

This subsection deals with calculating the cardinality of PCP subclasses. Let's start from counting strings over the alphabet $\{0, 1\}$, then pairs and instances:

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

Tables 2.1 and 2.2 show the cardinality of some cases using the above equations. In the latter table, exact values are given for numbers up to 10^{10}, and approximate values for larger numbers.

w	1	2	3	4	5	6
$\|Str(w)\|$	2	6	14	30	62	126
$\|Pair(w)\|$	2	30	182	870	3,782	15,750

Table 2.1: Cardinality of sets of strings and pairs

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 2.2: Cardinality of 24 PCP subclasses

Normalization

Eliminating isomorphic instances is important for scanning PCP subclasses, since there will be lots of redundant instances inside. Therefore, we use a normalization process to convert all isomorphic instances to their standard forms based on a numerical scoring method.

Definition 2.4.7 *Let $S = s_1 s_2 \cdots s_n$ be a string over the alphabet $\{0, 1\}$. Its score, denoted by $Score(S)$, is $\sum_{i=1}^{n}(3^i \cdot (s_i + 1))$.*

Definition 2.4.8 *Let $P = (S_1, S_2)$ be a string pair. Its score, denoted by $Score(P)$, is $(Score(S_1), Score(S_2))$.*

We use lexicographic ordering to order $Score(P)$. For example, if $P_1 = (S_1, S_2)$ and $P_2 = (T_1, T_2)$ are two string pairs, then $Score(P_1) > Score(P_2)$ 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 using lexicographic ordering.

Definition 2.4.7 ensures that the mapping function is injective.[2] Generally speaking, there are countless mapping functions that can work here, and

[2]If two strings are not identical, their scores must be different.

we chose a simple one to perform the mapping. Using the lexicographic ordering, we can always find one instance with the highest score among all its isomorphic instances, and this one is regarded as their standard forms. When scanning through all instances of a PCP subclass, any instance different from its standard form will be eliminated.

Results of removing isomorphic instances

We conducted experiments to determine the exact number of non-isomorphic instances in 11 PCP subclasses. The results are presented in Table 2.3. 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 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 2.3: Number of non-isomorphic instances in 11 PCP subclasses

As the table shows, for the PCP subclasses of $PCP[s]$, the value *ratio* quickly approaches to $8 \cdot s!$ as the width increases. This can be justified as follows: when the width becomes larger, for an isomorphic instance generated from transformations, the chance of being identical to the original instance will become smaller. For a similar reason, when the width is fixed, the value $ratio/s!$ approaches 8 gradually as the size increases. This can be justified by comparing results from subclass $PCP[2,3]$, $PCP[3,3]$ and $PCP[4,3]$. All instances in these 11 PCP subclasses were fully scanned for their solvability. The details are given in Section 6.9.

2.4.5 Constructing new instances from known instances

Adding pairs

Suppose we have an instance P, and we change P to P' by inserting some pairs, then any solution to P is also a solution to P'. However, the reverse proposition cannot hold, because the added pairs may bring some new solutions for P'. Therefore, the optimal length of P' cannot be greater than that of P.

Replacing an element

In instance P, if all 0's are replaced with a string $0S$, where S is any string over the alphabet $\{0, 1\}$, the resulting instance P' shares the same solvability as P. Thus, they have the same optimal length if P is solvable. Similarly we can replace all 1's by a string starting with 1. This is an effective method to construct hard instances from ones with smaller sizes. Here is a simple example:

$$\begin{pmatrix} \mathtt{1101} & \mathtt{01} & \mathtt{1} \\ \mathtt{010} & \mathtt{1} & \mathtt{101} \end{pmatrix} \tag{2.8}$$

PCP (2.8) in subclass $PCP[3,4]$ has only one optimal solution of length 216. If we replace all occurrences of 0 with 00 and 01 respectively, two hard instances in $PCP[3,5]$ are created: PCP (2.9) and PCP (2.10). If we replace 0 with 01 in the reversal of PCP (2.8), with the same result as substituting 10 for 0 in PCP (2.8), we get PCP (2.11) in $PCP[3,5]$. These three new instances all have a unique optimal solution of length 216, the same as PCP (2.8). Note that the replacing strings in these new instances have been emphasized by a bold font.

$$\begin{pmatrix} 11\mathbf{00}1 & \mathbf{00}1 & 1 \\ \mathbf{00}1\mathbf{00} & 1 & 1\mathbf{00}1 \end{pmatrix} \tag{2.9}$$

$$\begin{pmatrix} 11\mathbf{01}1 & \mathbf{01}1 & 1 \\ \mathbf{01}1\mathbf{01} & 1 & 1\mathbf{01}1 \end{pmatrix} \tag{2.10}$$

$$\begin{pmatrix} 11\mathbf{10}1 & \mathbf{10}1 & 1 \\ \mathbf{10}1\mathbf{10} & 1 & 1\mathbf{10}1 \end{pmatrix} \tag{2.11}$$

Chapter 3

Search in Post's correspondence problem

Search plays a critical role in solving PCP instances and it is the key point in our objective to expedite the discovery of optimal solutions. We have stated that a brute-force search is unrealistic for many difficult instances, and thus, novel ideas need to be developed to search for solutions efficiently and effectively. This represents the two main aims that we are pursuing: search efficiency (examining as few nodes as possible to finish a task) and search speed (going through as many nodes as possible in a fixed time).

Lorentz's paper [8] mentioned two methods that worked satisfactorily in his PCP solver: iterative-deepening and the cache table. We restate them in the following sections and also express our opinions about the way they worked in our PCP solver. There were still some instances that Lorentz's solver could find solutions successfully, yet was unable to prove their optimality. This intriguing uncertainty encouraged us to explore some problem-specific properties, as shown in Chapter 2, and to invent three new methods in order to enhance the capabilities of our solver. They are the *mask method*, *forward pruning* and *bidirectional probing*.

An instance of a search problem, when formalized, consists of a state-space graph, a starting state, and one or more goal states. The question is how to find a path from the starting state to a goal state such that the path satisfies certain conditions, for example, having the minimum cost. A state-space graph is a directed graph, whose nodes are states and whose arcs represent transitions between states. A state-space graph may be implicit, where states and arcs are not given explicitly, but rules to make legal transitions are provided instead. Goal states may not be given explicitly either, but defined in terms of a goal test function.

There are many famous search problems in the literature, such as the Travelling Salesperson Problem (TSP), 15-puzzle and Rubik's cube. These problems have received considerable attention and have been extensively discussed by many researchers. PCP as a search problem is fairly new, and the infinite search space naturally places it in a formidable position. The state space of a PCP instance is given implicitly and may be infinite. Its states are configurations, and an arc from state x to y exists if and only if x can transfer to y by selecting some pair. The goal is to find a shortest non-empty path from a starting configuration (empty string) to a goal configuration (empty string too).

In general, a state-space graph may contain loops, and we normally try to detect and remove them during the search, as they are useless yet exhaust search efforts. Thus the relevant part of the graph is a DAG (Directed Acyclic Graph).

3.1 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 [11]. This algorithm has been applied to a variety of single-agent search problems and two-player games, and been combined with other algorithms, for example, heuristic search as in A^*, bidirectional search and alpha-beta search.

This algorithm can be briefly described as follows: first, a starting *depth threshold* is set and the state space is searched to such a threshold. If no solution is found, the threshold is augmented by a *depth increment*. Then a new search is performed. The above process is repeated until solutions are discovered or the depth threshold reaches a final threshold.

The final depth threshold acts as a stopping condition, preventing the search process from running forever. Therefore, when the whole search process finishes, either the optimal solution has been found, or a conclusion is drawn that no solution exists up to the final depth threshold.

Initially, we have no clue about the optimal length of an instance. Therefore, if we simply search to a predefined depth, for instance 300, while actually its solution length is much smaller, for instance 100, then it is very likely that the search process will fall into a pitfall and run a very long time in vain before finding the solution. Iterative deepening can help to alleviate or avoid the trap. For example, we may start at the depth threshold of 20, and if no solution is found and the final threshold is not reached, we increase the threshold by 20 and restart the search. If the average branching factor of a search tree is very large, then the number of the revisited nodes is very small compared to the number of the nodes that have to be visited. Since the average branching factors in PCP instances with smaller sizes are not very large (typically less than 2), a much larger increment is employed in our PCP solver, compared to the increment of 1 or 2 typically adopted in other search problems.

Depth-first iterative-deepening functions like a search framework in solving PCP instances. Figure 3.1 illustrates C++ pseudo-code of this algorithm.

In the framework, the parameter `Depth_Increment` is a very important parameter. The increment cannot be too small, otherwise the solver may do excessive redundant work, because all nodes visited in an iteration will be revisited in the next iteration. On the other hand, if the increment is very 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.

In section 2.4.3, it has been shown that the balance effects may be applied to deduce a factor of the solution length. If it is used as the starting depth and the increment, it is guaranteed that no node deeper than the optimal solution will be visited. In the extreme case where the factor is 1, the above assertion is obviously correct, though the search is quite inefficient. In the other extreme

```
// predefined global parameters
int Final_Threshold;
int Starting_Threshold;
int Depth_Increment;

void Iterative_search(PCPInstance *pcp)
{
    int threshold = Starting_Threshold;
    while(1)
    {
       if (threshold > Final_Threshold)
       threshold = Final_Threshold;
       Search_to_depth(pcp, threshold);

       // solutions found
       if (Solution_found()) break;

       // no solution found, and final threshold reached
       if (threshold == Final_Threshold) break;

       threshold += Depth_Increment;
    }
}
```

Figure 3.1: Depth-first iterative-deepening routine in the PCP solver

where the factor is just the optimal length, such a precise estimate will ensure that one search iteration is enough and no node is visited more than once, which is the ideal situation that we are hoping for. Therefore, the factor is used only when it is large enough, or in other words, comparable to the default increment. But as pointed out, balance effects are effective only in a few instances in $PCP[3]$.

3.2 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 no less than that of the old one. Since we focus on optimal solutions, whenever this case emerges, we can simply prune the new configuration. This operation can be justified through the following two observations:

1. If a configuration at depth k is unable to lead to a solution within a depth threshold, then the same configuration at depth $t \geq k$ cannot lead

to a solution within the threshold either.

2. If a configuration at depth k leads to some solutions (may be optimal) within a depth threshold, then the same configuration at depth $t > k$ cannot lead to an optimal solution.

As the depth threshold may be infinity, these observations can be generalized to the depth-unbounded case. Care must be taken that while a deeper revisited node can be safely pruned, a shallower revisited one must be fully examined, since it still has a chance to reach an optimal solution while the deeper one does not.

Since a deeper revisited node is useless, we do not need to reexamine all of its descendants. Instead, it can be pruned once its identity is determined. 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.

3.3 Mask method

Is it possible to decide that a configuration in an instance has no hope of reaching a solution by merely checking its position (in the top or bottom)? The answer is encouragingly positive. First, let's see definitions concerning *critical configuration* and *masks*.

Definition 3.3.1 *In an instance, a configuration in the top (bottom) can be turned* upside-down *if by selecting some pair, the newly generated configuration* *is in the bottom (top). When no configuration in the top (bottom) can be turned upside-down, this instance has a* top-to-bottom (bottom-to-top) mask*.*

Definition 3.3.2 *A* critical configuration *in an instance is a non-empty configuration that can be fully matched by some pair (configuration becomes empty) or be turned upside-down.*

It is apparent that an instance without critical configurations can never lead to a solution.

Definition 3.3.3 *A configuration is* valid *for an instance if it can be generated through a sequence of selections from the empty configuration.*

Definition 3.3.4 *An instance has a* top (bottom) mask *if any configuration in the top (bottom) is either invalid or cannot lead to a solution.*

Corollary 3.3.1 *If in an instance, all ending pairs have their top (bottom) strings longer, and it has a top-to-bottom (bottom-to-top) mask, then it has a top (bottom) mask.*

In the following, we will explain how Definition 3.3.4 is used to detect masks. The use of Corollary 3.3.1 is similar, and its explanation is omitted for conciseness.

The mask method uses the possibility of ending the state of a configuration in one position (top or bottom) to prune configurations at the beginning, which links critical configurations to general configurations. *If an instance has no valid critical configuration in the top, then any generated configuration in the top will not lead to a solution.* So it has a top mask. Similarly, a bottom mask exists if no valid critical configuration can be in the bottom. Therefore, the task is to find all critical configurations and test if they can be generated.

All possible critical configurations can be found by enumeration. The testing procedure for validity of configurations can also be automated, because by Lemma 2.4.2 the question of whether one configuration in an instance can be generated can be nicely converted to another question about whether the turnover of the reversal of the configuration can reach a solution in the reversal of this instance. The following explains how to discover the top mask in PCP (3.1), whose reversal is PCP (3.2).

$$\begin{pmatrix} \texttt{01} & \texttt{00} & \texttt{1} & \texttt{001} \\ \texttt{0} & \texttt{011} & \texttt{101} & \texttt{1} \end{pmatrix} \tag{3.1}$$

$$\begin{pmatrix} \texttt{10} & \texttt{00} & \texttt{1} & \texttt{100} \\ \texttt{0} & \texttt{110} & \texttt{101} & \texttt{1} \end{pmatrix} \tag{3.2}$$

At first, we need to find all critical configurations in the top. Since they could either be fully matched or turned upside-down, any possibly matched pair must have a longer bottom string than its top one. In PCP (3.1), the candidate 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. string 10, which can be fully matched by *pair 3*. Secondly, we check whether the configuration 10 in the top can possibly be generated by PCP (3.1), or equivalently, whether its reversal, 01 in the bottom in PCP (3.2), can lead to a solution. Actually it cannot be properly matched by any pair in PCP (3.2). Hence in PCP (3.1), no valid critical configuration in the top exists. It has a top mask.

For some instances, the mask method is an effective tool to find their optimal solutions. Take PCP (3.1) 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 us to quickly find the unique optimal solution of length 160. If we did not know this fact, *pair 1* can be chosen as the starting pair and a huge useless search space would have to be explored before concluding that the optimal length is 160. That's 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.

During the search process, the transformation when a configuration is turned upside-down is called an *oscillation*. From the number of possible critical configurations in one instance, we can infer an upper bound on the number of oscillations in possible solutions. This information can be useful for finding masks or pruning branches. In our experiments, we seldom encountered an instance oscillating more than once, and oscillations most occurred in the early or late phase of search.

3.3.1 GCD enhancement

The step to prove critical configurations invalid can be strengthened by using a new checking rule, namely *GCD* (Greatest Common Divisor). If the length differences of all pairs have a greatest common divisor d, then the length of every possibly generated configuration must be a multiple of d.[1] Consider the following PCP (3.3) as an example, its GCD of all length differences is 2. Although we can find a critical configuration 0 in the bottom, which can be turned upside-down by *pair 2*, it is invalid since it is not a multiple of 2. As a result, PCP (3.3) can be proven to have a bottom mask, revealing that the starting selection must be *pair 2*. Finally, within several steps of enumeration, we can easily prove it has no solution.

$$\begin{pmatrix} \texttt{111} & \texttt{001} & \texttt{1} \\ \texttt{001} & \texttt{0} & \texttt{111} \end{pmatrix} \tag{3.3}$$

The above example uses the difference of length, and similarly, we can use the difference of element 0 or element 1. For example, if the difference of the number of occurrences of element 1 in the two strings of any pair has a GCD d, then the number of element 1 in any valid configuration should be a multiple of d. This property can also be used to disprove the validity of configurations.

