國立交通大學

資訊科學與工程研究所

博 士 論 文

利用志願型運算
解決電腦遊戲問題

On Solving Computer Game Problems
Based on Volunteer Computing

研 究 生：林宏軒

指導教授：吳毅成　教授

中 華 民 國 一零二 年 七 月

利用志願型運算解決電腦遊戲問題
On Solving Computer Game Problems Based on Volunteer Computing

研 究 生：林宏軒　　Student：Hung-Hsuan Lin

指導教授：吳毅成　　Advisor：I-Chen Wu

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

博 士 論 文

A Dissertation

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

July 2013

Hsinchu, Taiwan, Republic of China

中華民國 一零二 年 七 月

利用志願型運算解決電腦遊戲問題

研究生：林宏軒　　指導教授：吳毅成 博士

國立交通大學資訊科學與工程研究所博士班

摘要

電腦遊戲是人工智慧(Artificial Intelligence)中一項非常重要的研究領域，其中有些問題需要非常大量的運算才能解出，而志願型運算(Volunteer Computing)非常適合用來解決這些問題。由於電腦遊戲的特性，大部份問題需要即時地產生及取消工作，這在傳統的志願型運算上會非常沒有效率，因為傳統的志運型運算為非連線模式，無法即時更改工作。本論文提出工作層級運算模式(Job-level Computing Model)，並以此模式為基礎發展一套新的且具有一般化之工作層級志願型運算系統，此系統使用的是連線模式，以避免傳統志願型運算解決電腦遊戲效率不佳的問題。本論文利用傳統型志運型運算與工作層級志願型運算分別解決兩個電腦遊戲問題：數獨最小提示數問題(Minimum Sudoku Problem)與六子棋開局問題(Connect6 Opening Problem)。

在解決數獨最小提示數問題中，本論文亦改善了 2006 年的數獨 Checker 程式，加快了約 128 倍效率，使用此改善程式可將原本需要約 30 萬年單核時間的數獨最小提示數問題減少成約 2417 年可解。而在解決六子棋開局問題中，成功地將證明數搜尋演算法應用於工作層級志運型運算系統中，並提出了延遲兄弟節點產生法及假設證明數相同之方法來展開搜尋樹節點，成功地解出許多六子棋盤面的勝敗，其中包含多個開局，例如米老鼠開局，此開局在過去是很受歡迎的開局之一。根據實驗數據顯示，在 16 核的環境下其速率可提升 8.58 倍。

On Solving Computer Game Problems Based on Volunteer Computing

Student：Hung-Hsuan Lin Advisor：Dr. I-Chen Wu

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Computer game is an important field of research in artificial intelligence, while some computer game problems take huge amount of computation time to solve, which is suitable to use volunteer computing to solve. However, due to the property of computer games, most computer game problems generate or abort jobs dynamically when solving, which makes computer game problems cannot be solved efficiently on traditional volunteer computing, which uses connectionless model and cannot support the function. This thesis proposes job-level computation model, and based on this model to propose a new generic job-level volunteer computing system, which is a connection model, to solve compute game problems efficiently. This thesis uses traditional volunteer computing and job-level volunteer computing to solve the minimum Sudoku problem and Connect6 game openings, respectively.

For solving the minimum Sudoku problem, this thesis speedups the Sudoku program Checker written by McGuire in 2006 by a factor of about 128, and reduce the computation time for solving the minimum Sudoku problem from about 300 thousand year on a one-core

machine to about 2417 years. For solving Connect6 openings, this thesis successfully incorporates proof number search into the job-level volunteering computing system, and proposes postponed sibling generation and virtual-equivalence methods to generate nodes in search trees. Based on this system, many new Connect6 game positions are solved efficiently, including several Connect6 openings, especially the Mickey-Mouse Opening, which was one of the popular openings. The experiments showed 8.58 speedups in a system with 16 cores.

致謝

經過了孜孜矻矻的多年，終於在 2013 年提出博士學位口試。

非常感謝指導老師吳毅成教授多年來的照顧，老師除了在課業及學術方面指導我之外，更常教導我人生方面的道理，對於我在課業以外的困難更是不遺餘力地幫忙。老師更是常利用平常假日的休閒時間指導我們，讓我們每當有問題時都能及時地與老師討論。

感謝口試委員王才沛教授、吳毅成教授、林秉宏博士、林順喜教授、徐慰中教授、徐讚昇教授、陳穎平教授、許舜欽教授和顏士淨教授（以上按姓氏筆劃排列），提出論文與報告的不足之處並提出改進方法，讓我的論文與表達能更加進步。

感謝待了十年的交通大學（大學四年、研究所一年完逕博、博士五年），讓我能順利完成學士與博士學位，也感謝在碩博六年的期間的實驗室同學兼好友們，包括林秉宏學長、黃裕龍學長、典餘學長、RC 學長、哲毅學長、汶傑學長、冠翬、柏甫、益嘉、聰哥、宜智、平平、賴打、小雄、gy、家茵、柏廷、郁雯、修全、tataya、正宏、草莓、蔡心、mark、派大星、拉拉、kk、小吉、嘉嘉、青蛙、左左、piggy、東東、小閔、TF、Miso、黑月、博玄、包子、庭築、Ting、阿賢與小康。

最後，我要感謝我最愛的父母，沒有他們就不會有我，也非常謝謝他們對於我讀博士班的支持與鼓勵,在我遇到任何困難的時候我都可以回到溫暖的家找我父母閒聊,在我高興的時候也能與他們分享我的喜悅，也感謝所有親戚對我的關心與支持，謹以此論文獻給我最摯愛的家人與所有的親朋好友。

林宏軒

2013 年 7 月 22 日

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Computer game is an important field of research in artificial intelligence, while the goal of artificial intelligence is to make computers to be more intelligent and useful. Schaeffer and Herik [45] said “chess is to AI as the fruit fly is to genetics”, which shows the importance of chess, a kind of computer game, for artificial intelligence. A milestone in the field of computer game was that Deep Blue beat the chess championship, Kasparov [10].

The research topics for computer games include solving computer games and computer game problems. However, many computer games or computer game problems difficult to solve have high complexity. The solution to solve these with low cost is to use volunteer computing. This thesis uses BOINC [5], a kind of traditional volunteer computing, to solve the minimum Sudoku problem.

However, many computer game problems cannot be solved efficiently on the traditional volunteer computing because the problems may generate or abort the jobs dynamically, which will be described in Section 1.2. Thus, this thesis also proposes a *job-level computation model*. Based on this model, we propose a new *generic job-level volunteer computing system* [63] to solve these kind of computer game problems efficiently. This thesis uses Connect6 openings to demonstrate the job-level volunteer computing.

This chapter is organized as follows. Section 1.1 introduces computer games and computer game problems. Section 1.2 introduces traditional volunteer computing. Section 1.3 describes the motivation and goal. Section 1.4 describes the organization of this thesis.

1.1 Computer Games and Computer Game Problems

Computer games can be categorized as single player games, two-player games, and multi-player games, according to the number of players. For example, the game Sudoku, Connect6, and Mahjong are popular single player game, two-player game, and multi-player game, respectively. Also, computer games can be categorized as perfect information games, imperfect-information games, and stochastic games, according to the information obtained by each player. Each player can get all the information in a perfect information game, but some information is not available in an imperfect-information game. For example, the games Sudoku and Connect6 are perfect information games, and the game Mahjong is an imperfect-information game. Stochastic games include the element of possibility such as dice rolls.

For computer games, there are two kinds of complexities to decide how hard to prove the results of the games, state-space complexity and game-tree complexity [21]. State-space complexity presents the total number of positions or states of a game. Game-tree complexity presents the number of positions needed to be evaluated in a minimax search manner [51] to determine the value of initial state of a game.

A computer game with low state-space complexity can be solved by brute-force methods, namely, by evaluating the results of all the positions in the game. And, a computer game with low game-tree complexity can be solved by knowledge-based method, namely, by heuristic searching.

Games	State-Space Complexity	Game-Tree Complexity
Go	10^{172}	10^{360}
Shogi	10^{71}	10^{226}
Connect6	10^{172}	10^{140}-10^{188}
Chinese chess	10^{48}	10^{150}
Chess	10^{46}	10^{123}
Hex	10^{57}	10^{98}
Go-Moku	10^{105}	10^{70}
Renju	10^{105}	10^{70}
Othello	10^{28}	10^{58}
Nine Men's Morris	10^{10}	10^{50}
Qubic	10^{30}	10^{34}
Checkers	10^{21}	10^{31}
Connect-Four	10^{14}	10^{21}

Table 1: The complexity of computer games

As shown in Table 1, Go [40], Shogi [25], and Chinese chess [69] have the highest, second highest, and third highest complexity, 10^{360}, 10^{226}, and 10^{150}, respectively.

Many researchers have been trying to solve computer games. Some computer games have been weakly solved since the games have low state-space complexity and game-tree complexity, such as Connect-Four [1] and Qubic [37]. Some computer games have mainly been weakly solved by brute-force methods because they have low state-space complexity, such as Nine Men's Morris [19] and Nine-Layer Triangular Nim [49]. And, some computer games have mainly been weakly solved by knowledge-based methods, such as Go-Moku [3], Renju [57], checkers [46], and k-in-a-row games.

Unsolved and hard computer games are tournament items for computer game tournaments. For example, many computer game tournaments were held, such as TCGA Tournaments [54], TAAI Tournaments [28][70], ICGA Tournaments [24], etc, and many computer games competition were held in these tournaments, such as chess, Chinese chess, Connect6, Go, Hex, Shogi, etc.

Besides the above, there are some computer game problems, such as building the opening databases, solving the endgame positions, or some other interesting problems. Here we introduce two computer game problems which this thesis tends to solve, the minimum Sudoku problems and Connect6 game openings.

1.1.1 Minimum Sudoku Problem

Sudoku is a popular puzzle game invented by Harold Garns (cf. [33]) in 1979 and has been popular and printed in daily newspapers, magazines, and websites since 2005. A *Sudoku puzzle* is played on a 9×9 grid which is divided into nine boxes each with 3×3 cells. In a puzzle, some digits between 1 and 9 are initially given on the grid as the clues.

The aim of a Sudoku puzzle is to fill the 9×9 grid up from the initial grid of Sudoku puzzle into a *valid Sudoku complete grid*, or simply called a *complete grid*, where each column, each row and each box contains distinct digits 1-9.

								1
					2			
		3		4			5	
								6
	1					4		2
7			3	5				
			6					
		8					4	
5			1					

(a)

4	9	7	5	6	3	8	2	1
8	5	6	7	1	2	3	9	4
1	2	3	8	4	9	6	5	7
3	8	9	4	2	1	5	7	6
6	1	5	9	8	7	4	3	2
7	4	2	3	5	6	9	1	8
2	3	1	6	9	4	7	8	5
9	6	8	2	7	5	1	4	3
5	7	4	1	3	8	2	6	9

(b)

Figure 1: (a) A 17-clue puzzle and (b) its complete grid.

A Sudoku puzzle is called a *valid Sudoku puzzle*, or simply a *valid puzzle*, if it is solved with a unique complete grid. A valid puzzle with n clues initially is called an *n-clue*

puzzle. Figure 1 (a) illustrates a 17-clue puzzle, while Figure 1 (b) shows the complete grid of this puzzle. And, the minimum Sudoku problem is investigating the minimum of clues for Sudoku puzzles. This thesis uses traditional volunteer computing, BOINC [5], to help solve this problem by exhaustively checking all 16-clue puzzles.

1.1.2 Connect6 Game Openings

Connect6 [61][62] is a kind of six-in-a-row game that was introduced by Wu *et al*. Two players, named *Black* and *White*, alternately play one move by placing two black and white stones respectively on empty intersections of a Go board (a 19×19 board) in each turn. Black plays first and places one stone initially. The winner is the first to get six consecutive stones of his own horizontally, vertically or diagonally.

One issue for Connect6 is that the game lacks openings for players, since the game is still young when compared with other games such as chess, Chinese chess and Go. Hence, it is important for the Connect6 player community to investigate more openings quickly. This problem is not suitable to solve on traditional volunteer computing, and this thesis proposes job-level volunteer computing to help solve.

1.2 Traditional Volunteer Computing

As described above, some computer game problems are hard to solve and need huge amount of computing resources. Volunteer computing can be used to help solve these problems with low cost. Volunteer computing uses the spare time of computers without influencing the users' usage. For most computers, CPU idle percentage is very high. If the idle CPU can support, then many problems can be solved. For example, BOINC [5], Berkeley Open Infrastructure for Network Computing, is a popular middleware for

volunteer computing. Many projects are running on it, such as SETI@home [6][48], Einstein@home [14], and PrimeGrid [39], etc.

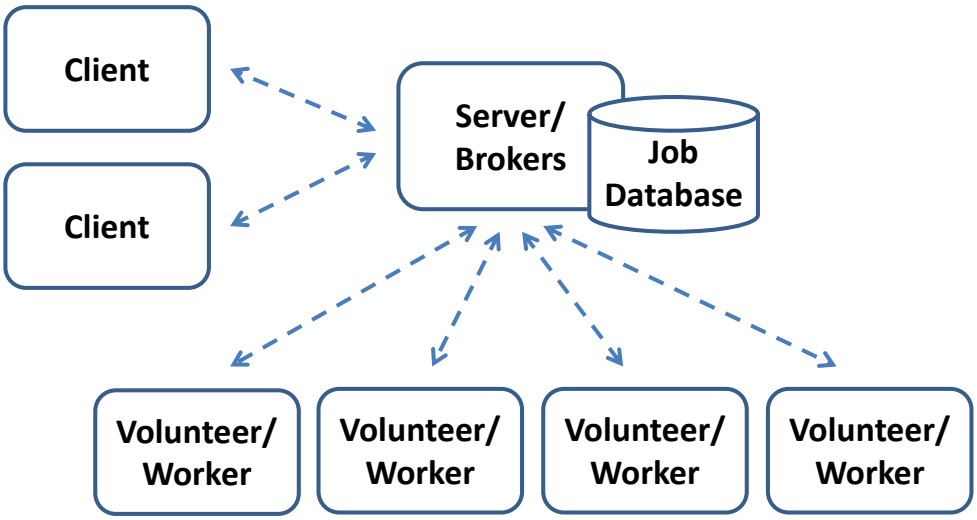


Figure 2: The roles of volunteer computing

Figure 2 shows the roles of volunteer computing, a server with a database, workers, and clients. A computer game problem which needs to be solved is divided into many small *jobs*, and uploaded by a *client* and stored in a *server* and a *database*. A *volunteer* donates spare time and helps run the jobs, which performs as a *worker*. These workers download jobs from the server and send the results back to the server after the jobs are completed, and the server will verify the results.

Traditional volunteer computing uses connectionless model. The workers connect to the server when they download the jobs, which can be run offline. The workers will connect to the server again when they want to upload the results and download more jobs. For example, as a worker of BOINC, the worker can download many jobs at a time which may take tens of days to run, and upload the results after completed.