3.4 Forward pruning

Similar to search algorithms such as A^* and IDA^* [13] [11], a heuristic function of a configuration in a PCP instance can also be calculated and used as a lower bound of the optimal length (for an unsolvable instance, its optimal length is infinity). Heuristic search algorithms can use heuristics to choose the most promising nodes to explore first, and to stop nodes that violate constraints at the same time. In this problem, heuristics are valuable to prune those nodes that cannot lead to a solution within a given threshold. A heuristic value of a

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

configuration is an estimate of how many more selections are needed at least before reaching a solution. When the heuristic value of one configuration added to its depth exceeds current depth threshold, this configuration definitely has no hope of reaching a solution within that threshold. Hence we can reject it even if it is still far away from the threshold. Since the heuristic function is admissible, the pruning is safe and will not hurt the optimality of solutions. A typical resulting search tree is illustrated in Figure 3.2.

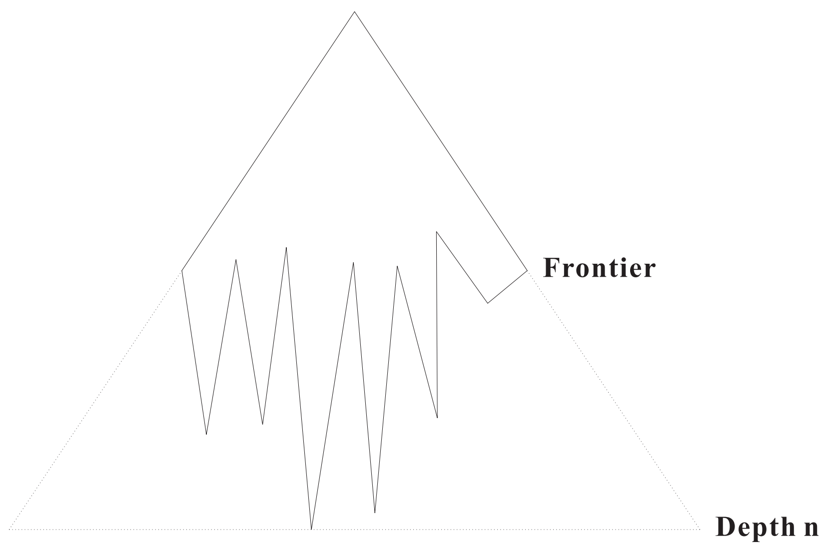

Figure 3.2: Search tree with forward pruning

In this figure, the task is to check all nodes within depth n to see whether a solution exists. However, not all those nodes must be visited. Some nodes at depth less than n that can be proven hopeless, i.e., they cannot lead to a solution within depth n, so we can prune them without exploring their descendants. As the figure shows, an unnecessary search of a significant portion of the state space can be avoided in the way.

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 whose bottom (top) string is longer. This heuristic is based on the balance of length, and similarly, we can calculate heuristics on the balance of element 0 and element 1.

A more complex heuristic can be developed, analogous to the pattern databases used to efficiently solve instances of the 15-puzzle [12]. We can pre-compute matching results for some strings as the prefixes of configurations and use them to calculate a much closer estimate of the solution length than simple heuristics.

The experimental results concerning improvements using heuristics are given in Section 6.5.

3.5 Bidirectional probing

Since a PCP instance and its reversal share the same solvability, we only need to solve one of them. Sometimes these two forms are amazingly different in terms of search difficulties, as shown experimentally in Section 6.6.

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

The mask method, forward pruning and bidirectional probing were incorporated into our PCP solver. The pseudo-code presented in figures 3.3 and 3.4 illustrates how these methods work together within simplified search routines.

3.6 Program optimizations

In order to make the search process run as fast as possible, we also put effort into code optimizations. Although this inevitably reduces the code readability and increases the difficulty of maintenance, it does accelerate search speed satisfactorily. Here we will introduce two main improvements: removing tail recursions and using our own memory allocation routines.

```
// predefined global parameters
int Comparison_Depth;

// global variable, reset to zero before each search
int node_num;

int Search_solution(PCPInstance *pcp)
{
    PCPInstance reversal_pcp, *better_pcp;
    int original_node_num, reversal_node_num;

    // check if it is unsolvable
    if (No_solution(pcp)) return -1;

    // create reversal
    Create_PCP_reversal(pcp, &reversal_pcp);

    // find masks
    Find_masks(pcp, &reversal_pcp);

    // try original direction
    Search_to_depth(pcp, Comparison_Depth);
    original_node_num = node_num;

    // try reverse direction
    Search_to_depth(&reversal_pcp, Comparison_Depth);
    reverse_node_num = node_num;

    // choose the easier one to search
    if (original_node_num <= reverse_node_num)
        better_pcp = pcp;
    else better_pcp = &reversal_pcp;
    Iterative_search(better_pcp);
    return 1;
}

void Search_to_depth(PCPInstance *pcp, int threshold)
{
    Config config;
    Set_empty(&config);
    node_num = 0;
    Solve_config(pcp, &config, 1, threshold);
}
```

Figure 3.3: Search framework in the PCP solver

```
// predefined global parameters
int Length_Increment;

// global variable
Cache cache;

void Solve_config(PCPInstance *pcp, Config *config, int depth, int threshold)
{
    int i, sel_num, selections[MAXSIZE];

    // test if exceeding the threshold
    if (depth > threshold)  return;

    // try forward pruning
    if (Heuristic_pruned(pcp, config, threshold)) return;

    // generate all selections
    sel_num = Generate(selections);

    Config *newConfig = Allocate(length(config)+Length_Increment);
    for (i=0; i<sel_num; i++)
    {
        // get a duplicate
        Copy(new_config, config);

        // choose pair selection[i] and update the config
        Update_config(pcp, selection[i], new_config);

        // check masks
        if (Mask_pruned(new_config)) continue;

        // check in the cache and update it if config not pruned
        if (Look_up_cache(cache, new_config) == PRUNED) continue;
        else Update_cache(cache, new_config);

        // if solution found, update threshold and continue searching
        // this step ensures to find all optimal solutions
        if (Is_Empty(new_config))
        {
            threshold = depth;
            Update_solution_info();
            continue;
        }
        Solve_config(pcp, new_config, depth+1, threshold)
    }
    Free(newConfig);
}
```

Figure 3.4: Solving configuration routine in the PCP solver

3.6.1 Tail recursion removal

The main CPU resources are spent in the recursive function `Solve_config` in Figure 3.4, which forms a bottleneck to search efficiency. A minor improvement in this function can result in big savings during the search. Our first step is to remove some of these recursions.[2] There are two types of tail recursions:

1. If only one pair can be selected to match a configuration, we can simply jump back to the starting point of the function after the configuration is updated.

2. If more than one pair can be selected to match a configuration, when the last possible pair is tried, the search process will update the configuration and jump back instead of calling the recursive function.

The significance of this method lies in the decrease of unnecessary stack operations involved in function calls. When the search space is very large, the savings can be aggregated to produce a big improvement. In our experiments, the solving time was decreased by 11% on average by using this method.

There is a trade-off between readability and efficiency of the code, and for this reason, we do not remove all recursions but only those easily done.

3.6.2 Memory allocation

It is easy to see that the memory space used in every configuration during the search process is not of fixed size. If we employ the dynamic memory allocation provided by the operating system, for example, the standard library function `malloc()`, it will exhaust too many CPU resources in the running time with frequent operations of allocating and freeing of memory. Therefore, we designed our own memory allocation routines specifically catering to PCP configurations.

A large enough memory block is statically allocated during program initialization and is used to simulate stack operations. Then only simple operations are needed to allocate the memory space for new configurations and reclaim

[2]This idea was first mentioned in [8].

them when they are obsolete. In the experiment, this method resulted in a 15% improvement on search speed compared to using the standard library function `malloc()` and `free()`.

Chapter 4

Identifying unsolvable PCP instances

As shown in Chapter 2, many PCP instances have no solution, but methods to prove instances unsolvable may vary a lot. For some instances, it is quite obvious that they have no solution, while for others, it may be very difficult to prove their unsolvability. What's more, there are still some instances that are still unsolved, yet the search results show they are very likely to have no solution.

When scanning all instances in a specific PCP subclass to dig out difficult instances, the PCP solver will frequently meet unsolvable instances or those seemingly unsolvable instances that exhaust lots of search time but yield no result. Thus proving instances unsolvable is a hurdle lying before us that urgently needs to be overcome.

Lorentz's paper [8] mentioned three types of filters used to discover instances with no solution, namely *prefix/postfix filter*, *length balance filter*, and *element balance filter*. They can be considered as the direct uses of Lemmas 2.3, 2.4 and 2.5 in Chapter 2. This simple method is an amazingly effective way to identify a great percentage of unsolvable instances for subclasses of smaller sizes and widths. The scanning results and the effectiveness of filters will be shown in Chapter 6.

However, there are still lots of instances that can successfully pass filters but have no solution. In such a situation, the search methods discussed in the previous chapter may serve as an unsolvability proof. Furthermore, the effort of tackling unsolved instances has resulted in five new disproof methods: *mask method*, *exclusion method*, *group method*, *symmetry method* and *pattern method*. They will be explained in the following sections. Please be advised that these methods should not be treated separately. In many cases, only when several methods are used together can an instance be proven unsolvable.

4.1 Search as an unsolvability proof

If the state space of an instance is finite, an exhaustive search can be performed to decide whether the instance has a solution or not. But this approach is based on the assumption that all revisited nodes can be detected. If a PCP solver fails to detect some revisited nodes, the state space may become infinite from the horizon of the solver, and thus, the searched instance remains unsolved and no meaningful result will come out. For this reason, the cache must be large enough to hold the whole state space of an instance if we expect to use exhaustive search to prove its unsolvability.

When the state space is infinite, exhaustive search becomes inapplicable and cannot do any work for an unsolvability proof except for proving that no solution can be found up to a depth threshold. Therefore, some new approaches have been devised to specifically cope with instances that have an infinite search space.

4.2 Mask method

The mask method as well as the GCD enhancement discussed in Section 3.2 can help to prove an instance unsolvable. If an instance has the top mask and the generated configurations after choosing any possible staring pair are in the top, this instance definitely has no solution. Similarly, if we can prove that an instance has both top and bottom masks, then it is obviously unsolvable.

Masks of an instance and its reversal are interdependent since an instance has a top (bottom) mask if and only if its reversal has a bottom (top) mask. Besides, the masks of one instance may be discovered with the assistance of the masks of its reversal and vice versa, as we discussed in Section 3.3. Therefore, the process to find the masks of an instance and its reversal must be iterated until there is no change on the masks. The number of iterations is at most 4 since the work is to detect the existence of up to 4 masks.

4.3 Exclusion method

The exclusion method is utilized to detect pairs that will never be used when 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 this pair is useless and can be safely removed. PCP (4.1) is such an example.

$$\begin{pmatrix} \texttt{1} & \texttt{0} & \texttt{101} \\ \texttt{0} & \texttt{001} & \texttt{1} \end{pmatrix} \tag{4.1}$$

If we start from *pair 2*, then the following selections will always jump between *pair 1* and *pair 2*. The proof can be separated into three steps:

1. Since the bottom strings of *pair 1* and *pair 2* are no shorter than their corresponding top strings, configurations will always stay in the bottom if only these two pairs can be chosen.

2. Any combination of the bottom strings of these two pairs cannot have a substring of 101, which is the top string of *pair 3*. Thus when a configuration generated by these two pairs has its length greater than or equal to 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 string 10 in the bottom. But it cannot be generated by *pair 1* and *pair 2*.

Therefore, after selections start at *pair 2*, every configuration will stay in the bottom and selections can only be *pair 1* or *pair 2*. No selection of *pair 3* can be made. Thus, after *pair 2* is chosen, we only need to solve an instance consisting of *pair 1* and *pair 2*. This new instance never leads to a solution because of the length balance. Hence starting at *pair 2* is hopeless.

Combined with the fact that this instance has a top mask, we successfully prove it unsolvable.

4.4 Symmetry method

From various properties we discussed in Chapter 2, particularly those related to reversal properties, we can get the following lemmas on relations between starting and ending pairs.

Lemma 4.4.1 *An instance can start at pair i if and only if its reversal can end at pair i.*

Lemma 4.4.2 *In the solution of an instance, if both its starting pair and ending pair have top (bottom) strings longer, then configurations must oscillate an odd number of times.*

Starting and ending pairs of an instance can be found with the help of Lemma 4.4.1. Combined with the bottom-to-top mask or top-to-bottom mask (see Section 3.3), some hopeless starting pairs can be excluded if no oscillation can happen. When there are no other starting pairs available for an instance, it can thus be proven unsolvable.

The symmetry method utilizes the above procedure to find unsolvable instances, and the following is an example to solve PCP (4.2).

$$\begin{pmatrix} \texttt{111} & \texttt{11} & \texttt{00} \\ \texttt{010} & \texttt{111} & \texttt{0} \end{pmatrix} \tag{4.2}$$

First we can use the exclusion method to find that if we start at *pair 3*, both *pair 1* and *pair 2* will be excluded, thus this starting point is eliminated. Now the starting pair can only be *pair 2*. As this instance is identical to its reversal, by using Lemma 4.4.1, this instance can only end at *pair 2* as well, so it has the same starting and ending pair. Since this pair has its bottom string longer, after starting at *pair 2* we must turn configurations from bottom to top somewhere to make it possible to end at *pair 2*. Yet judging from the length difference, this instance has a bottom-to-top mask, revealing no possibility of turning upside-down. So starting at *pair 2* is also hopeless. In sum, PCP (4.2) has no solution.

4.5 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 another word, if any character in the group is matched after 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.

$$\begin{pmatrix} \mathtt{011} & \mathtt{10} & \mathtt{0} \\ \mathtt{1} & \mathtt{0} & \mathtt{010} \end{pmatrix} \tag{4.3}$$

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* in PCP (4.3), 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:

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

$$g = \mathtt{01}$$

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*. As there are no other possible starting pairs, PCP (4.3) is unsolvable.

4.6 Pattern method

If a configuration has a prefix α and any possible path starting from it will always generate a configuration having prefix α after some steps, then this prefix cannot be completely removed whatever selections are made, and 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. Figure 4.1 gives an example of a prefix pattern, where arrow-headed lines represent transformations after a finite number of selections. Note that configurations never become empty in the middle of transformations.