The connectionless model cannot solve most computer game problems efficiently because the computer game problems are usually *highly dynamic* as follows. When solving the computer game problems, the result of a job may generate new jobs or abort other jobs

and makes the jobs dynamically changes. For example, for the game Connect6, the player to move needs to consider every possible move and verify how good the moves are, and then choose the best one to move. If the player finds that one of the possible moves can get a win, then other moves can be aborted immediately. However, the traditional volunteer computing uses connectionless model and the server cannot ask the workers to abort the jobs, so the workers would waste much time on running useless jobs when solving computer game problems.

1.3 Motivation and Goal

This thesis uses volunteer computing to solve two computer game problems, the minimum Sudoku problem and the Connect6 game openings. The minimum Sudoku problem can be divided into many small independent jobs which can be run in parallel, so the problem can be solved on the traditional volunteer computing, and this thesis uses BOINC [5] to help solve.

For solving the minimum Sudoku problem, this thesis modifies the program written by McGuire in 2006 [34], and speedups the program by a factor of about 128, which reduces the computation time of solving the minimum Sudoku problem from about 300,000 one-core years to about 2417 one-core years. After improving the program, we started to solve the minimum Sudoku problem using BOINC framework [5]. At the end of July 2013, our project has completed the checking of more than 93% primitive grids, and no 16-clue grids have been found yet. We expect to complete the result soon.

Independently, McGuire also improved his own program Checker. According to his article [35], their modified Checker was about twice faster than mine and he took one whole year to run the jobs through January 2011 to December 2011. They claimed the result that

no 16-clue puzzles exist in January 2012, while our BOINC was still running at that time. Our project still continues on BOINC. At the end of July 2013, our project has completed the checking of more than 93% primitive grids, and no 16-clue grids have been found yet. We expect to complete the result soon.

Another problem this thesis uses volunteer computing to solve is the Connect6 game openings. However, as described above, the problem of solving Connect6 game openings is highly dynamic and cannot be solved efficiently on the traditional volunteer computing. Thus, this thesis proposes a new generic job-level volunteer computing system, which is connection model, to help solve the problem efficiently. This thesis introduces a new approach, named *generic job-level search*, where a search tree is maintained by the client. Search tree nodes are evaluated or expanded/generated by leveraging the game-playing programs which are already well-written and encapsulated as jobs, usually heavy-weight jobs requiring tens of seconds or more. The generic job-level search approach also has the following advantages:

- Develops jobs (usually heavy-weight jobs or well-written programs) and the job-level search independently, except for a few extra processes required to support job-level search from these jobs. As described in this thesis, these processes are relatively low-level.
- Dispatches jobs to remote processors in parallel. Job-level search is suited to parallel processing since the jobs are heavy-weight.
- Maintains the job-level search tree inside the memory of clients without much problem. Since well-written game-playing programs normally support accurate domain-specific knowledge to a certain extent, the search trees require fewer nodes to solve the game positions (when compared with a best-first search such as proof number search [4] using one process only). In our experiments for Connect6, the search tree usually

contains no more than one million nodes, which fits well into (client) process memory. For example, assume that it takes one minute to run a job (to generate one node). A parallel system with 60 processors takes about 11 days to build a tree of up to one million nodes. Should we need to run much more than one million nodes, we can split the job-level search tree into several nodes, each per client.

- Easily monitors the search tree. Since the maintenance cost for the job-level search tree is low, the client that maintains the job-level search tree can support more GUI utilities to let users easily monitor the running of the whole job-level search tree in real time. In fact, our client is embedded into a game record editor environment. An extra benefit of this is to allow users or experts to look into the search tree during the running time, and to help choose the best move to search in the case that the program does not find the best move to win (see [59][65]).

For node generation of generic job-level search, we need to select nodes and then expand them. For node expansion, this thesis proposes a method, named *postponed sibling generation method*, to help expand the selected nodes.

This thesis also proposes a new policy, named *virtual-equivalence*. In this policy, it is assumed that the value of a game position is close to (or equal to) that of the position for the best move, and that the value for the nth best move is close to (or equal to) that for the $(n+1)$ best move. This thesis also proposes some variants of virtual-equivalence. The experiments showed that one of the virtual-equivalence variants performed the best and improved the virtual-win/virtual-loss policies by a factor of about 1.86.

Using proof number search to maintain the search tree with the job NCTU6, on desktop grids (a kind of volunteer computing system[1] [5][16][48][59]), this thesis solved

[1] A desktop grid is developed for volunteer computing which aimed to harvest idle computing resources for speeding up high throughput. It is a kind of distributed computing.

several Connect6 positions including several difficult 3-move openings, as shown in Figure 32 (in Section 4.3). For some of these openings, none of the human Connect6 experts had been able to find the winning strategies. These solved openings include the popular *Mickey-Mouse Opening*[2], [55], as shown in Figure 32 (i).

1.4 Organization

The organization of this thesis is as follows. Chapter 2 uses traditional volunteer computing, BOINC [5] volunteer computing (BVC), to solve the minimum Sudoku problem. Chapter 3 defines job-level volunteer computing (JLVC). Chapter 4 uses JLVC to solve Connect6 game openings. Chapter 5 concludes this thesis.

[2] The opening was so called by Connect6 players since White 2 and Black 1 together look like the face of Mickey Mouse to them.

Chapter 2 Solving Games Using BVC

The first problem this thesis tends to solve is: what is the minimum number of clues for a valid Sudoku puzzle? This is the so-called *minimum-clue Sudoku problem*, or the *minimum Sudoku problem*.

Since this problem can be divided into many small jobs, which can be run independently, this problem is suitable to be solved on the traditional volunteer computing. This thesis uses BOINC [5] volunteer computing (BVC) to help solve this problem.

This chapter is organized as follows. Section 2.1 introduces this problem. Section 2.2 describes traditional approaches for the minimum Sudoku problem including the program Checker [34]. Section 2.3 describes our new approach. Section 2.4 does experiments for analyzing the performance improvements by our approach. Section 2.5 makes concluding remarks.

2.1 Introduction

The minimum Sudoku problem is asking for the smallest n for valid n-clue puzzles. Currently, many 17-clue puzzles have been found. One of these puzzles is shown in Figure 1 (a). The approach to solving the problem is briefly described as follows.

For a given grid, we can easily generate more *isomorphic grids* [42] by the following operations.

- Relabel digits. For example, relabel all 2s and 5s to 5s and 2s, respectively.
- Permute single rows (columns) within the same box row (column), or permute box rows (columns). A box row (column) indicates the three boxes in the same rows

(columns). For example, permute the first and second rows; permute the first box column (the leftmost three boxes) and the second box column (the middle three boxes).

- Rotate and mirror boards.

An important assertion is: if a puzzle P with initial grid G is valid, then another puzzle P' with initial grid G' which is isomorphic to G is valid, too. The puzzle P' is said to be *isomorphic* to P. For simplicity of discussion, let $iso(G)$ denote *the group of isomorphic grids* generated from G. Note that G is also included in $iso(G)$. From above, if a puzzle with initial grid G is valid, all the puzzles with initial grids in $iso(G)$ are valid too.

The numbers of isomorphic grids in groups are usually enormous. For example, a complete grid may have up to $2*9!*6^8$ = 1,218,998,108,160 isomorphic complete grids. Similarly, a valid puzzle such as the one in Figure 1 (a) normally has enormous isomorphic valid puzzles, too. Thus, it becomes less interesting to find valid puzzles which are isomorphic to some found valid puzzles. Currently, Royle [41] collected 49151 17-clue puzzles, each of which is not isomorphic to any others. These puzzles are called *essentially different Sudoku puzzles*.

The total number of complete grids is 6,670,903,752,021,072,936,960 [17]. In fact, many of them are isomorphic. The total number of distinct isomorphic groups is 5,472,730,538 [42]. Fowler [18] also generated 5,472,730,538 complete grids, one for each isomorphic group. These complete grids are *essentially different Sudoku grids*, and are called *primitive grids* in this thesis. Two features of primitive grids are as follows.

1. Each complete grid is isomorphic to one of these primitive grids.
2. Each primitive grid is not isomorphic to any other primitive grids.

An important approach to solving the minimum Sudoku problem is investigating exhaustively all these primitive grids only to check whether 16-clue puzzles exist in these

primitive grids or not. The approach must be able to find one 16-clue puzzle, if there exists a 16-clue puzzle, for the following reason. Assume that some 16-clue puzzle P can be solved with a unique complete grid G. From the first feature, there exists one and only one primitive grid G', isomorphic to G. This implies that there exists a 16-clue puzzle P' solved with the complete grid G' uniquely. Namely, we can translate the initial grid of puzzle P into that of P' by using the same transformation from G to G'. Thus, the puzzle P' should be found when the primitive grid G' is investigated.

Using this approach, McGuire [34] wrote a program, named *Checker*, in 2006 to help solve this problem. Given a number n and a complete grid, the program checks whether there exist n-clue puzzles which can be solved with the complete grid, and outputs the found n-clue puzzles, if any. Hence, we can solve the minimum Sudoku problem by using Checker to search 16-clue puzzles from all the 5,472,730,538 primitive grids.

This approach has two advantages. First, the program does not need to investigate isomorphic complete grids redundantly. Second, these primitive grids can be checked independently. That is, they can be done on top of traditional volunteer computing, such as BOINC volunteer computing [5].

Unfortunately, the total computation time for solving the problem is still too high. According to our experiment (see Section 2.4), the program Checker written in 2006 actually required on the average about 1792.31 seconds to check a primitive grid on one core of a computer equipped with the CPU, Intel(R) Xeon(R) E5520 @ 2.27GHz. Thus, for 5,472,730,538 primitive grids, it would take about 311,000 years. The total time was unfortunately too long.

In this thesis, we propose a new approach to improve the program Checker. We design a new algorithm, named *DMUS algorithm* [29], incorporate it into the program Checker, and make some more tunings on the program. According to our experiment (see Section

2.4), the modified program could check one on the average on one core of a computer in 13.93 seconds. Thus, it would only take about 2417 years to check all 5,472,730,538 primitive grids.

Since the 5,472,730,537 primitive grids are independent, we can have many independent jobs to run on traditional volunteer computing, such as BOINC. We started our Sudoku project to solve the minimum Sudoku problem on BOINC on October 2010. At the end of July 2013, the Sudoku project has completed the checking of more than 93% primitive grids, and no 16-clue grids have been found yet. We expect to complete the result soon.

McGuire, the author of the program Checker, with his team also improved their own program independently. According to their article [35], their modified Checker was about twice faster than mine and he took 7.1 million core hours on the Stokes machine, an SGI Altix ICE 8200EX cluster with 320 compute nodes, to solve the minimum Sudoku problem. They started running the jobs in January 2011, finished in December 2011, and claimed the result that no 16-clue puzzles exist in January 2012. In contrast, our first paper was submitted to TAAI in June 2010 [30] and we started running jobs in October 2010. Although the minimum Sudoku problem was solved by McGuire in January 2012, we still continued our project because our independent program can also help verify the result.

2.2 Traditional Approach

Solving the minimum Sudoku problem is a very difficult job as described above. Most researchers tend to seek 16-clue or 17-clue puzzles at random, instead of searching all cases exhaustively. In case that there exists some 16-clue puzzle, the 16-clue puzzle implies the existence of another 65 17-clue puzzles by simply filling one more cell on the 16-clue

puzzle. On the other hand, if one of the 65 17-clue puzzles is found, then we can easily find the 16-clue puzzle by removing one clue and checking whether or not it is still valid. Most researchers seek 17-clue puzzles in this approach.

In the rest of this section, Subsection 2.2.1 describes the traditional approaches of finding more 17-clue puzzles, while Subsection 2.2.2 describes the traditional approaches of checking whether or not 16-clue puzzles exist.

2.2.1 Finding 17-clue Puzzles

One of the most popular algorithms of finding new 17-clue puzzles is called *gene restructuring* in [23][32]. This algorithm starts with an n-clue puzzle and then performs the following operation. First, remove p existing clues on the puzzle, and then add q clues back to the puzzle. For simplicity, let $-p+q$ indicate such an operation.

We introduce two common methods from the Sudoku Forum [53] to obtain more 17-clue puzzles from the existing valid puzzles as follows:

1. Do $-k+k$ operations from 17-clue puzzles.
2. Do the following from n-clue puzzles, where $18 \le n \le 23$.
   a. Repeat $-2+1$ operations until 18-clue puzzles are obtained.
   b. Then, repeat $-1+1$ operations many times to obtain more 18-clue puzzles.
   c. Finally, do one $-2+1$ operation to obtain more 17-clue puzzles.

The first method starts with 17-clue puzzles and does a $-k+k$ operation to obtain new 17-clue puzzles. Running with $k \le 2$ is very fast, while it takes much longer time with $k \ge 4$.

The second method starts with n-clue puzzles where $18 \le n \le 23$, and repeats $-2+1$ operations until it gets 18-clue puzzles. Also it does an extra $-1+1$ operation many times on the 18-clue puzzles to obtain more 18-clue puzzles. Finally, a $-2+1$

operation is used on these 18-clue puzzles to get 17-clue puzzles.

Both methods above are very useful to find 17-clue puzzles. Many of the 49151 17-clue puzzles were obtained in this way. However, since no 16-clue puzzles were found, they failed to conclude whether or not any 16-clue puzzles exist.

2.2.2 Checking All 16-clue Puzzles

Another approach to solving the minimum Sudoku problem is to exhaustively search for 16-clue puzzles. This can be done by the program Checker [34], written by McGuire in 2006. This program was motivated when Royle (cf. [34]) found a special complete grid shown in Figure 3, where we can find exactly 29 17-clue puzzles. That is, these 29 17-clue puzzles can be solved uniquely with this complete grid. Since a 16-clue puzzle could produce 65 17-clue puzzles as describe above, it is more likely that this complete grid contains a 16-clue puzzle, though no other 16-clue puzzles have been found from this puzzle by Checker.

6	3	9	2	4	1	7	8	5
2	8	4	7	6	5	1	9	3
5	1	7	9	8	3	6	2	4
1	2	3	8	5	7	9	4	6
7	9	6	4	3	2	8	5	1
4	5	8	6	1	9	2	3	7
3	4	2	1	7	8	5	6	9
8	6	1	5	9	4	3	7	2
9	7	5	3	2	6	4	1	8

Figure 3: The complete grid with 29 17-clue puzzles.

Given a complete grid and a number n, the program Checker runs in the following two phases. Phase 1 is to search the grid for unavoidable sets, defined in Subsection 2.2.2.1. Phase 2, described in Subsection 2.2.2.2, uses these unavoidable sets to search n-clue puzzles.