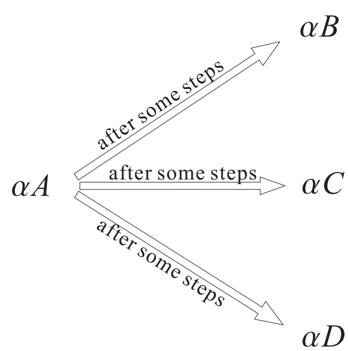

Figure 4.1: Example of a prefix pattern

The pattern method is based on prefix patterns as well as three other types of patterns, and they are given together in Figure 4.2.

Type 1:	prefix pattern	αA	$\Longrightarrow$	αB
Type 2:	postfix pattern	$A\alpha$	$\Longrightarrow$	$B\alpha$
Type 3:	infix pattern	$A\alpha B$	$\Longrightarrow$	$C\alpha D$
Type 4:	prefix & postfix pattern	$\alpha A\beta$	$\Longrightarrow$	$\alpha B\beta$

Note: α and β are patterns consisting of non-empty strings,
A, B, C, D are substrings that can be empty.

Figure 4.2: Four types of patterns

The pattern method is a very powerful tool to prove instances unsolvable, probably because it discloses a deep property inherent in PCP. The following example illustrates how this method is effective to prove PCP (4.4) unsolvable, which has a prefix pattern of 11 in the top.

$$\begin{pmatrix} \mathtt{011} & \mathtt{01} & \mathtt{0} \\ \mathtt{1} & \mathtt{0} & \mathtt{100} \end{pmatrix} \tag{4.4}$$

For a configuration of $11A$ in the top, 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 some number of steps. The prefix cannot be removed, showing that any configuration in the top with a prefix of 11 will lead to no solution. The procedure to find a prefix in PCP (4.4) is presented in Figure 4.3. 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 solution, its left part from the line must be matched exactly by some pairs. 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.

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

Figure 4.3: Deduction of 11 prefix pattern in PCP (4.4)

It can be proven that PCP (4.4) has a bottom mask, and thus it can only start at *pair 2*. Then only *pair 1* can be selected and the choice of *pair 3* can be pruned because of the bottom mask. Now the configuration is 011 in the top, and only *pair 2* can be chosen in the next step, generating a configuration of 1101 in the top. But this configuration is hopeless because of having the prefix of 11. Therefore, PCP (4.4) has no solution.

It is quite intuitive to discover the pattern in PCP (4.4), yet to find similar patterns in other instances may not be simple. For example, Figure 4.4

illustrates the procedure to detect the prefix pattern of 000 in the top in PCP (4.5), and this prefix pattern is indispensable for the unsolvability proof of this instance.

$$\begin{pmatrix} \texttt{01} & \texttt{0} & \texttt{00} \\ \texttt{0} & \texttt{100} & \texttt{10} \end{pmatrix} \tag{4.5}$$

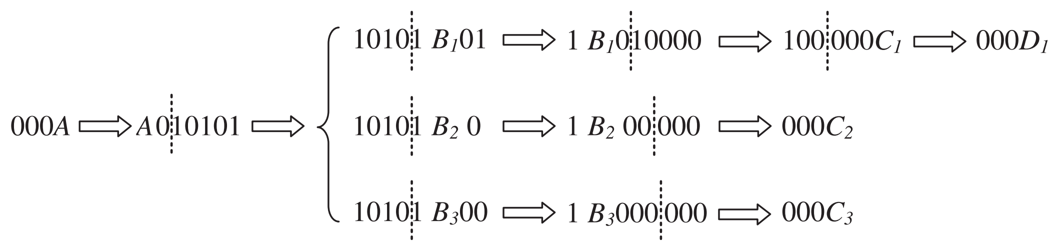

Figure 4.4: Deduction of 000 prefix pattern in PCP (4.5)

Chapter 5

Creating difficult PCP instances

A strong PCP solver enhanced by the methods and techniques discussed in Chapters 3 and 4 is essential for efficiently finding many difficult PCP instances with small sizes and widths; on the other hand, the hard instances that we have discovered naturally attracted us to find their solutions as quickly as possible, and those instances we could not solve were intriguing for us to come up with new ideas to prove their unsolvability. Thus, the three directions we are working on are interrelated, as shown in Figure 5.1.

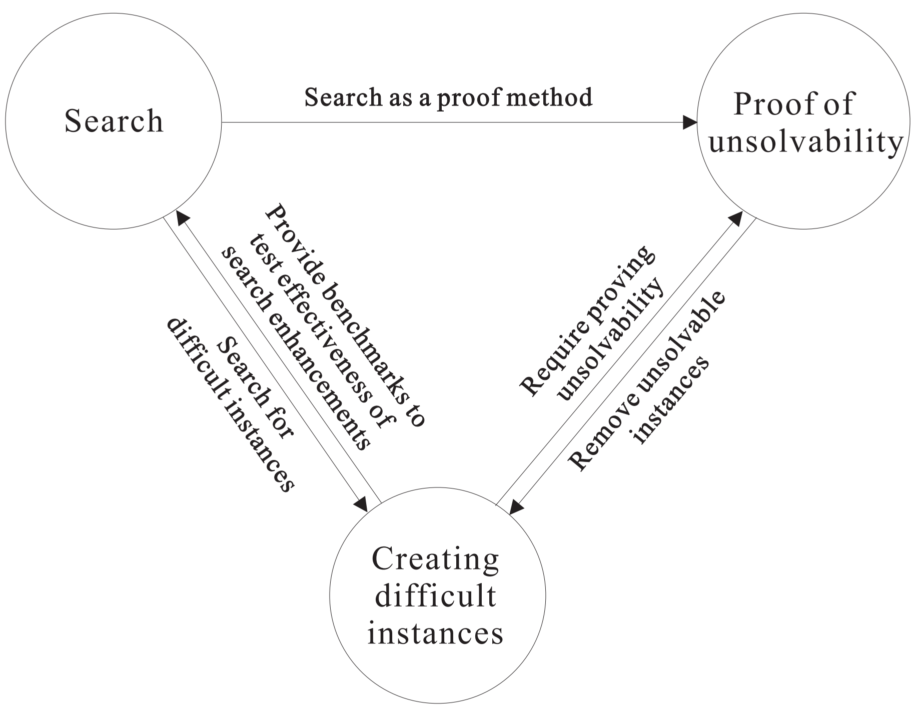

Figure 5.1: Relations between three research directions in PCP

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

5.1 Random search for hard instances

A random instance generator plus a PCP solver is a straightforward means of discovering interesting instances. The generator creates random PCP instances, which are then fed to the solver. 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.

Using several search enhancements and various methods that help to prove unsolvable instances, the program can quickly stop searching hopeless instances and find the optimal solutions to solvable instances fairly quickly.

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 pruned nodes (cutoff nodes). Using the number of visited nodes as a stopping condition makes the search process 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. Thus in the implementation, we also use the number of cutoff nodes as one of the stopping conditions.

Another 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 suggestion 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 (see Section 6.9).

5.2 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 stimulate us to find better approaches to deal with them.

It is not hard to generate all instances in a specific PCP subclass, and the issue of removing all isomorphic instances has been addressed in Section 2.4.4. But this method can only be effective for those PCP subclasses with smaller sizes and widths as in Table 2.3. For larger PCP subclasses, our program is unable to examine all of their instances in a reasonable time.

5.3 Neighborhood effect

One interesting property that we found helpful for creating difficult instances is the *neighborhood effect*. In the optimal solution of a hard instance, if one pair is very infrequently used, i.e. less than 5% of the total number of selections, it is likely that a new hard instance can be found by removing that pair or replacing it with other valid pairs. For example, PCP (5.1) with an optimal length of 120 was first found by our PCP random generator, and then, the infrequent use of pair 2 in its optimal solution led to the discovery of PCP (5.2) with an optimal length of 200. These two instances differ only in the top string of *pair 2*.

$$\begin{pmatrix} \texttt{1010} & \texttt{101} & \texttt{1} & \texttt{110} \\ \texttt{1} & \texttt{1111} & \texttt{1011} & \texttt{01} \end{pmatrix} \tag{5.1}$$

$$\begin{pmatrix} \texttt{1010} & \texttt{010} & \texttt{1} & \texttt{110} \\ \texttt{1} & \texttt{1111} & \texttt{1011} & \texttt{01} \end{pmatrix} \tag{5.2}$$

5.4 New PCP records

The random and systematic search schemes for creating difficult instances helped us to achieve new records in 4 PCP subclasses. These records are new instances with the longest optimal length over all instances 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. The records of hardest instances known are presented in Table 5.1.

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 5.1: Records of hardest instances in 5 PCP subclasses

The hardest instance in $PCP[3,3]$ was discovered by R. Lorentz and J. Waldmann independently. More details are given on the websites [14] [15].

Chapter 6

Experimental results and analysis

This chapter discusses experimental results on solving and creating difficult PCP instances. All experiments were done on 200 hard instances. 199 instances have optimal lengths no shorter than 100 and were collected from 4 PCP subclasses through the methods described in Chapter 5; the remaining test case is the hardest instance in $PCP[3,3]$. Table 6.1 lists the number of test instances in each subclass. Please refer to Appendix A for a list of these instances and their solution information. In the following, if an instance shown in this chapter also appears in Appendix A, we will give its corresponding index in the appendix and its solution length together with the instance.

PCP subclass	number
$PCP[3,3]$	1
$PCP[3,4]$	5
$PCP[3,5]$	21
$PCP[4,3]$	72
$PCP[4,4]$	101
Total	200

Table 6.1: Number of test instances in PCP subclasses

In our experiments, the default options are that the program used depth-first iterative-deepening, cache table, mask method, forward pruning based on the balance of length, bidirectional probing and program optimizations, as well as three unsolvability proof methods including the mask method, symmetry method and exclusion method. The group method and pattern method were not implemented because we could not find a general way to automate them. Table 6.2 gives details about these options.

Method	Implemented?	Parameters or comments
Depth-first iterative-deepening	Yes	depth increment = 20
Cache table	Yes	cache size = 2^{13} entries
Mask method	Yes	with GCD enhancement
Forward pruning	Yes	based on balance of length
Bidirectional probing	Yes	comparison depth = 40
Tail recursion removal	Yes	
Own memory allocation	Yes	
Symmetry method	Yes	
Exclusion method	Yes	
Group method	No	
Pattern method	No	

Table 6.2: Default configurations of the PCP solver

Our PCP solver program was implemented in C++ in about 4,500 lines of source code. The machine configuration we used for compiling and running the program is given in Table 6.3. With the default configurations of the program and the hardware, solving all 200 instances took about 1 hour.

OS	Redhat Linux 7.1
Compiler	gcc 2.96
Compiling option	-O3
CPU	Pentium III 600
RAM size	128M

Table 6.3: Machine configuration used in experiments

6.1 Search speed

We first conducted an experiment on the program's search speed, in which the solver searched to the optimal length of an instance without iterative-deepening. Search speed for an instance is measured by the number of nodes visited divided by the time spent solving it. As there were 83 instances that took less than 1 second to solve, we did not take them into account due to the lack of precision. The distribution of search speed in the remaining 117 instances is given in Figure 6.1. The final version of our PCP solver achieved a search speed of 1.38×10^6 nodes per second on average.

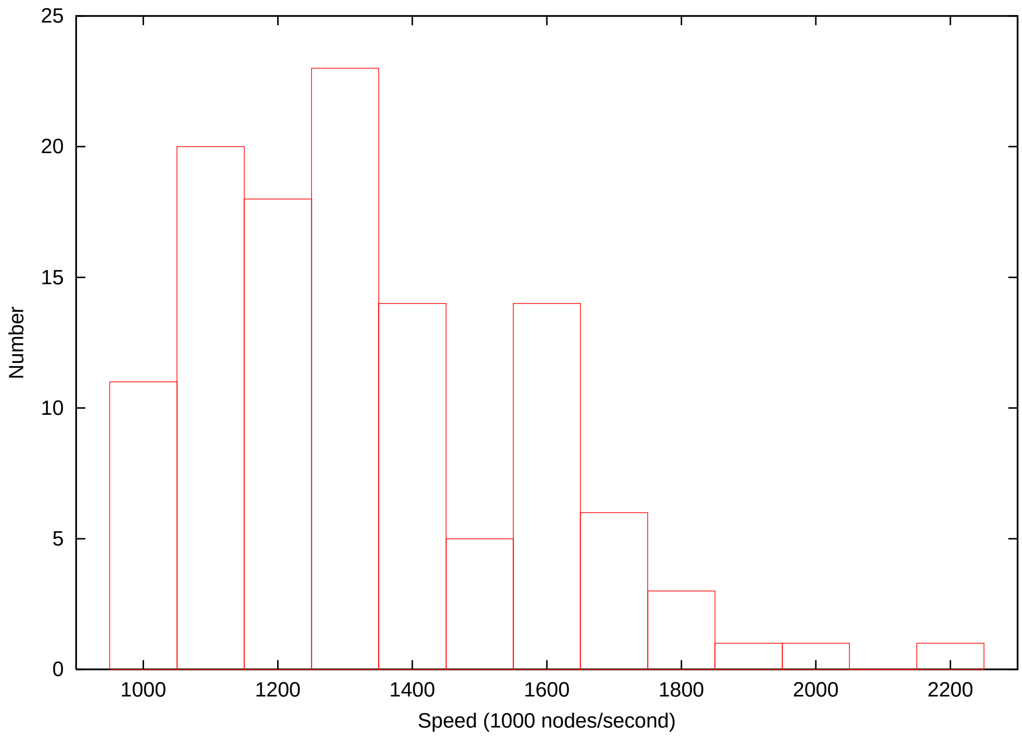

Figure 6.1: Distribution of search speed in 117 instances

The speed for different instances shows a lot of variability. PCP (6.1) has the highest speed: roughly 2.2×10^6 nodes per second; yet the search speed in PCP (6.2) is only about 1.0×10^6 nodes per second, the slowest one in the test set. We had thought that the optimal length might be the reason for such a large difference, but when looking at the whole data set, we found that many instances with quite long solutions did not have a very high search speed. We could not find a simple factor that controls the speed of an instance, and thus can only regard it as one of the special properties of PCP instances.

$$\begin{pmatrix} \texttt{111} & \texttt{011} & \texttt{10} & \texttt{0} \\ \texttt{110} & \texttt{1} & \texttt{100} & \texttt{11} \end{pmatrix} \tag{6.1}$$

$$\textit{No.28 optimal length} = 302$$

$$\begin{pmatrix} \texttt{1111} & \texttt{1110} & \texttt{1} & \texttt{0} \\ \texttt{110} & \texttt{11} & \texttt{10} & \texttt{1110} \end{pmatrix} \tag{6.2}$$

No.125 optimal length $= 140$

6.2 Branching factor

Our next experiment was on the branching factor in the search, where the final depth threshold for searching an instance was set to the optimal length of that instance. Since the branching factors in the test instances are very small, all between 1.00 and 1.25, we instead calculated the 20-level branching factor, which is the branching factor to the $20th$ power. Similar to the definition of branching factor, the 20-level branching factor of depth k is the number of nodes between depth $k+1$ and $k+20$ divided by the number of nodes between depth $k-19$ and k. The formulas to calculate the 20-level branching factor and the average 20-level branching factor in our experiment are given in Figure 6.2. We chose a depth of 41 as the starting point for computing the average 20-level branching factor solely for the reason of obtaining a better precision.

$$bf_{20}(k) = \frac{Num(k+1, k+20)}{Num(k-19, k)}$$

$$average\_bf_{20} = \left(\frac{Num(61, len)}{Num(41, 60)}\right)^{\frac{20}{len-60}}$$

Notes: len is the optimal length of the instance; $Num(a, b)$ is the number of nodes between depth a and b.