2.2.2.1. Phase 1: Unavoidable Sets and Finding Unavoidable Sets

In a complete grid, an *unavoidable set* is a set of cells on which the digits can be permuted to form another distinct complete grid. In other words, if we remove all the digits in an unavoidable set from the complete grid and let the remaining digits form a new puzzle, then the new puzzle can be solved with more than one complete grid. For example, for a complete grid including the bolded digits shown in Figure 4, the four bolded digits, two 1s and two 2s, in the upper left corner form an unavoidable set. The complete grid is transformed to another complete grid by exchanging the 1s and 2s in this unavoidable set. In fact, all bolded 4s and 5s form one unavoidable set; all bolded 6s, 7s, 8s and 9s form one; and all of these 1s, 2s, 4s and 5s also form one. From the definition, we have the following assertion, which is important in Phase 2 of Checker.

5	3	7	9	1	**6**	**8**	2	4
4	9	8	5	2	**7**	1	3	**6**
1	**2**	6	4	3	**8**	**9**	5	**7**
2	**1**	5	7	8	3	**6**	4	**9**
3	7	4	6	9	2	5	1	8
8	6	9	1	**5**	**4**	3	7	2
7	8	2	3	6	1	4	9	5
6	**5**	3	2	**4**	9	7	8	1
9	**4**	1	8	7	**5**	2	6	3

Figure 4: Three minimum unavoidable sets.

Assertion 1. Assume P to be a valid puzzle uniquely solved with a complete grid G. For each unavoidable set in G, at least one of the cells in the unavoidable set must be a clue in P. ▌

An unavoidable set S is called a *minimal unavoidable set*, or simply called a *MUS* in this thesis, if there exist no other smaller unavoidable sets $S' \subset S$. For example, in Figure 4, there are three MUSs: one with all bolded 1s and 2s, one with all bolded 4s and 5s, and one with all bolded 6s, 7s, 8s and 9s. The unavoidable set with all bolded 1s, 2s, 4s and 5s is not a MUS. In this example, the smallest size of MUSs is four and the second smallest size is six. In fact, four is also the smallest size among all MUSs.

Here, we introduce two approaches used in Checker to find MUSs from a complete grid in the following two subsections respectively.

Remove-Region Approach

The first, called the *remove-region approach*, is to quickly find the MUSs in a designated region of a complete grid. This approach is performing the following four steps.

1. Remove the digits from the designated region of a complete grid G, and let the

remaining digits form a new puzzle P.

2. Use a solver to solve P, producing many complete grids.
3. For each of the solved complete grids, the cells with different digits from those in G form an unavoidable set.
4. Among these found unavoidable sets, keep the minimum ones (MUSs).

3	8	9	4	2	1	5	7	6
6	1	5	9	8	7	4	3	2
7	4	2	3	5	6	9	1	8
2	3	1	6	9	4	7	8	5
9	6	8	2	7	5	1	4	3
5	7	4	1	3	8	2	6	9

(a)

			5	6	3			
			7	1	2			
			8	4	9			
3	8	9	4	2	1	5	7	6
6	1	5	9	8	7	4	3	2
7	4	2	3	5	6	9	1	8
			6	9	4			
			2	7	5			
			1	3	8			

(b)

Figure 5: Removing a region of digits, (a) one box row and (b) 2x2 boxes, from a complete grid.

1	9	7	5	6	3	8	2	**4**
8	5	6	7	**4**	2	3	9	**1**
4	2	3	8	**1**	9	6	5	7
3	8	9	4	2	1	5	7	6
6	1	5	9	8	7	4	3	2
7	4	2	3	5	6	9	1	8
2	3	1	6	9	4	7	8	5
9	6	8	2	7	5	1	4	3
5	7	4	1	3	8	2	6	9

Figure 6: Another solved complete grid.

Let us illustrate the approach by the complete grid, denoted by G, shown in Figure 1 (b). By using the approach, remove the upper box row (the upper three boxes) from the

complete grid G as a puzzle as shown in Figure 5 (a). Then, use a solver to solve the new puzzle. Surely, the original G must be one of the solved complete grids. Another one of the solved complete grids is shown in Figure 6, where the digits on the gray cells are different from those in the original G. Obviously, these gray cells form an unavoidable set, which is also a MUS since there exist no smaller unavoidable sets.

In the remove-region approach, the program Checker tried to remove three kinds of regions. The first is to remove a box row or a box column as shown in Figure 5 (a). Since there are three box rows and three box columns in a Sudoku grid, Checker needs to check six times for this kind of regions. The second is to remove 2x2 boxes as shown in Figure 5 (b). For this kind of regions, Checker needs to check nine times for a Sudoku grid. The third is to select three distinct digits, say 1, 2 and 3, and then remove all the 1s, 2s and 3s in the complete grid. For this kind of regions, Checker needs to check $C(9,3)$ $(=84)$ times for a complete grid.

The advantages of the remove-region approach is to find quickly all the MUSs in a designated region, regardless of the sizes of MUSs, sometimes up to 20 or more. However, the drawback of this approach is that some MUSs with small sizes cannot be found. For example, some MUSs with sizes about 10 cannot be found in this approach. Note that the search in Phase 2, described in the next subsection, performs more efficiently for smaller size MUSs.

Brute-Force Approach

The second, called the *brute-force approach*, uses a kind of brute-force method that is to search exhaustively all MUSs with different sizes, starting from 4 (the smallest size of MUSs). Namely, an initial set of MUSs with different sizes is prepared in advance, such as the one with all 1s and 2s in Figure 4. For each of these MUSs, the method checks all of its *isomorphic MUSs* and then find all matched in the complete grid. A MUS is said to be

isomorphic to another MUS, if both are the same after we relabel digits and rotate/mirror columns or rows of one like those described in the beginning of this chapter. Surely, the MUSs in the initial set are not isomorphic to one another.

The advantage of the brute-force approach is that one can find MUSs with small sizes that cannot be found in the above approach. In Checker, most MUSs[3] with sizes 12 or less were prepared in this approach.

The drawback of the approach is that checking all the isomorphic MUSs performs inefficiently since one MUS has many isomorphic MUSs but a complete grid contains only a few of them. Since Checker took much longer times in Phase 2 (about 1754.89 seconds for a primitive grid, described in greater details in Section 2.4), the overhead incurred by the brute-force becomes negligible. Thus, the brute-force approach is also used in Checker.

2.2.2.2. Phase 2: Searching n-clue Puzzles

Phase 2 is to use a tree search to find n-clue puzzles based on the MUSs found. By Assertion 1 described above, for each MUS, at least one clue in a valid puzzle must be located on one of cells in the MUS. Thus, given a number n, a complete grid G, and a set of MUSs, the program Checker in this phase is to find n-clue puzzles by recursively calling the tree search routine, named *ProcessTuple*(n, G, X_{cur}, C), where X_{cur} is the set of active MUSs and C is the set of clues being chosen. A MUS is called *active* in this thesis, if none of cells in the MUS are chosen as clues (in C) yet, and *inactive*, otherwise. Initially, all MUSs are viewed as active MUSs, and there are no clues initially. The routine is described as follows.

Routine *ProcessTuple*(n, G, X_{cur}, C):

[3] Checker prepares 47 kinds of MUSs for size 10, 44 for size 11, and 417 for size 12 in the initial set without proving that those are all.

1. If there exists at least one active MUS in X_{cur}, do the following.
    a. If the number of clues, denoted by $|C|$, is already n, return without any puzzles found.
    b. If $|C| < n$, find the active MUS, S, with the smallest size of cells. For each cell c in S, do the following.
        i. Choose c as a clue, and add it into C.
        ii. Update X_{cur} according to c. Namely, remove all active MUSs containing c from X_{cur}.
        iii. Recursively call the routine.
2. If there exists no active MUSs, that is, the set X_{cur} is empty, do the following.
    a. If $|C| = n$, check whether the puzzle with these n clues is valid or not. If valid, return this puzzle, an n-clue puzzle. If not, simply return without any puzzles found.
    b. If $|C| < n$, repeatedly perform the operations 1.b.i to 1.b.iii for each non-clue $c \notin C$ on the grid G.

At Step 1, the routine checks whether there exists at least one active MUS in X_{cur}, and performs, if so, the substeps 1.a and 1.b as follows. Consider the case that the routine has chosen n clues and at least one of MUSs is still active, not containing any clues. Then, the chosen n clues do not form a valid puzzle according to the Assertion 1. Thus, no more search is required, as described in Substep 1.a.

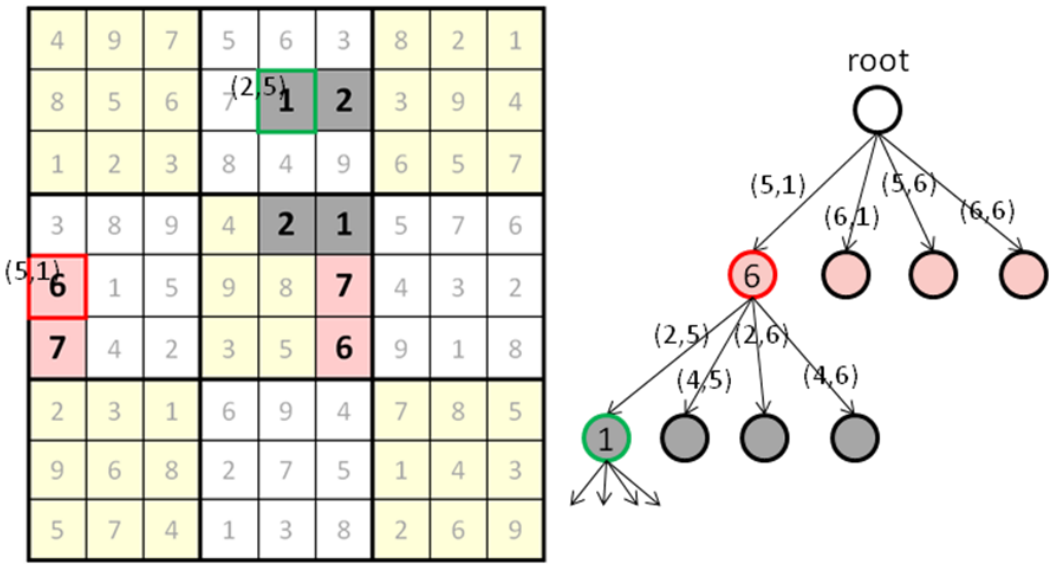


Figure 7: The search tree in Phase 2 of Checker.

In the case that the routine has chosen less than n clues (as described in Substep 1.b), choose one MUS S for further search. For each cell in S, add it into C, update the set of MUSs X_{cur} accordingly, and search more n-clue puzzles by recursively calling the routine itself. Note that the routine chooses the MUS with the smallest size of cells, since the one with less cells will expand a less number of subtrees. For example, in the complete grid given in Figure 7, if the cell with digit 6 at (5, 1) has been chosen, the routine finds another MUS with size 4 next, marked as gray in the figure, chooses one of these cells in the MUS as a clue, say the cell with digit 1 at (2, 5), and then recursively calls the routine to find more.

At Step 2, the routine performs Substeps 2.a and 2.b when no more active MUSs exist. In the case that the routine has chosen n clues (Substep 2.a), a solver is used to check whether the puzzle with the n clues is valid. If the puzzle can be solved with at least two distinct complete grids, the solver reports invalid. Otherwise, the solver reports valid, that is, a puzzle with n clues is found.

In the case that the routine has chosen less than n clues (Substep 1.b), it becomes more promising to find n-clue puzzles. In this case, we need to check all non-clue cells by

recursively performing the operations 1.b.i to 1.b.iii to search all n-clue puzzles.

The above routine needs to maintain the set of active MUSs efficiently. The maintenance includes the following two important operations, (a) finding the active MUS with the smallest size of cells, and (b) removing all MUSs containing a designated cell (chosen as a clue).

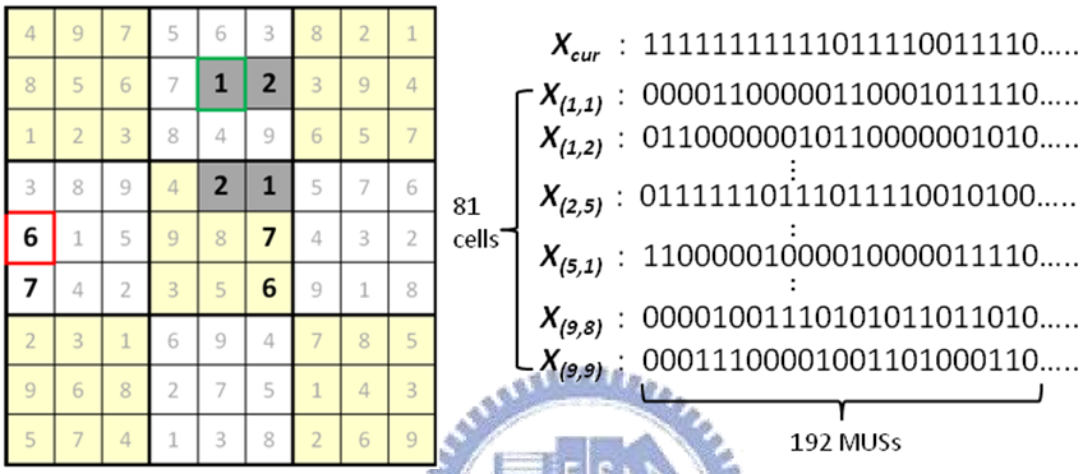


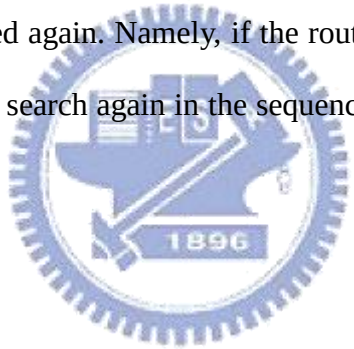
Figure 8: Data structures for the set of MUSs.

In the program Checker, the bit set data structure was used to implement the set of active MUSs, X_{cur}, as shown in Figure 8. Let each bit in the data structure be corresponding to a designated distinct MUS. Namely, the *i*th bit with 1 indicates the *i*th MUS to be inactive, while that with 0 indicates active. It is the same when switching the representation of values 0 and 1. Thus, for 192 MUSs, the default setting of Checker, the data structure requires 6 words each with 32 bits.

For operation (a), we can arrange the MUSs with small sizes to the front. For example, the MUSs with size 4, if any, are arranged to the front of bits of X_{cur}. So, if we want to find the *active* MUS with the smallest size, we simply scan bits of X_{cur} from the front to rear and find the first bit with 0 (indicating active). For example, if there exists some active MUS with size 6 and no active MUSs with size 4, we will find a MUS with size 6 by the scanning.