Figure 6.2: Computing the 20-level branching factor

The experimental results show that the average 20-level branching factor in 200 test instances is 9.830, and therefore, the average branching factor is only 1.121. The distribution of the 20-level branching factor for the test set is presented in Figure 6.3.

The 20-level branching factor of PCP (6.3) is 67.759, the highest over all test instances. Such a high branching factor is largely due to its *pair 2*, which

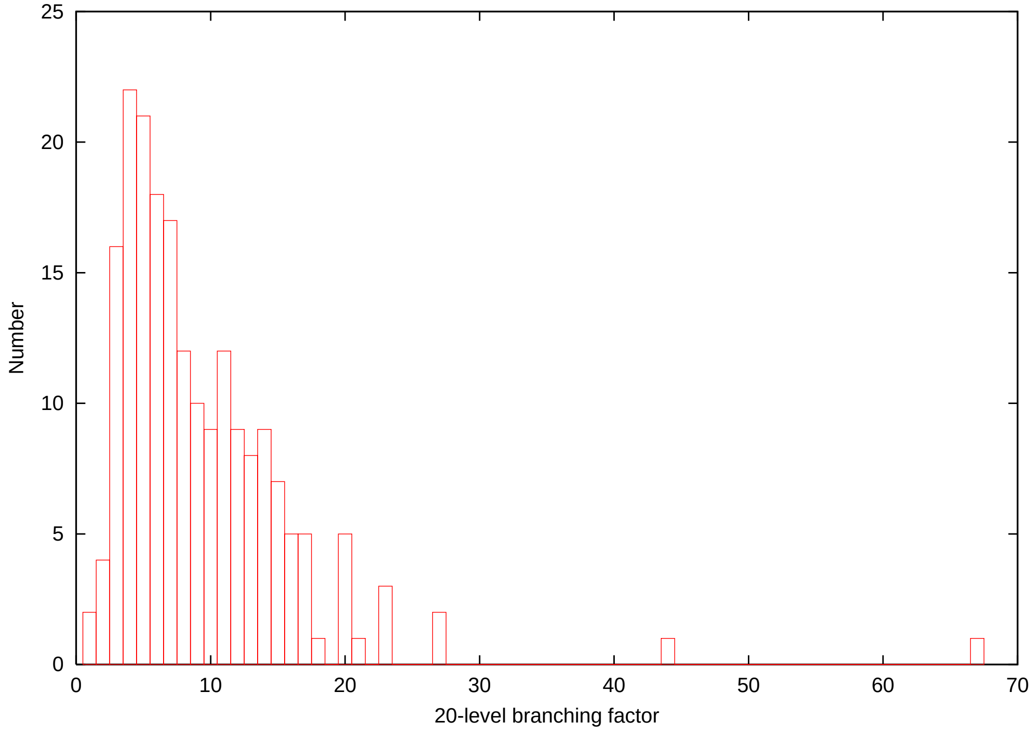

Figure 6.3: Distribution of branching factor in 200 instances

consists only of 0's. When the configuration is in the top with a prefix of 0, *pair 2* can be selected to match it, and three consecutive zeroes are appended to the end of the configuration. These three zeroes later can cause *pair 2* to be selected three times, or *pair 1* to be selected once. This increases the chance for more pairs to be chosen, and thus results in the branching factor of this instance being much larger than that of all others in our test set.

$$\begin{pmatrix} \mathtt{111} & \mathtt{000} & \mathtt{1} & \mathtt{0} \\ \mathtt{000} & \mathtt{0} & \mathtt{10} & \mathtt{11} \end{pmatrix} \tag{6.3}$$

$$\textit{No.91 optimal length} = 104$$

Over all test instances, PCP (6.4) has the smallest 20-level branching factor, only 1.544. In the process of solving this instance, configurations always stay in the bottom. As the top strings of *pair 1* and *pair 2* in this instance are very long, most configurations generated have no more than one pair to match. This results in a search tree with very few branches, and therefore, a very small branching factor.

$$\begin{pmatrix} \texttt{11111} & \texttt{01011} & \texttt{1} \\ \texttt{10} & \texttt{11} & \texttt{10101} \end{pmatrix} \tag{6.4}$$

$$\textit{No.13 optimal length} = 189$$

6.3 Cache table

The size of the cache table is an important factor influencing the performance of the PCP solver. First, using a cache table will inevitably introduce overhead in the solving process, and the larger the size, the bigger the overhead will be. Secondly, too small a cache table will cause a failure to hold some important visited nodes, and the solver cannot prune those nodes when they are revisited, so the solver needs more time to solve the instance. Therefore, the size of cache should be selected carefully.

We did experiments for the cache size ranging from 2^0, 2^1 until 2^{19} entries, and measured the time needed to solve 200 instances. The results are illustrated in Figure 6.4.

In our experiments on cache sizes as well as on depth increments in the next section, our aim is to describe the general trends of change according to different parameters by using empirical data. For example, in Figure 6.4, we can see that if the cache size is very small, the solving time will increase significantly. This phenomenon also happens when the cache size is very large. However, for a cache size between these two extremes, for example, from 8 entries to 65536 entries, the solving time is quite stable. We believe this phenomenon is largely dependent on the test instances chosen, and therefore we consider it unnecessary for deep investigations about which one results in minimum solving time and why.

Although 8 entries seem good enough to solve all test instances, we still use a larger cache table when solving a new instance, in case we need the cache table to prove the unsolvability of that instance.

We also did an experiment on the number of nodes pruned by the cache (cutoff nodes). PCP (6.5) has the largest number of cutoff nodes in its last

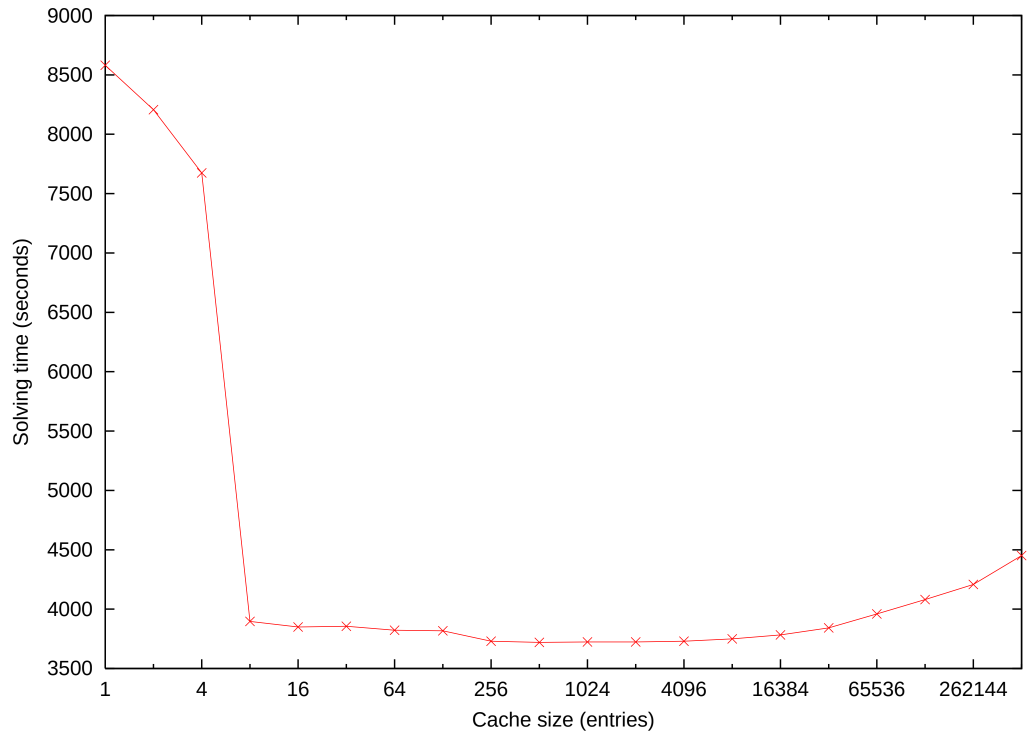

Figure 6.4: Solving time with respect to different cache sizes

search iteration of all test instances, yet the number actually is not large at all, only 28,972. This indicates that many hard instances do not have a large number of cutoff nodes, justifying our approach to use the cutoff number as a threshold to find hard instances when using the random search scheme discussed in Chapter 5.

$$\begin{pmatrix} \texttt{1110} & \texttt{1001} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{1} & \texttt{10} & \texttt{01} \end{pmatrix} \tag{6.5}$$

$$\textit{No.161 optimal length} = 118$$

6.4 Iterative deepening

The depth increment in iterative deepening is also an important search parameter. A small increment will result in too much redundant work, but a large increment may cause the solver to examine lots of useless nodes whose lengths are larger than the optimal length. Thus, there is a trade-off between benefit and overhead. We experimented with 13 different depth increments, and the solving time for them is summarized in Figure 6.5. The minimum time occurs when the increment is 20, and we chose it as the default depth increment. This setting is also convenient in data processing and analysis. In the figure, the solving time changes quite irregularly when the depth increment increases. We believe this phenomenon is largely dependent on the specific test instances we chose.

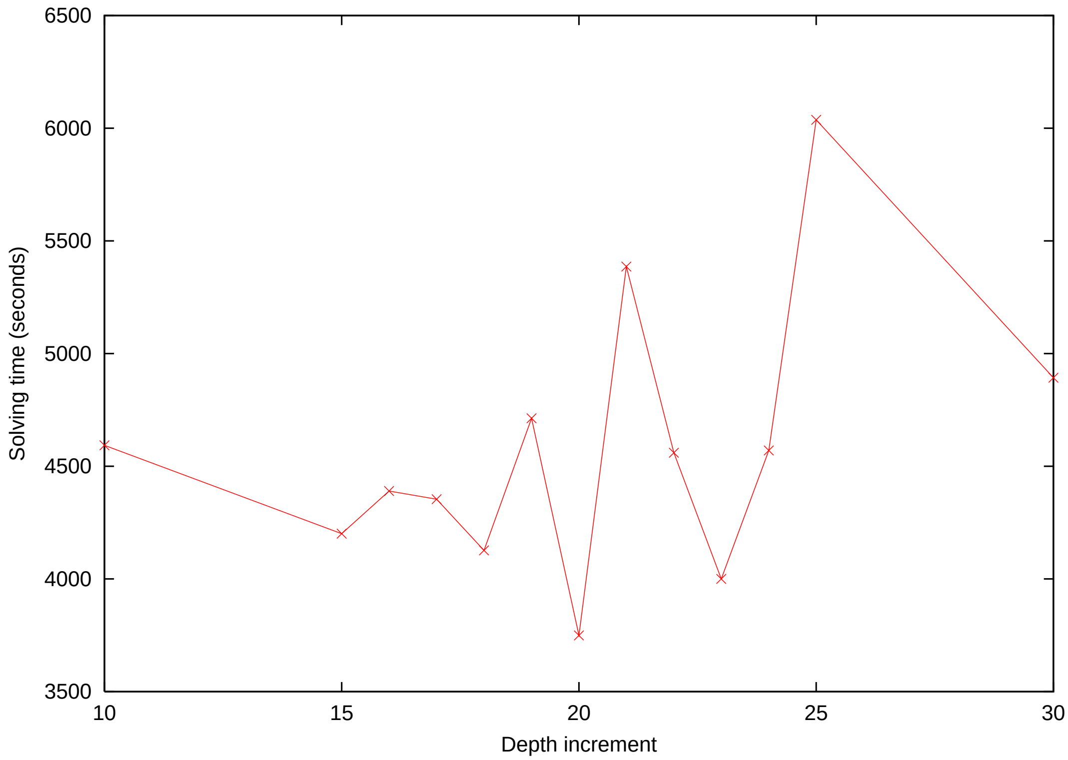

Figure 6.5: Solving time with respect to different depth increments

It is very interesting to take a look at the overhead introduced by iterative deepening. The last iteration is necessary and all other iterations are overhead. We represent the overhead by dividing total number of nodes visited in all iterations except the last one by the number of visited nodes in the last iteration. The average overhead is 19.8%, which is much larger than the reciprocal of the 20-level average branching factor we computed in Section 6.2. This is because the increment is much larger than 1, and the penultimate iteration may have its depth threshold very close to the optimal length, thus incurring considerable overhead.

6.5 Forward pruning

We implemented two types of admissible heuristic functions to prune hopeless nodes. The first one is based on the balance of length. The depth heuristic of a configuration is computed by its length divided by the maximum length difference of all pairs. Therefore, if the depth heuristic of a configuration added by its current depth exceeds the depth threshold, this configuration can be pruned. The second type of heuristic functions is based on the balance of element, where the depth heuristic of a configuration is computed as the number of one element (0 or 1) in it divided by the maximum difference of this element in all pairs (see Section 3.4). This type will generate two heuristic values, and the larger one is used for pruning. Note that the heuristic values can round up to their next larger integers, and since they never overestimate, the pruning is safe.

We did three separate experiments on forward pruning, namely, only using the first type of heuristic, only using the second type and using both types. The results are shown in Table 6.4.

Pruning method	Solving time (seconds)
on balance of length	3749
on balance of element	19026
on balance of both	4593

Table 6.4: Solving time with respect to different forward pruning methods

We tried to compare the improvement achieved by the heuristic pruning compared with the situation when no pruning is done, but we could not finish the task since it would take too much time. PCP (6.6) 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 improvement in the solving time!

$$\begin{pmatrix} \texttt{11011} & \texttt{110} & \texttt{1} \\ \texttt{0110} & \texttt{1} & \texttt{11011} \end{pmatrix} \tag{6.6}$$

No.18 optimal length $= 120$

Table 6.4 shows clearly that only using the heuristic on the balance of length excels in the performance. This heuristic is simple and elegant, but is also very powerful and introduces very little overhead. This result also reminds us that it is not always true that a more complicated approach will gain a better performance.

6.6 Bidirectional probing

The search difficulty in an instance and its reversal may be extremely different, though they are isomorphic to each other. We performed an experiment on 200 test instances by searching in two directions at depth 40 separately, and then compared the number of visited node in each direction. The distribution of the difference of the search difficulty in two directions is given in Figure 6.6.

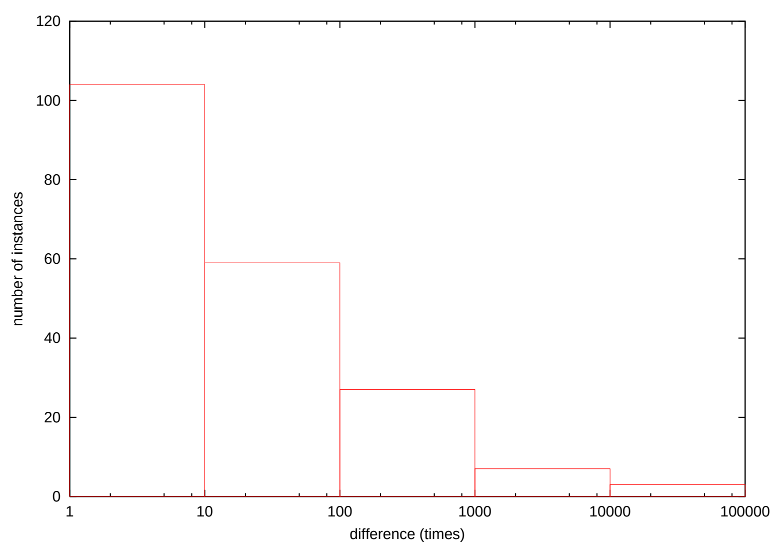

Figure 6.6: Distribution of the difference of search difficulty in two directions

In the test set, PCP (6.7) has the largest difference. Up to depth 40, search in PCP (6.7) is more than 15,000 times harder than search in its reversal in terms of visited nodes.

$$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{1} & \texttt{0} \\ \texttt{0} & \texttt{101} & \texttt{00} & \texttt{11} \end{pmatrix} \tag{6.7}$$

No.61 optimal length $= 134$

Consider that searching to depth 40 has already made such a big difference, if searching to depth 100 or more, the difference will explode exponentially, and will make it unrealistic to solve the instance by searching in the harder direction. This fact clearly demonstrates how important bidirectional probing is.

6.7 Program optimizations

In this section, we report the benefits brought by our configuration allocation routines and tail recursion removal. The specific allocation routines gain a 15.4% improvement in solving time and the tail recursion removal decreases the solving time by 11.4%. The test results are given in Table 6.5.

Method	Solving time (seconds)
no own memory allocation	4328
no tail recursion removal	4176
both are used	3749

Table 6.5: Solving time with respect to program optimizations

6.8 Solution structures

The most difficult instance in terms of solving time in our test instances is PCP (6.8), which needs 598 seconds to solve. In its last search iteration, 621,887,191 nodes are visited, and 22 nodes are pruned.

$$\begin{pmatrix} \texttt{110} & \texttt{001} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{0} & \texttt{01} & \texttt{11} \end{pmatrix} \tag{6.8}$$

No.66 optimal length $= 131$

The 200 test instances altogether have 226 optimal solutions, and Table 6.6 shows the distribution of the number of optimal solutions in these instances.

number of solutions	number of instances
1	179
2	17
3	3
4	1
Total	200

Table 6.6: Distribution of the number of optimal solutions

The only instance that has 4 optimal solutions is PCP (6.9).

$$\begin{pmatrix} \texttt{11011} & \texttt{0110} & \texttt{1} \\ \texttt{1} & \texttt{11} & \texttt{110} \end{pmatrix} \tag{6.9}$$

$$\textit{No.12 optimal length} = 190$$

Now let's consider how the configurations change along the optimal solution. These configurations actually constitute the optimal solution, and we call them *solution configurations*. Figure 6.7 gives a typical form of solution configurations, which belongs to PCP (6.10). The starting and ending configurations are both the empty strings, and are therefore omitted in the figure.

While it seems no regularity exists in the solution configurations of PCP (6.10), the solution configurations in PCP (6.11) given in Figure 6.8 are very special.

If we only consider how the length of configurations changes, we can find the waves that they generate are quite regular and have many forms similar to triangles. These kind of waves are really amazing and hard to forget.

We did experiments on two properties of solution configurations, that is, the maximum length[1] and the number of local maxima on length. Local maxima are those configurations whose lengths are larger than their local neighboring configurations. For example, PCP (6.11) has 10 such local maxima.

The maximum length of solution configurations of all test instances is only 79, encountered in PCP(6.12). For this reason, it may be true from the statistical point of view that very long configurations are unlikely to lead to a

[1]Please note that configuration length and solution length are different.

$$\begin{pmatrix} \mathtt{1110} & \mathtt{1010} & \mathtt{01} & \mathtt{1} \\ \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1101} \end{pmatrix} \tag{6.10}$$

No.117 optimal length $= 152$

```
101
011101
11010
10101101
11011
10111101
01111011101
1110111010
1110100
1101001101
1010011011101
0110111011
101110110
011101101101
11011011010
10110110101101
01101101011011101
1011010110111010
0110101101110101101
101011011101011010
010110111010110101101
01101110101101011010
1011101011010110100
0111010110101101001101
110101101011010011010
101011010110100110101101
010110101101001101011011101
01101011010011010110111010
1010110100110101101110100
1101001101011011101001
1010011010110111010011101
0110101101110100111011
101011011101001110110
010110111010011101101101
01101110100111011011010
1011101001110110110100
0111010011101101101001101
110100111011011010011010
101001110110110100110101101
011101101101001101011011
11011011010011010110110
10110110100110101101101101
01101101001101011011011011101
1011010011010110110110111010
0110100110101101101101110101101
101001101011011011011101011010
011010110110110111010110101
10101101101101110101101010
01011011011011101011010101101
0110110110111010110101011010
101101101110101101010110100
011011011101011010101101001101
10110111010110101011010011010
01101110101101010110100110101101
1011101011010101101001101011010
0111010110101011010011010110101101
110101101010110100110101101011010
101011010101101001101011010110101101
110101011010011010110101101011011
101010110100110101101011010110111101
101101001101011010110101101111011
011010011010110101101011011110111101
10100110101101011010110111101111010
01101011010110101101111011110101
1010110101101011011110111101010
1101011010110111101111010101
1010110101101111011110101011101
1101011011110111101010111011
1010110111101111010101110111101
1101111011110101011101111011
1011110111101010111011110111101
0111101111010101110111101111011101
111011110101011101111011110111010
111101010111011110111101110100
111010101110111101111011101001101
10101110111101111011101001101
111011110111101110100110101
111101111011101001101010
111011110111010011010101101
111101110100110101011010
111011101001101010110101101
111010011010101101011010
110100110101011010110101101
101001101010110101101011011101
011010101101011010110111011
10101011010110101101110110
10110101101011011101101
01101011010110111011011101
1010110101101110110111010
1101011011101101110101
1010110111011011101011101
1101110110111010111011
1011101101110101110111101
0111011011101011101111011101
110110111010111011110111010
101101110101110111101110101101
011011101011101111011101011011101
10111010111011110111010110111010
01110101110111101110101101110101101
1101011101111011101011011101011010
1010111011110111010110111010110101101
1110111101110101101110101101011011
1111011101011011101011010110110
1110111010110111010110101101101101
1110101101110101101011011011010
1011011101011010110110110100
0110111010110101101101101001101
101110101101011011011010011010
011101011010110110110100110101101
11010110101101101101001101011010
10101101011011011010011010110101101
11010110110110100110101101011011
10101101101101001101011010110111101
11011011010011010110101101111011
10110110100110101101011011110111101
01101101001101011010110111101111011101
1011010011010110101101111011110111010
0110100110101101011011110111101110101101
101001101011010110111101111011101011010
011010110101101111011110111010110101
10101101011011110111101110101101010
01011010110111101111011101011010101101
0110101101111011110111010110101011010
101011011110111101110101101010110100
010110111101111011101011010101101001101
01101111011110111010110101011010011010
1011110111101110101101010110100110100
0111101111011101011010101101001101001101
111011110111010110101011010011010011010
111101110101101010110100110100110100
111011101011010101101001101001101001101
111010110101011010011010011010011010
101101010110100110100110100110100
011010101101001101001101001101001101
10101011010011010011010011010011010
10110100110100110100110100110101
01101001101001101001101001101011101
1010011010011010011010011010111010
0110100110100110100110101110101
101001101001101001101011101010
011010011010011010111010101
10100110100110101110101010
01101001101011101010101
1010011010111010101010
0110101110101010101
101011101010101010
111010101010101
101010101010
101010101
101011
111
```

Figure 6.7: Solution configurations in PCP (6.10)

$$\begin{pmatrix} \texttt{11100} & \texttt{0000} & \texttt{0} \\ \texttt{00} & \texttt{1} & \texttt{00111} \end{pmatrix} \qquad (6.11)$$

No.15 optimal length $= 168$

```
0111
11100111
11100
00
000111
0011100111
01110011100111
111001110011100111
111001110011100
111001110000
111000000
000000
0000000111
00000011100111
000001110011100111
0000111001110011100111
00011100111001110011100111
001110011100111001110011100111
0111001110011100111001110011100111
11100111001110011100111001110011100111
11100111001110011100111001110011100
11100111001110011100111001110000
11100111001110011100111000000
11100111001110011100000000
11100111001110000000000
11100111000000000000
11100000000000000
00000000000000
00000000001
00000011
00111
011100111
1110011100111
1110011100
1110000
0000
00000111
000011100111
0001110011100111
00111001110011100111
011100111001110011100111
1110011100111001110011100111
1110011100111001110011100
1110011100111001110000
1110011100111000000
1110011100000000
1110000000000
0000000000
00000000000111
000000000011100111
0000000001110011100111
00000000111001110011100111
000000011100111001110011100111
0000001110011100111001110011100111
00000111001110011100111001110011100111
000011100111001110011100111001110011100111
0001110011100111001110011100111001110011100111
00111001110011100111001110011100111001110011100111
011100111001110011100111001110011100111001110011100111
1110011100111001110011100111001110011100111001110011100111
1110011100111001110011100111001110011100111001110011100
1110011100111001110011100111001110011100111001110000
1110011100111001110011100111001110011100111000000
1110011100111001110011100111001110011100000000
1110011100111001110011100111001110000000000
1110011100111001110011100111000000000000
1110011100111001110011100000000000000
1110011100111001110000000000000000
1110011100111000000000000000000
1110011100000000000000000000
1110000000000000000000000
0000000000000000000000
0000000000000000001
0000000000000011
0000000000111
00000000011100111
000000001110011100111
0000000111001110011100111
00000011100111001110011100111
000001110011100111001110011100111
0000111001110011100111001110011100111
00011100111001110011100111001110011100111
001110011100111001110011100111001110011100111
0111001110011100111001110011100111001110011100111
11100111001110011100111001110011100111001110011100111
11100111001110011100111001110011100111001110011100
11100111001110011100111001110011100111001110000
11100111001110011100111001110011100111000000
11100111001110011100111001110011100000000
11100111001110011100111001110000000000
11100111001110011100111000000000000
11100111001110011100000000000000
11100111001110000000000000000
11100111000000000000000000
11100000000000000000000
00000000000000000000
00000000000000001
00000000000011
00000000111
000000011100111
0000001110011100111
00000111001110011100111
000011100111001110011100111
0001110011100111001110011100111
00111001110011100111001110011100111
011100111001110011100111001110011100111
1110011100111001110011100111001110011100111
1110011100111001110011100111001110011100
1110011100111001110011100111001110000
1110011100111001110011100111000000
1110011100111001110011100000000
1110011100111001110000000000
1110011100111000000000000
1110011100000000000000
1110000000000000000
0000000000000000
0000000000001
0000000011
0000111
00011100111
001110011100111
0111001110011100111
11100111001110011100111
11100111001110011100
11100111001110000
11100111000000
11100000000
00000000
000000000111
0000000011100111
00000001110011100111
000000111001110011100111
0000011100111001110011100111
00001110011100111001110011100111
000111001110011100111001110011100111
0011100111001110011100111001110011100111
01110011100111001110011100111001110011100111
111001110011100111001110011100111001110011100111
111001110011100111001110011100111001110011100
111001110011100111001110011100111001110000
111001110011100111001110011100111000000
111001110011100111001110011100000000
111001110011100111001110000000000
111001110011100111000000000000
111001110011100000000000000
111001110000000000000000
111000000000000000000
000000000000000000
000000000000001
000000000011
000000111
0000011100111
00001110011100111
000111001110011100111
0011100111001110011100111
01110011100111001110011100111
111001110011100111001110011100111
111001110011100111001110011100
111001110011100111001110000
111001110011100111000000
111001110011100000000
111001110000000000
111000000000000
000000000000
000000001
000011
111
```

Figure 6.8: Solution configurations in PCP (6.11)

solution. Thus when using the random search scheme to discover hard instances, we may set a threshold on the length of configurations, say 200, to prune configurations.

$$\begin{pmatrix} \texttt{11111} & \texttt{01111} & \texttt{1} \\ \texttt{11110} & \texttt{11} & \texttt{10111} \end{pmatrix} \tag{6.12}$$

$$No.23 \ optimal \ length = 112$$

In all test instances, PCP (6.13) has the maximum number of local maxima in its solution configurations, which is 80. PCP (6.14) has only 4, the minimum in all test instances.

$$\begin{pmatrix} \texttt{1101} & \texttt{0110} & \texttt{1} \\ \texttt{1} & \texttt{11} & \texttt{110} \end{pmatrix} \tag{6.13}$$

$$No.12 \ optimal \ length = 252$$

$$\begin{pmatrix} \texttt{111} & \texttt{011} & \texttt{001} & \texttt{1} \\ \texttt{110} & \texttt{1} & \texttt{111} & \texttt{100} \end{pmatrix} \tag{6.14}$$

$$No.77 \ optimal \ length = 112$$

6.9 Results of creating difficult instances

We systematically scanned seven PCP subclasses that are easy to handle. All instances in these subclasses have been completely solved. The results are shown in Table 6.7, which also gives statistics on the effectiveness of those unsolvability proof methods. Besides, we can see how small a percentage of a PCP subclass are solvable instances!

The symmetry method typically is used together with other methods, so it is not listed separately in the table. The hardest instances in terms of the longest optimal length in these 7 subclasses are presented in Table 6.8. There are three hardest instances in $PCP[2,2]$ and 5 in $PCP[4,2]$, and for brevity, we only list one of them in each subclass.