For operation (b), for each cell c, we initialize a set of MUSs X_c that contain the cell c. If we choose the cell c to be a clue, then the MUSs which contain the cell c become all inactive. Using bit set data structure, we can easily use bitwise operations to remove the MUSs from the set of active MUSs easily. Let X_c be also implemented by a bit set data structure. The *i*th bit in X_c is set to 1 to indicate that the *i*th MUS contains the cell c, and 0, otherwise. For example, in Figure 8, if we choose one more cell with digit 1 at (2,5) to be a clue, X_{cur} becomes the value of performing OR operation on the original X_{cur} and $X_{(2,5)}$. Since a Sudoku grid contains 81 cells, only 81 X_c need to be initialized.

From above, it can be shown that all the n-clue puzzles, if any, can be found by Checker. In addition, some more optimizations are done by this program. For example, the same set of clues are not searched again. Namely, if the routine selects the clue at (5,1) and then at (6,1), the routine will not search again in the sequence, selecting the one at (6,1) and then at (5,1).



2.3 DMUS Algorithm

As described at the beginning of this chapter, it would take a huge amount of time to solve the minimum Sudoku problem by Checker even in the job-level computation model with a lot of resources. In this section, we design a new algorithm in Phase 2, named Disjoint MUSs (DMUS) algorithm, and tune the code to improve the performance of Checker. The details of code tuning in both two phases are omitted in this thesis. This section focuses on the DMUS algorithm. Subsection 2.3.1 proposes the basic DMUS algorithm, while Subsection 2.3.2 proposes the improved DMUS algorithm.

2.3.1 Basic DMUS Algorithm

The basic DMUS algorithm improved the program Checker by modifying Step 1.b, described in Subsection 2.2.2.2. In Step 1.b, an initial operation is added to find $r+1$ disjoint active MUSs. Let r denote $n - |C|$, representing the number of remaining clues to be chosen, where $|C|$ is the number of clues in C. A set of MUSs are called *disjoint MUSs*, if any two of these MUSs do not overlap (namely, any two do not contain the same cells). An important assertion related to disjoint MUSs is described as follows.

Assertion 2. Use the program Checker to find n-clue puzzles as described in Subsection 2.2.2.2. If there exists at least $r+1$ disjoint active MUSs as above, then there exist no n-clue puzzles with C[4]. ▌

From Assertion 1 in Subsection 2.2.2.1, for each MUS, a valid puzzle must include at least one clue in the MUS. Since there exist at least $r+1$ disjoint active MUSs in addition to the clues in C, a valid puzzle with C must also contain at least $r+1$ disjoint clues, each from one distinct MUS. Thus, the number of clues in the valid puzzle must be at least $|C| + (r+1) = |C| + (n - |C| + 1) = n+1$. This implies that there exist no n-clue puzzles with C, that is, Assertion 2 is satisfied.

Given a set of MUSs, the problem of finding the largest number of disjoint active MUSs can be reduced to the maximum clique problem (cf. [34]). Unfortunately, the maximum clique problem is NP-complete [7]. Since it is intractable to find a maximum clique, it is also intractable to find the largest number of disjoint active MUSs via finding the maximum clique.

In the basic DMUS algorithm, we simply use a greedy algorithm to find $r+1$ disjoint active MUSs one by one without exhaustively searching all kinds of disjoint active

[4] A puzzle with *C* indicates a puzzle that contains at least the clues in *C*.

MUSs, such as backtracking. The algorithm repeatedly performs the following two operations until $r+1$ disjoint active MUSs are found or no more disjoint active MUSs exist.

1. Choose one additional disjoint active MUS with the smallest size in X_{cur}.
2. Add the chosen MUS into the set of disjoint MUSs.

In the first operation above, we choose the one with the smallest size, since it is more likely to find $r+1$ disjoint active MUSs in this way. This operation is the same as operation 1.b in Subsection 2.2.2.2, and therefore can be implemented by using the same bit operation.

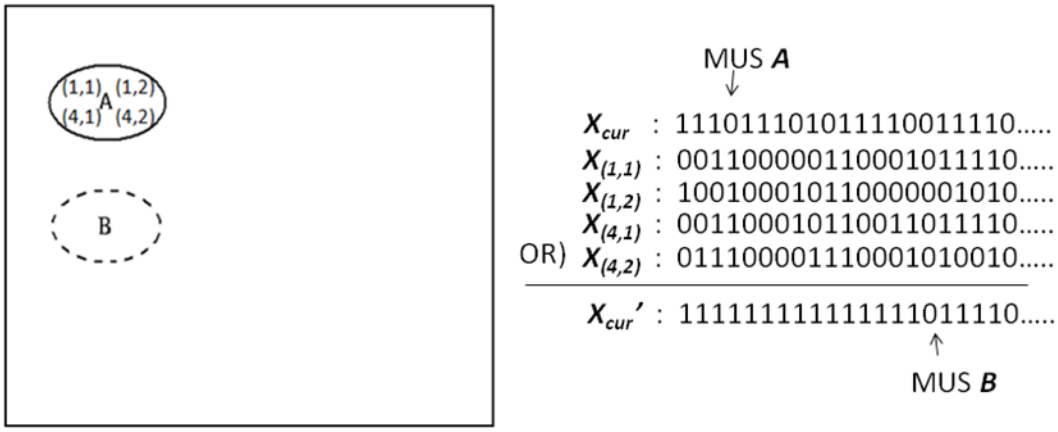


Figure 9: Finding the next disjoint MUS.

In the second operation, we add the chosen MUS, S, into the set of disjoint MUSs. We can implement it by pretending to select all cells in S as clues. Namely, we simply remove all the active MUSs in X_c from X_{cur} for all cells c in S. Thus, all the next chosen MUSs must not contain any cells in S. For example, in Figure 9, after we find the active MUS A, we can simply update the X_{cur} by removing X_c for all the four cells c in A. Thus, the next chosen MUS must not have any intersected cells with MUS A.

In fact, the operation can be easily improved by making a union of X_c for all cells c in S in advance. At the beginning of Phase 2 (or the end of Phase 1), for each MUS S, we

make an X_S which is the value of doing the OR operation on X_c for all cells c in S. Thus for the case in Figure 9, after selecting MUS A, we can simply update X_{cur} by making one OR operation for X_A, instead of four X_c for all cells c in A.

In the case that $r+1$ disjoint active MUSs are found by using the above algorithm, we can prune the whole subtree, since there exist no n-clue puzzles according to Assertion 2. In the case that r disjoint active MUSs or less are found, the program simply goes back to the normal operation 1.b (in Subsection 2.2.2.2) to traverse the whole search subtree.

2.3.2 Improved DMUS Algorithm

This subsection further improves the basic DMUS algorithm described in the previous subsection in the case that exactly r disjoint active MUSs are found. Let the r disjoint active MUSs be $S_1, S_2, \ldots, S_r$. Combining both Assertion 1 and Assertion 2, we obtain the following assertion.

Assertion 3. Use the program Checker to find n-clue puzzles as described in Subsection 2.2.2.2. Assume one finds r disjoint active MUSs, denoted by $S_1, S_2, \ldots, S_r$, as above. An n-clue puzzle with C must contain at least one of the cells as clues in each S_i with $1 \le i \le r$. ▌

Based on the assertion, a straightforward search tree needs to search about $\prod_i |S_i|$ puzzles, where $|S_i|$ is the size of S_i. In general, the performance is related to the sizes of these S_i. So, if these sizes are reduced, the performance is further improved.

In this subsection, we propose a new method to reduce the size of each Z_i, subset of S_i, while maintaining Assertion 4 (below), similar to Assertion 3, where $Z_1, Z_2, \ldots, Z_r$ are disjoint sets of cells, initialized to $S_1, S_2, \ldots, S_r$, respectively.