One special phenomenon we observed from the table is that the hardest instances in PCP subclasses with size 2 can all be represented in the following

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

Table 6.7: Solving results of 7 PCP subclasses

subclass	hardest instance	optimal length	number of optimal solution
$PCP[2,2]$	$\begin{pmatrix} \texttt{10} & \texttt{1} \\ \texttt{1} & \texttt{01} \end{pmatrix}$	2	1
$PCP[2,3]$	$\begin{pmatrix} \texttt{110} & \texttt{1} \\ \texttt{1} & \texttt{011} \end{pmatrix}$	4	1
$PCP[2,4]$	$\begin{pmatrix} \texttt{1110} & \texttt{1} \\ \texttt{1} & \texttt{0111} \end{pmatrix}$	6	1
$PCP[2,5]$	$\begin{pmatrix} \texttt{11110} & \texttt{1} \\ \texttt{1} & \texttt{01111} \end{pmatrix}$	8	1
$PCP[2,6]$	$\begin{pmatrix} \texttt{111110} & \texttt{1} \\ \texttt{1} & \texttt{011111} \end{pmatrix}$	10	1
$PCP[3,2]$	$\begin{pmatrix} \texttt{11} & \texttt{00} & \texttt{1} \\ \texttt{00} & \texttt{0} & \texttt{11} \end{pmatrix}$	5	2
$PCP[4,2]$	$\begin{pmatrix} \texttt{11} & \texttt{11} & \texttt{00} & \texttt{0} \\ \texttt{10} & \texttt{1} & \texttt{11} & \texttt{00} \end{pmatrix}$	5	2

Table 6.8: Hardest instances in 7 PCP subclasses

form:[2]

$$\begin{pmatrix} \texttt{1}^n\texttt{0} & \texttt{1} \\ \texttt{1} & \texttt{01}^n \end{pmatrix}$$

It is not hard to prove that the optimal length of this kind of instances is $2n$. Lorentz's paper [8] mentioned it and 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 [4]. Table 6.8 supports this conjecture with data from 5

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

subclasses.

We used the systematic method to further examine three PCP subclasses that are much harder to conquer. Table 6.9 summarizes the results from $PCP[3, 3]$ at first.

Total number	127,303
After filter	8,428
After mask	2,089
After exclusion	2,002
Solvable instances	1,968
Unsolvable instances	33
Unsolved instances	1

Table 6.9: Scanning results of subclass $PCP[3, 3]$

In Tables 6.9 and 6.10, an instance removed by the exclusion method may still have solutions, but it cannot have a *valid* solution. A solution to an instance is *valid* if in the solution all pairs of the instance are used. Since the result of solving such an instance is identical to combinations of results from instances with smaller sizes, we do not give it any further processing. Similarly, instances removed by the element balance filter may also have invalid solutions, but these solutions are of no interest to us. In addition, 32 instances are unsolved by our PCP solver, but are proven unsolvable by hand using the methods discussed in Chapter 4. The difference comes from the fact that though some methods can successfully prove several instances unsolvable, some parts of them cannot be generalized or are too complicated to implement, and thus they were not incorporated in our PCP solver. The 33 instances that could not be solved by our solver are given in Appendix B.

The only one unsolved instance in $PCP[3, 3]$ is PCP (6.15). 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 the optimal length of 75, we believe this instance is very likely to have no solution, but apparently new approaches are needed to deal with it.

$$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{01} & \texttt{110} \end{pmatrix} \tag{6.15}$$

We used a similar method to scan all instances in $PCP[3,4]$ and $PCP[4,3]$, and the results are summarized in Table 6.10. The scanning process took about 30 machine days to finish with three bounds when searching an instance:

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

This effort resulted in the discovery of 77 hard instances whose optimal lengths are no shorter than 100 (see Appendix A). 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 should 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. You can find these instances from the website [15].

	$PCP[3,4]$	$PCP[4,3]$
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 6.10: Scanning results of subclass $PCP[3,4]$ and $PCP[4,3]$

In the 72 hard instances we collected from $PCP[4,3]$, 13 instances (18.1%) have the pair purity feature (see Section 5.1), 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 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 larger than or equal to 100. The whole process took more than 200 machine days to finish. Please refer to Appendix A for these difficult instances.

6.10 Summary

In the last part of this chapter, we will give a brief summary of those search and disproof methods. All disproof methods implemented are performed before the search process begins, and bring very little overhead. Many search methods are indispensable to find the optimal solutions to some solvable instances, which means that if these methods are not employed, those instances cannot be solved in a realistic time. These important methods include the mask method, bidirectional probing and forward pruning.

Chapter 7

Conclusions and Future Work

In this thesis, we presented an experimental approach to solving instances of Post's correspondence problem. Our research contrasts nicely with the theoretical work done on this problem. The main contributions of this thesis can be summarized as follows:

1. Devise new search techniques to search for solutions of solvable PCP instances more effectively and efficiently.
2. Develop new disproof methods to prove the unsolvability of PCP instances.
3. Incorporate various search techniques and disproof methods to a PCP solver, and use the systematic and random search schemes to scan 10 PCP subclasses. This helped us to find 199 hard instances with optimal lengths no less than 100. We also set new records for the hardest instances known in subclass $PCP[3,4]$, $PCP[3,5]$, $PCP[4,3]$ and $PCP[4,4]$.

We have discovered many new characteristics and properties of PCP, and provided empirical results for solving PCP instances including the search speed, branching factor, search parameters, effectiveness of different disproof methods and solution structures. These results present a new angle on PCP, and may be helpful to investigate some theoretical issues related to this problem.

7.1 Future work

There is still much room for improvement.

First, many instances are still unsolved, especially PCP (6.15) (see page 63) which is the only one remaining unsolved in $PCP[3,3]$. New ideas are needed to conquer these instances.

Next, some further search enhancement ideas can be implemented. For example, PCP is suitable for bidirectional search. With large memory quite common nowadays, we may use it to investigate how much improvement we can achieve from a proper implementation of bidirectional search. What's more, in Section 3.4 on forward pruning, we briefly mentioned a heuristic method similar to pattern databases; we believe this method can decrease the solving time significantly if nicely built into our solver.

Two methods, the group method and pattern method, discussed in Sections 4.5 and 4.6, were not implemented due to the difficulty of finding a general way to automate the identification procedures in them. However, if they can be coded, a great portion of unsolved instances will be proved unsolvable.

Finally, we expect $PCP[3,5]$ and $PCP[4,4]$ and more PCP subclasses to be systematically examined, because this step is meaningful for finding regularities among those hardest instances and for better understanding the complexity of this problem.

Bibliography

[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, Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, 1996.

[7] T. Rado. On non-computable functions. Bell Sys. Tech. J., 41, 3, 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] J. Waldmann and H. Stamer. Neuigkeiten zum PCP, Presentation slides (in German), 2000.

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

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

[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] http://www.informatik.uni-leipzig.de/~pcp

[15] http://www.cs.ualberta.ca/~zhao/PCP

Appendix A: List of 200 Hard Instances

No.	instance	optimal length	number of optimal solutions
1	$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{110} \end{pmatrix}$	75	2
2	$\begin{pmatrix} \texttt{1101} & \texttt{0110} & \texttt{1} \\ \texttt{1} & \texttt{11} & \texttt{110} \end{pmatrix}$	252	1
3	$\begin{pmatrix} \texttt{1101} & \texttt{01} & \texttt{1} \\ \texttt{010} & \texttt{1} & \texttt{101} \end{pmatrix}$	216	1
4	$\begin{pmatrix} \texttt{1100} & \texttt{11} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{110} \end{pmatrix}$	132	1
5	$\begin{pmatrix} \texttt{1101} & \texttt{1} & \texttt{0} \\ \texttt{01} & \texttt{1011} & \texttt{1} \end{pmatrix}$	119	1
6	$\begin{pmatrix} \texttt{1101} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{110} \end{pmatrix}$	112	2
7	$\begin{pmatrix} \texttt{11101} & \texttt{110} & \texttt{1} \\ \texttt{0110} & \texttt{1} & \texttt{1011} \end{pmatrix}$	240	1
8	$\begin{pmatrix} \texttt{11011} & \texttt{110} & \texttt{1} \\ \texttt{10110} & \texttt{1} & \texttt{1101} \end{pmatrix}$	216	1
9	$\begin{pmatrix} \texttt{11101} & \texttt{101} & \texttt{1} \\ \texttt{10110} & \texttt{1} & \texttt{1101} \end{pmatrix}$	216	1
10	$\begin{pmatrix} \texttt{11011} & \texttt{011} & \texttt{1} \\ \texttt{10110} & \texttt{1} & \texttt{1101} \end{pmatrix}$	216	2
11	$\begin{pmatrix} \texttt{11011} & \texttt{0110} & \texttt{0} \\ \texttt{01100} & \texttt{0} & \texttt{011} \end{pmatrix}$	216	1
12	$\begin{pmatrix} \texttt{11011} & \texttt{0110} & \texttt{1} \\ \texttt{1} & \texttt{11} & \texttt{110} \end{pmatrix}$	190	4
13	$\begin{pmatrix} \texttt{11111} & \texttt{01011} & \texttt{1} \\ \texttt{10} & \texttt{11} & \texttt{10101} \end{pmatrix}$	189	1
14	$\begin{pmatrix} \texttt{11101} & \texttt{110} & \texttt{1} \\ \texttt{1} & \texttt{1011} & \texttt{01101} \end{pmatrix}$	182	1

15	$\begin{pmatrix} 11100 & 0000 & 0 \\ 00 & 1 & 00111 \end{pmatrix}$	168	1
16	$\begin{pmatrix} 11010 & 1 & 0 \\ 1 & 0101 & 11 \end{pmatrix}$	156	1
17	$\begin{pmatrix} 11010 & 1 & 0 \\ 1 & 10 & 11010 \end{pmatrix}$	144	2
18	$\begin{pmatrix} 11011 & 110 & 1 \\ 0110 & 1 & 11011 \end{pmatrix}$	120	1
19	$\begin{pmatrix} 11101 & 10 & 1 \\ 101 & 1 & 11011 \end{pmatrix}$	119	1
20	$\begin{pmatrix} 11101 & 110 & 1 \\ 1 & 11011 & 10 \end{pmatrix}$	119	1
21	$\begin{pmatrix} 11001 & 00 & 1 \\ 001 & 1 & 10011 \end{pmatrix}$	119	1
22	$\begin{pmatrix} 11111 & 1100 & 1 \\ 0 & 11 & 0011 \end{pmatrix}$	115	1
23	$\begin{pmatrix} 11111 & 01111 & 1 \\ 11110 & 11 & 10111 \end{pmatrix}$	112	1
24	$\begin{pmatrix} 11101 & 10 & 1 \\ 1 & 1110 & 10 \end{pmatrix}$	112	2
25	$\begin{pmatrix} 11011 & 10 & 1 \\ 1 & 1011 & 10 \end{pmatrix}$	112	2
26	$\begin{pmatrix} 11001 & 00 & 1 \\ 1 & 1100 & 00 \end{pmatrix}$	112	2
27	$\begin{pmatrix} 11101 & 101 & 1 \\ 0110 & 1 & 11011 \end{pmatrix}$	110	1

28	$\begin{pmatrix} 111 & 011 & 10 & 0 \\ 110 & 1 & 100 & 11 \end{pmatrix}$	302	1
29	$\begin{pmatrix} 110 & 100 & 1 & 0 \\ 011 & 1 & 00 & 110 \end{pmatrix}$	273	1
30	$\begin{pmatrix} 111 & 10 & 00 & 1 \\ 1 & 100 & 1 & 011 \end{pmatrix}$	240	1
31	$\begin{pmatrix} 110 & 100 & 1 & 0 \\ 001 & 1 & 10 & 110 \end{pmatrix}$	217	1
32	$\begin{pmatrix} 111 & 10 & 00 & 1 \\ 1 & 110 & 1 & 000 \end{pmatrix}$	204	1
33	$\begin{pmatrix} 111 & 101 & 00 & 1 \\ 0 & 1 & 11 & 100 \end{pmatrix}$	192	1
34	$\begin{pmatrix} 110 & 101 & 1 & 0 \\ 0 & 11 & 100 & 01 \end{pmatrix}$	183	1
35	$\begin{pmatrix} 110 & 01 & 00 & 1 \\ 10 & 00 & 1 & 110 \end{pmatrix}$	170	1

36	$\begin{pmatrix} 110 & 10 & 1 & 0 \\ 01 & 0 & 100 & 11 \end{pmatrix}$	168	1
37	$\begin{pmatrix} 110 & 101 & 1 & 0 \\ 00 & 1 & 100 & 10 \end{pmatrix}$	160	1
38	$\begin{pmatrix} 110 & 011 & 00 & 1 \\ 00 & 1 & 1 & 100 \end{pmatrix}$	160	1
39	$\begin{pmatrix} 111 & 110 & 011 & 0 \\ 110 & 000 & 1 & 011 \end{pmatrix}$	155	1
40	$\begin{pmatrix} 111 & 101 & 10 & 0 \\ 0 & 1 & 010 & 110 \end{pmatrix}$	155	1
41	$\begin{pmatrix} 110 & 101 & 100 & 1 \\ 11 & 0 & 10 & 011 \end{pmatrix}$	155	2
42	$\begin{pmatrix} 110 & 101 & 1 & 0 \\ 0 & 00 & 101 & 11 \end{pmatrix}$	155	1
43	$\begin{pmatrix} 110 & 010 & 1 & 0 \\ 10 & 1 & 00 & 001 \end{pmatrix}$	154	1
44	$\begin{pmatrix} 110 & 101 & 1 & 0 \\ 0 & 11 & 00 & 01 \end{pmatrix}$	154	1
45	$\begin{pmatrix} 111 & 111 & 10 & 0 \\ 010 & 1 & 100 & 111 \end{pmatrix}$	150	1
46	$\begin{pmatrix} 111 & 100 & 01 & 0 \\ 00 & 1 & 00 & 100 \end{pmatrix}$	149	1
47	$\begin{pmatrix} 110 & 010 & 1 & 0 \\ 01 & 0 & 100 & 110 \end{pmatrix}$	149	1
48	$\begin{pmatrix} 111 & 010 & 10 & 1 \\ 11 & 11 & 011 & 101 \end{pmatrix}$	146	1
49	$\begin{pmatrix} 110 & 11 & 01 & 0 \\ 1 & 10 & 101 & 010 \end{pmatrix}$	146	1
50	$\begin{pmatrix} 111 & 100 & 01 & 1 \\ 1 & 11 & 001 & 110 \end{pmatrix}$	144	1
51	$\begin{pmatrix} 110 & 100 & 01 & 1 \\ 10 & 11 & 0 & 011 \end{pmatrix}$	144	1
52	$\begin{pmatrix} 111 & 110 & 010 & 1 \\ 1 & 01 & 1 & 011 \end{pmatrix}$	142	1
53	$\begin{pmatrix} 111 & 100 & 10 & 0 \\ 010 & 1 & 0 & 001 \end{pmatrix}$	140	1
54	$\begin{pmatrix} 111 & 100 & 01 & 0 \\ 0 & 11 & 0 & 010 \end{pmatrix}$	140	1
55	$\begin{pmatrix} 110 & 10 & 01 & 0 \\ 10 & 11 & 0 & 100 \end{pmatrix}$	140	1
56	$\begin{pmatrix} 110 & 100 & 1 & 0 \\ 10 & 0 & 00 & 011 \end{pmatrix}$	138	1

57	$\begin{pmatrix} 110 & 101 & 1 & 0 \\ 01 & 1 & 00 & 110 \end{pmatrix}$	138	1
58	$\begin{pmatrix} 110 & 101 & 10 & 0 \\ 010 & 1 & 100 & 01 \end{pmatrix}$	136	1
59	$\begin{pmatrix} 110 & 100 & 01 & 1 \\ 001 & 0 & 1 & 101 \end{pmatrix}$	136	2
60	$\begin{pmatrix} 111 & 001 & 1 & 0 \\ 10 & 0 & 01 & 11 \end{pmatrix}$	134	1
61	$\begin{pmatrix} 110 & 1 & 1 & 0 \\ 0 & 101 & 00 & 11 \end{pmatrix}$	134	1
62	$\begin{pmatrix} 111 & 110 & 001 & 11 \\ 100 & 1 & 01 & 111 \end{pmatrix}$	132	1
63	$\begin{pmatrix} 110 & 100 & 1 & 0 \\ 01 & 1 & 00 & 110 \end{pmatrix}$	132	1
64	$\begin{pmatrix} 101 & 11 & 01 & 0 \\ 1 & 10 & 00 & 01 \end{pmatrix}$	132	1
65	$\begin{pmatrix} 111 & 10 & 0 & 0 \\ 0 & 1 & 100 & 011 \end{pmatrix}$	131	1
66	$\begin{pmatrix} 110 & 001 & 1 & 0 \\ 10 & 0 & 01 & 11 \end{pmatrix}$	131	1
67	$\begin{pmatrix} 111 & 101 & 1 & 0 \\ 10 & 1 & 100 & 1 \end{pmatrix}$	126	1
68	$\begin{pmatrix} 110 & 101 & 11 & 0 \\ 010 & 1 & 011 & 11 \end{pmatrix}$	124	1
69	$\begin{pmatrix} 111 & 01 & 00 & 0 \\ 10 & 1 & 010 & 000 \end{pmatrix}$	120	1
70	$\begin{pmatrix} 110 & 001 & 11 & 0 \\ 100 & 1 & 0 & 011 \end{pmatrix}$	120	1
71	$\begin{pmatrix} 110 & 010 & 1 & 0 \\ 00 & 0 & 011 & 10 \end{pmatrix}$	120	1
72	$\begin{pmatrix} 111 & 110 & 0 & 0 \\ 00 & 0 & 001 & 11 \end{pmatrix}$	119	1
73	$\begin{pmatrix} 110 & 011 & 01 & 0 \\ 01 & 0 & 1 & 100 \end{pmatrix}$	119	1
74	$\begin{pmatrix} 110 & 001 & 11 & 1 \\ 11 & 1 & 0 & 101 \end{pmatrix}$	118	1
75	$\begin{pmatrix} 110 & 011 & 00 & 1 \\ 10 & 0 & 11 & 011 \end{pmatrix}$	117	1
76	$\begin{pmatrix} 111 & 101 & 1 & 0 \\ 1 & 11 & 100 & 01 \end{pmatrix}$	114	1
77	$\begin{pmatrix} 111 & 011 & 001 & 1 \\ 110 & 1 & 111 & 100 \end{pmatrix}$	112	1

78	$\begin{pmatrix} \texttt{111} & \texttt{101} & \texttt{11} & \texttt{0} \\ \texttt{100} & \texttt{1} & \texttt{011} & \texttt{11} \end{pmatrix}$	110	1
79	$\begin{pmatrix} \texttt{111} & \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{000} & \texttt{1} & \texttt{01} & \texttt{110} \end{pmatrix}$	110	1
80	$\begin{pmatrix} \texttt{110} & \texttt{001} & \texttt{01} & \texttt{1} \\ \texttt{010} & \texttt{0} & \texttt{1} & \texttt{101} \end{pmatrix}$	110	3
81	$\begin{pmatrix} \texttt{110} & \texttt{10} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{010} & \texttt{01} & \texttt{110} \end{pmatrix}$	110	2
82	$\begin{pmatrix} \texttt{111} & \texttt{101} & \texttt{1} & \texttt{0} \\ \texttt{0} & \texttt{1} & \texttt{00} & \texttt{10} \end{pmatrix}$	108	1
83	$\begin{pmatrix} \texttt{110} & \texttt{011} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{1} & \texttt{001} & \texttt{011} \end{pmatrix}$	108	2
84	$\begin{pmatrix} \texttt{110} & \texttt{010} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{0} & \texttt{0} & \texttt{110} \end{pmatrix}$	108	1
85	$\begin{pmatrix} \texttt{110} & \texttt{11} & \texttt{11} & \texttt{0} \\ \texttt{1} & \texttt{01} & \texttt{00} & \texttt{110} \end{pmatrix}$	108	1
86	$\begin{pmatrix} \texttt{111} & \texttt{100} & \texttt{10} & \texttt{1} \\ \texttt{1} & \texttt{11} & \texttt{110} & \texttt{01} \end{pmatrix}$	107	1
87	$\begin{pmatrix} \texttt{111} & \texttt{110} & \texttt{101} & \texttt{1} \\ \texttt{010} & \texttt{01} & \texttt{1} & \texttt{111} \end{pmatrix}$	106	2
88	$\begin{pmatrix} \texttt{111} & \texttt{101} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{1} & \texttt{00} & \texttt{01} \end{pmatrix}$	105	1
89	$\begin{pmatrix} \texttt{110} & \texttt{001} & \texttt{01} & \texttt{1} \\ \texttt{1} & \texttt{1} & \texttt{0} & \texttt{110} \end{pmatrix}$	105	1
90	$\begin{pmatrix} \texttt{110} & \texttt{001} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{1} & \texttt{001} & \texttt{01} \end{pmatrix}$	105	1
91	$\begin{pmatrix} \texttt{111} & \texttt{000} & \texttt{1} & \texttt{0} \\ \texttt{000} & \texttt{0} & \texttt{10} & \texttt{11} \end{pmatrix}$	104	1
92	$\begin{pmatrix} \texttt{111} & \texttt{10} & \texttt{01} & \texttt{0} \\ \texttt{0} & \texttt{010} & \texttt{1} & \texttt{001} \end{pmatrix}$	104	2
93	$\begin{pmatrix} \texttt{110} & \texttt{100} & \texttt{1} & \texttt{0} \\ \texttt{001} & \texttt{1} & \texttt{00} & \texttt{110} \end{pmatrix}$	104	1
94	$\begin{pmatrix} \texttt{110} & \texttt{001} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{1} & \texttt{00} & \texttt{011} \end{pmatrix}$	104	1
95	$\begin{pmatrix} \texttt{110} & \texttt{001} & \texttt{00} & \texttt{1} \\ \texttt{01} & \texttt{0} & \texttt{11} & \texttt{001} \end{pmatrix}$	103	2
96	$\begin{pmatrix} \texttt{111} & \texttt{011} & \texttt{10} & \texttt{1} \\ \texttt{100} & \texttt{1} & \texttt{110} & \texttt{111} \end{pmatrix}$	102	1
97	$\begin{pmatrix} \texttt{111} & \texttt{110} & \texttt{011} & \texttt{1} \\ \texttt{10} & \texttt{11} & \texttt{01} & \texttt{011} \end{pmatrix}$	102	1
98	$\begin{pmatrix} \texttt{110} & \texttt{11} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{101} & \texttt{100} & \texttt{110} \end{pmatrix}$	102	1

99	$\begin{pmatrix} 110 & 11 & 11 & 0 \\ 1 & 101 & 00 & 110 \end{pmatrix}$	100	1
100	$\begin{pmatrix} 1101 & 001 & 1 & 0 \\ 11 & 0101 & 0 & 10 \end{pmatrix}$	256	1
101	$\begin{pmatrix} 1110 & 11 & 01 & 0 \\ 1 & 011 & 1 & 010 \end{pmatrix}$	206	1
102	$\begin{pmatrix} 1111 & 1101 & 10 & 1 \\ 010 & 1 & 011 & 0101 \end{pmatrix}$	200	1
103	$\begin{pmatrix} 1101 & 0110 & 11 & 0 \\ 1 & 0 & 100 & 110 \end{pmatrix}$	198	1
104	$\begin{pmatrix} 1110 & 10 & 10 & 1 \\ 0 & 111 & 01 & 1101 \end{pmatrix}$	193	2
105	$\begin{pmatrix} 1111 & 10 & 01 & 1 \\ 10 & 11 & 0 & 0101 \end{pmatrix}$	192	3
106	$\begin{pmatrix} 1110 & 1011 & 1010 & 1 \\ 011 & 0101 & 1 & 1101 \end{pmatrix}$	189	1
107	$\begin{pmatrix} 1110 & 100 & 1 & 0 \\ 10 & 00 & 1000 & 1 \end{pmatrix}$	182	1
108	$\begin{pmatrix} 1110 & 0001 & 10 & 1 \\ 0 & 1 & 1 & 011 \end{pmatrix}$	178	1
109	$\begin{pmatrix} 1111 & 101 & 010 & 1 \\ 1101 & 1 & 1011 & 10 \end{pmatrix}$	178	1
110	$\begin{pmatrix} 1111 & 101 & 010 & 1 \\ 110 & 1 & 1011 & 10 \end{pmatrix}$	168	1
111	$\begin{pmatrix} 1101 & 111 & 0 & 0 \\ 1 & 0 & 100 & 11 \end{pmatrix}$	168	1
112	$\begin{pmatrix} 1111 & 011 & 10 & 1 \\ 1010 & 0 & 1 & 0101 \end{pmatrix}$	164	1
113	$\begin{pmatrix} 1111 & 0011 & 01 & 1 \\ 100 & 0 & 1 & 1010 \end{pmatrix}$	162	1
114	$\begin{pmatrix} 1110 & 11 & 01 & 0 \\ 1 & 0 & 11 & 100 \end{pmatrix}$	162	1
115	$\begin{pmatrix} 1100 & 100 & 10 & 1 \\ 1 & 111 & 0 & 011 \end{pmatrix}$	159	1
116	$\begin{pmatrix} 1100 & 1 & 1 & 0 \\ 1 & 0101 & 001 & 11 \end{pmatrix}$	156	1
117	$\begin{pmatrix} 1110 & 1010 & 01 & 1 \\ 0 & 1 & 0 & 1101 \end{pmatrix}$	152	1
118	$\begin{pmatrix} 1111 & 100 & 0 & 0 \\ 10 & 0 & 0100 & 11 \end{pmatrix}$	152	1
119	$\begin{pmatrix} 1100 & 010 & 1 & 0 \\ 1 & 1 & 0 & 110 \end{pmatrix}$	150	1

120	$\begin{pmatrix} \texttt{1101} & \texttt{11} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{110} & \texttt{1100} & \texttt{111} \end{pmatrix}$	145	1
121	$\begin{pmatrix} \texttt{1100} & \texttt{1010} & \texttt{11} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{0} & \texttt{011} \end{pmatrix}$	144	2
122	$\begin{pmatrix} \texttt{1101} & \texttt{10} & \texttt{1} & \texttt{0} \\ \texttt{01} & \texttt{0110} & \texttt{1001} & \texttt{1} \end{pmatrix}$	144	1
123	$\begin{pmatrix} \texttt{1110} & \texttt{1011} & \texttt{11} & \texttt{0} \\ \texttt{0} & \texttt{0} & \texttt{1101} & \texttt{1} \end{pmatrix}$	144	1
124	$\begin{pmatrix} \texttt{1110} & \texttt{111} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{1011} & \texttt{00} & \texttt{1110} \end{pmatrix}$	142	1
125	$\begin{pmatrix} \texttt{1111} & \texttt{1110} & \texttt{1} & \texttt{0} \\ \texttt{110} & \texttt{11} & \texttt{10} & \texttt{1110} \end{pmatrix}$	140	1
126	$\begin{pmatrix} \texttt{1100} & \texttt{0101} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{1} & \texttt{00} & \texttt{01} \end{pmatrix}$	138	1
127	$\begin{pmatrix} \texttt{1110} & \texttt{000} & \texttt{11} & \texttt{1} \\ \texttt{11} & \texttt{1} & \texttt{0} & \texttt{1101} \end{pmatrix}$	138	1
128	$\begin{pmatrix} \texttt{1111} & \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{11} & \texttt{0} & \texttt{1110} \end{pmatrix}$	136	1
129	$\begin{pmatrix} \texttt{1111} & \texttt{011} & \texttt{10} & \texttt{1} \\ \texttt{1110} & \texttt{0} & \texttt{1} & \texttt{0101} \end{pmatrix}$	136	1
130	$\begin{pmatrix} \texttt{1100} & \texttt{101} & \texttt{010} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1010} \end{pmatrix}$	134	1
131	$\begin{pmatrix} \texttt{1110} & \texttt{100} & \texttt{01} & \texttt{1} \\ \texttt{0} & \texttt{1} & \texttt{0} & \texttt{110} \end{pmatrix}$	134	1
132	$\begin{pmatrix} \texttt{1110} & \texttt{10} & \texttt{1} & \texttt{0} \\ \texttt{10} & \texttt{11} & \texttt{0} & \texttt{0101} \end{pmatrix}$	133	1
133	$\begin{pmatrix} \texttt{1110} & \texttt{010} & \texttt{00} & \texttt{0} \\ \texttt{01} & \texttt{0} & \texttt{1110} & \texttt{0010} \end{pmatrix}$	133	1
134	$\begin{pmatrix} \texttt{1110} & \texttt{110} & \texttt{01} & \texttt{1} \\ \texttt{1} & \texttt{0} & \texttt{11} & \texttt{10} \end{pmatrix}$	133	1
135	$\begin{pmatrix} \texttt{1111} & \texttt{1100} & \texttt{1} & \texttt{0} \\ \texttt{0} & \texttt{0} & \texttt{100} & \texttt{1} \end{pmatrix}$	132	1
136	$\begin{pmatrix} \texttt{1110} & \texttt{010} & \texttt{1} & \texttt{0} \\ \texttt{0100} & \texttt{0} & \texttt{0111} & \texttt{10} \end{pmatrix}$	132	1
137	$\begin{pmatrix} \texttt{1110} & \texttt{101} & \texttt{010} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{1} & \texttt{1010} \end{pmatrix}$	132	1
138	$\begin{pmatrix} \texttt{1111} & \texttt{1100} & \texttt{11} & \texttt{0} \\ \texttt{00} & \texttt{1} & \texttt{0} & \texttt{110} \end{pmatrix}$	129	1
139	$\begin{pmatrix} \texttt{1110} & \texttt{100} & \texttt{1} & \texttt{0} \\ \texttt{01} & \texttt{1} & \texttt{00} & \texttt{1100} \end{pmatrix}$	129	1
140	$\begin{pmatrix} \texttt{1111} & \texttt{011} & \texttt{10} & \texttt{1} \\ \texttt{110} & \texttt{0} & \texttt{1} & \texttt{0101} \end{pmatrix}$	128	1

141	$\begin{pmatrix} 1110 & 1001 & 01 & 0 \\ 0 & 1 & 0 & 001 \end{pmatrix}$	128	1
142	$\begin{pmatrix} 1111 & 101 & 10 & 0 \\ 10 & 1 & 000 & 01 \end{pmatrix}$	126	1
143	$\begin{pmatrix} 1110 & 000 & 01 & 1 \\ 00 & 1 & 1 & 1110 \end{pmatrix}$	126	1
144	$\begin{pmatrix} 1101 & 11 & 00 & 0 \\ 11 & 0 & 1 & 0110 \end{pmatrix}$	126	1
145	$\begin{pmatrix} 1101 & 110 & 1 & 0 \\ 00 & 1 & 0011 & 01 \end{pmatrix}$	126	1
146	$\begin{pmatrix} 1100 & 111 & 1 & 0 \\ 1 & 0101 & 001 & 11 \end{pmatrix}$	125	1
147	$\begin{pmatrix} 1110 & 11 & 10 & 0 \\ 1 & 1010 & 0 & 011 \end{pmatrix}$	124	1
148	$\begin{pmatrix} 1100 & 11 & 1 & 0 \\ 1 & 0101 & 001 & 11 \end{pmatrix}$	121	1
149	$\begin{pmatrix} 1111 & 1101 & 10 & 1 \\ 101 & 1 & 011 & 0101 \end{pmatrix}$	120	1
150	$\begin{pmatrix} 1101 & 0011 & 001 & 1 \\ 00 & 11 & 0 & 1001 \end{pmatrix}$	120	1
151	$\begin{pmatrix} 1001 & 1001 & 01 & 0 \\ 10 & 1 & 0 & 001 \end{pmatrix}$	120	1
152	$\begin{pmatrix} 1110 & 1011 & 1 & 0 \\ 11 & 01 & 10 & 110 \end{pmatrix}$	120	1
153	$\begin{pmatrix} 1110 & 0110 & 01 & 1 \\ 00 & 0 & 1 & 011 \end{pmatrix}$	120	1
154	$\begin{pmatrix} 1110 & 01 & 1 & 0 \\ 00 & 1 & 1100 & 01 \end{pmatrix}$	120	1
155	$\begin{pmatrix} 1100 & 001 & 10 & 1 \\ 1 & 110 & 0 & 011 \end{pmatrix}$	120	1
156	$\begin{pmatrix} 1110 & 1010 & 1 & 0 \\ 0 & 1 & 00 & 10 \end{pmatrix}$	120	1
157	$\begin{pmatrix} 1111 & 0111 & 1 & 0 \\ 10 & 0 & 01 & 11 \end{pmatrix}$	119	1
158	$\begin{pmatrix} 1001 & 11 & 10 & 0 \\ 1 & 0 & 0 & 100 \end{pmatrix}$	118	1
159	$\begin{pmatrix} 1110 & 101 & 1 & 0 \\ 0 & 1 & 100 & 1 \end{pmatrix}$	118	1
160	$\begin{pmatrix} 1101 & 01 & 1 & 0 \\ 1 & 00 & 110 & 1011 \end{pmatrix}$	118	1
161	$\begin{pmatrix} 1110 & 1001 & 1 & 0 \\ 10 & 1 & 10 & 01 \end{pmatrix}$	118	1

162	$\begin{pmatrix} \texttt{1101} & \texttt{0011} & \texttt{000} & \texttt{0} \\ \texttt{0} & \texttt{00} & \texttt{1} & \texttt{0110} \end{pmatrix}$	117	1
163	$\begin{pmatrix} \texttt{1001} & \texttt{01} & \texttt{00} & \texttt{1} \\ \texttt{10} & \texttt{0} & \texttt{101} & \texttt{1001} \end{pmatrix}$	116	1
164	$\begin{pmatrix} \texttt{1101} & \texttt{0011} & \texttt{001} & \texttt{1} \\ \texttt{001} & \texttt{11} & \texttt{0} & \texttt{1001} \end{pmatrix}$	116	1
165	$\begin{pmatrix} \texttt{1100} & \texttt{1010} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{10} & \texttt{11} \end{pmatrix}$	116	1
166	$\begin{pmatrix} \texttt{1110} & \texttt{01} & \texttt{1} & \texttt{0} \\ \texttt{0} & \texttt{111} & \texttt{001} & \texttt{01} \end{pmatrix}$	115	2
167	$\begin{pmatrix} \texttt{1111} & \texttt{001} & \texttt{10} & \texttt{0} \\ \texttt{1010} & \texttt{1} & \texttt{0} & \texttt{001} \end{pmatrix}$	115	1
168	$\begin{pmatrix} \texttt{1101} & \texttt{011} & \texttt{10} & \texttt{0} \\ \texttt{1} & \texttt{000} & \texttt{0} & \texttt{110} \end{pmatrix}$	114	1
169	$\begin{pmatrix} \texttt{1111} & \texttt{10} & \texttt{01} & \texttt{0} \\ \texttt{0} & \texttt{0100} & \texttt{1} & \texttt{01} \end{pmatrix}$	114	1
170	$\begin{pmatrix} \texttt{1101} & \texttt{010} & \texttt{1} & \texttt{0} \\ \texttt{01} & \texttt{100} & \texttt{1011} & \texttt{1} \end{pmatrix}$	113	1
171	$\begin{pmatrix} \texttt{1101} & \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1001} & \texttt{1} & \texttt{0011} & \texttt{01} \end{pmatrix}$	112	1
172	$\begin{pmatrix} \texttt{1101} & \texttt{100} & \texttt{01} & \texttt{0} \\ \texttt{0101} & \texttt{1} & \texttt{0} & \texttt{100} \end{pmatrix}$	112	1
173	$\begin{pmatrix} \texttt{1110} & \texttt{1100} & \texttt{1} & \texttt{0} \\ \texttt{0} & \texttt{1} & \texttt{10} & \texttt{11} \end{pmatrix}$	112	1
174	$\begin{pmatrix} \texttt{1100} & \texttt{01} & \texttt{1} & \texttt{0} \\ \texttt{1010} & \texttt{1} & \texttt{0} & \texttt{0011} \end{pmatrix}$	112	1
175	$\begin{pmatrix} \texttt{1111} & \texttt{0100} & \texttt{1} & \texttt{0} \\ \texttt{110} & \texttt{0} & \texttt{100} & \texttt{1} \end{pmatrix}$	112	3
176	$\begin{pmatrix} \texttt{1110} & \texttt{1001} & \texttt{1} & \texttt{0} \\ \texttt{00} & \texttt{1} & \texttt{01} & \texttt{10} \end{pmatrix}$	111	1
177	$\begin{pmatrix} \texttt{1101} & \texttt{0101} & \texttt{1} & \texttt{0} \\ \texttt{01} & \texttt{000} & \texttt{1011} & \texttt{1} \end{pmatrix}$	110	1
178	$\begin{pmatrix} \texttt{1101} & \texttt{001} & \texttt{01} & \texttt{1} \\ \texttt{0} & \texttt{0101} & \texttt{1} & \texttt{1010} \end{pmatrix}$	110	1
179	$\begin{pmatrix} \texttt{1100} & \texttt{1001} & \texttt{010} & \texttt{0} \\ \texttt{1} & \texttt{00} & \texttt{1} & \texttt{100} \end{pmatrix}$	109	1
180	$\begin{pmatrix} \texttt{1110} & \texttt{0101} & \texttt{1} & \texttt{0} \\ \texttt{111} & \texttt{1} & \texttt{10} & \texttt{011} \end{pmatrix}$	109	1
181	$\begin{pmatrix} \texttt{1111} & \texttt{01} & \texttt{1} & \texttt{0} \\ \texttt{100} & \texttt{0} & \texttt{0001} & \texttt{1} \end{pmatrix}$	109	1
182	$\begin{pmatrix} \texttt{1100} & \texttt{01} & \texttt{1} & \texttt{0} \\ \texttt{11} & \texttt{001} & \texttt{0} & \texttt{10} \end{pmatrix}$	109	1

183	$\begin{pmatrix} 1101 & 11 & 1 & 0 \\ 1 & 0101 & 0 & 110 \end{pmatrix}$	108	1
184	$\begin{pmatrix} 1100 & 1010 & 00 & 1 \\ 1 & 001 & 11 & 0101 \end{pmatrix}$	108	1
185	$\begin{pmatrix} 1101 & 110 & 1 & 0 \\ 1 & 010 & 110 & 1011 \end{pmatrix}$	107	1
186	$\begin{pmatrix} 1101 & 1010 & 1 & 0 \\ 01 & 1 & 00 & 10 \end{pmatrix}$	107	1
187	$\begin{pmatrix} 1110 & 1101 & 1 & 0 \\ 0100 & 01 & 1011 & 1 \end{pmatrix}$	107	1
188	$\begin{pmatrix} 1110 & 1010 & 1 & 0 \\ 0 & 101 & 001 & 1 \end{pmatrix}$	105	1
189	$\begin{pmatrix} 1110 & 1010 & 1 & 0 \\ 0 & 10 & 01 & 1 \end{pmatrix}$	105	1
190	$\begin{pmatrix} 1110 & 001 & 01 & 0 \\ 1 & 111 & 0 & 100 \end{pmatrix}$	104	1
191	$\begin{pmatrix} 1111 & 1101 & 10 & 1 \\ 0 & 1 & 011 & 0101 \end{pmatrix}$	104	1
192	$\begin{pmatrix} 1110 & 001 & 1 & 0 \\ 10 & 1 & 00 & 0011 \end{pmatrix}$	104	1
193	$\begin{pmatrix} 1110 & 11 & 10 & 01 \\ 1 & 1011 & 011 & 1 \end{pmatrix}$	103	1
194	$\begin{pmatrix} 1111 & 1011 & 01 & 1 \\ 10 & 1 & 110 & 1010 \end{pmatrix}$	103	1
195	$\begin{pmatrix} 1101 & 0101 & 1 & 0 \\ 1 & 01 & 00 & 10 \end{pmatrix}$	103	1
196	$\begin{pmatrix} 1111 & 01 & 1 & 0 \\ 100 & 0 & 0011 & 1 \end{pmatrix}$	102	1
197	$\begin{pmatrix} 1110 & 1011 & 01 & 1 \\ 010 & 1 & 110 & 1010 \end{pmatrix}$	102	1
198	$\begin{pmatrix} 1110 & 11 & 1 & 0 \\ 0 & 0010 & 110 & 1 \end{pmatrix}$	100	1
199	$\begin{pmatrix} 1111 & 01 & 1 & 0 \\ 1010 & 0 & 0001 & 1 \end{pmatrix}$	100	1
200	$\begin{pmatrix} 1101 & 00 & 1 & 0 \\ 110 & 1 & 0 & 100 \end{pmatrix}$	100	1

Note: *all instances in the appendix are in their normalized forms.*

Appendix B: Results on 33 instances in $PCP[3,3]$ not solved by the solver

Exclusion method can prove 13 instances unsolvable

1	$\begin{pmatrix} 111 & 100 & 0 \\ 11 & 011 & 100 \end{pmatrix}$	2	$\begin{pmatrix} 111 & 10 & 0 \\ 11 & 111 & 000 \end{pmatrix}$
3	$\begin{pmatrix} 111 & 001 & 1 \\ 10 & 0 & 111 \end{pmatrix}$	4	$\begin{pmatrix} 111 & 000 & 1 \\ 10 & 0 & 111 \end{pmatrix}$
5	$\begin{pmatrix} 111 & 000 & 1 \\ 10 & 0 & 11 \end{pmatrix}$	6	$\begin{pmatrix} 111 & 01 & 1 \\ 10 & 1 & 110 \end{pmatrix}$
7	$\begin{pmatrix} 111 & 100 & 0 \\ 1 & 11 & 100 \end{pmatrix}$	8	$\begin{pmatrix} 111 & 100 & 0 \\ 1 & 11 & 00 \end{pmatrix}$
9	$\begin{pmatrix} 111 & 10 & 1 \\ 1 & 1 & 011 \end{pmatrix}$	10	$\begin{pmatrix} 111 & 1 & 0 \\ 1 & 10 & 1 \end{pmatrix}$
11	$\begin{pmatrix} 111 & 10 & 1 \\ 0 & 1 & 011 \end{pmatrix}$	12	$\begin{pmatrix} 110 & 00 & 1 \\ 010 & 0 & 011 \end{pmatrix}$
13	$\begin{pmatrix} 110 & 00 & 1 \\ 01 & 0 & 110 \end{pmatrix}$		

Group method can prove 12 instances unsolvable

14	$\begin{pmatrix} 110 & 10 & 0 \\ 1 & 0 & 010 \end{pmatrix}$	15	$\begin{pmatrix} 111 & 01 & 1 \\ 10 & 1 & 101 \end{pmatrix}$
16	$\begin{pmatrix} 111 & 10 & 1 \\ 1 & 1 & 110 \end{pmatrix}$	17	$\begin{pmatrix} 111 & 10 & 1 \\ 1 & 1 & 101 \end{pmatrix}$
18	$\begin{pmatrix} 111 & 00 & 1 \\ 1 & 1 & 100 \end{pmatrix}$	19	$\begin{pmatrix} 111 & 10 & 1 \\ 0 & 1 & 101 \end{pmatrix}$
20	$\begin{pmatrix} 110 & 110 & 1 \\ 11 & 0 & 101 \end{pmatrix}$	21	$\begin{pmatrix} 110 & 011 & 1 \\ 11 & 0 & 101 \end{pmatrix}$
22	$\begin{pmatrix} 110 & 11 & 1 \\ 11 & 0 & 101 \end{pmatrix}$	23	$\begin{pmatrix} 110 & 01 & 0 \\ 1 & 0 & 010 \end{pmatrix}$
24	$\begin{pmatrix} 110 & 00 & 1 \\ 0 & 1 & 100 \end{pmatrix}$	25	$\begin{pmatrix} 110 & 1 & 0 \\ 0 & 101 & 11 \end{pmatrix}$

Mask method can prove 1 instance unsolvable

26	$\begin{pmatrix} \texttt{111} & \texttt{0} & \texttt{0} \\ \texttt{1} & \texttt{100} & \texttt{11} \end{pmatrix}$

Pattern method can prove 6 instances unsolvable

27	$\begin{pmatrix} \texttt{111} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{110} & \texttt{111} \end{pmatrix}$	28	$\begin{pmatrix} \texttt{111} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{110} & \texttt{1} \end{pmatrix}$
29	$\begin{pmatrix} \texttt{110} & \texttt{10} & \texttt{1} \\ \texttt{1} & \texttt{11} & \texttt{01} \end{pmatrix}$	30	$\begin{pmatrix} \texttt{110} & \texttt{10} & \texttt{0} \\ \texttt{1} & \texttt{0} & \texttt{001} \end{pmatrix}$
31	$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{01} & \texttt{011} \end{pmatrix}$	32	$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{01} & \texttt{010} \end{pmatrix}$

1 instance is still unsolved

33	$\begin{pmatrix} \texttt{110} & \texttt{1} & \texttt{0} \\ \texttt{1} & \texttt{01} & \texttt{110} \end{pmatrix}$

Note: *all instances except the last one are solved by hand, and those methods used manually either are too complicated to implement, or have been implemented but could not deal with those particular cases.*