Assertion 4. From above, for each set Z_i, where $1 \le i \le r$, an n-clue puzzle with C must contain at least one of the cells in the set as clues. ▌

Assertion 4 is satisfied initially from Assertion 3. The new method to reduce the size of each Z_i is described in the following routine.

Routine *Shrink*(*i*):

1. Let $Z = (Z_1 \cup Z_2 \cup ... \cup Z_r) - Z_i$.
2. For each active MUS S (without containing any clues in C) disjoint with Z, let $Z_i = Z_i \cap S$.

Let us illustrate by an example in Figure 10. Assume that r is three, and assume to find the three active disjoint MUSs S_1, S_2 and S_3. As described above, Z_1, Z_2 and Z_3 are initialized to S_1, S_2 and S_3, and Assertion 4 is satisfied initially.

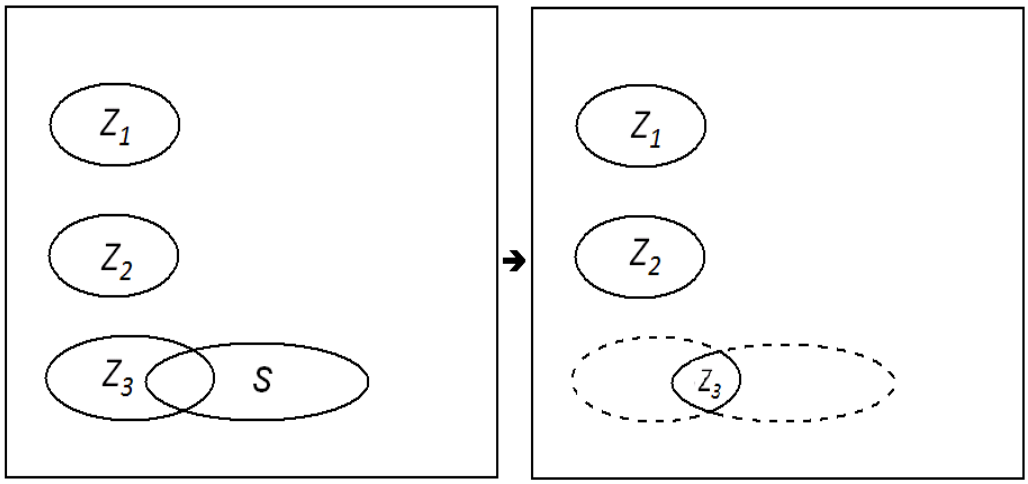


Figure 10: Shrink the Z_3 to the intersection of Z_3 and S.

Since r is three, we need to choose three more clues, each of which must be located in S_1, S_2 and S_3, respectively. Let us use *Shrink*(*3*) to shrink Z_3. In the routine, Z is initially set to $Z_1 \cup Z_2$. Assume that some other active MUS S is disjoint with Z (both Z_1 and Z_2) as shown in the left of Figure 10. The set Z_3 is shrunk to be the intersection of the original Z_3 and S as shown in the right of Figure 10.

Assertion 4 still holds for the new Z_3 together with both Z_1 and Z_2 for the following reason. Assume for contradiction that none of clues in an n-clue puzzle with C

are located in the new Z_3. From above, the clue that is located in the original Z_3 must be outside the MUS S. Since S is an active MUS disjoint with both Z_1 and Z_2, none of clues are located in S. Thus, according to Assertion 1, the n-clue puzzle is not valid, contradicting the assumption. This shows that the clue in the original Z_3 must be in the new Z_3, too.

Based on the above illustration, it can be easily derived that Assertion 4 still holds after performing *Shrink*(*i*). Namely, if Assertion 4 holds currently, then it will also hold after *Shrink*(*i*). By induction, Assertion 4 is maintained by repeatedly performing the routine *Shrink*.

Since the set Z_i may shrink after *Shrink*(*i*), it becomes very likely to shrink other Z_j further, where $j \neq i$. Therefore, it is reasonable to repeatedly perform *Shrink* many times.

Many strategies can be used to perform *Shrink* repeatedly. For the example in Figure 10, we may choose the sequence, *Shrink*(*1*), *Shrink*(*2*) and then *Shrink*(*3*), or the sequence, *Shrink*(*3*), *Shrink*(*2*) and then *Shrink*(*1*). We may even choose the sequence, *Shrink*(*1*), *Shrink*(*2*), *Shrink*(*3*), *Shrink*(*2*) and then *Shrink*(*1*). More discussion is given in our experiments in Subsection 2.4.4.

In the case that some Z_i becomes empty after the routine *Shrink*(*i*) is finished, we can easily derive from above that none of n-clue puzzles exist. Thus, we can prune the whole subtree at Step 1.b in the routine ProcessTuple (in Subsection 2.2.2.2), like the case that we have $r+1$ MUSs. In fact, the case of $r+1$ disjoint active MUSs can be simply viewed as a special case. Let Z_1, Z_2, …, Z_r be the first r disjoint active MUSs. Then, for *Shrink*(*r*), the set Z_r becomes empty when choosing the last disjoint MUS Z_{r+1} as S. Namely, $Z_r = Z_r \cap S = Z_r \cap Z_{r+1}$ is empty.

In the case that none of Z_i becomes empty, we choose the one, say Z_j, with the smallest size among all Z_i, and then continue the search in the operation 1.b by using Z_j,

instead of the original S_1, the MUS with smallest size. Thus, the branching factor of the search tree becomes smaller, and therefore the size of the whole search tree is greatly reduced.

Besides the algorithm above, we also did many other tunings on the modified program. The details of these tunings are omitted in this thesis.

2.4 Experiment

We implemented the basic DMUS algorithm and the improved one as described in the previous section by modifying the program Checker. For performance analysis, all experiments were done on a personal computer equipped with the CPU, Intel(R) Xeon(R) E5520 @ 2.27GHz. In the rest of this thesis, one core indicates the above computing power.

Since it took a long time for the original program Checker to find 16-clue puzzles from one primitive grid, we only chose 100 at random among the 5,472,730,538 primitive grids (generated by the Fowler's program [18] as mentioned above) as our benchmark for comparisons. The 100 primitive grids are listed in the webpage of the Sudoku project [52]. For simplicity of discussion, all experimental results in the rest of this section are given on the average of the chosen 100.

In the rest of this section, Subsection 2.4.1 analyzes the performance results in Phase 2 by comparing different versions of the program. Subsection 2.4.2 shows the results of the modified program in Phase 1. Subsection 2.4.3 shows the overall performance by including tuning the performances in Phase 1 of the program using different techniques. Subsection 2.4.4 compares the performances for different sequences of *Shrink*(*i*) in the DMUS algorithm. Subsection 2.4.5 shows the number of nodes in each level of Phase 2 in Checker.

2.4.1 The Results in Phase 2

In addition to the DMUS algorithm described in Section 2.3, our implementation also included many tunings, which are either omitted or briefly described due to tediousness. In this subsection, we analyze the performances of the following versions of implementations.

Version IDs	Descriptions of versions
V_1	Original Checker
V_2	V_1 with some turnings like reordering MUSs
V_3	V_2 with basic DMUS algorithm
V_4	V_3 with improved DMUS algorithm
V_5	V_4 with some tunings on Phase 2
V_6	V_5 with MUSs generated by new Phase 1

Table 2: The descriptions of all versions

As shown in Table 2, all versions are described as follows. The original version of Checker is denoted by V_1. Before implementing the basic DMUS algorithm, we tuned the program by reordering the selection sequence of MUSs based on the sizes of MUSs and some other factors. After the tuning, the version is denoted by V_2. The version is denoted by V_3 after incorporating only the basic DMUS algorithm into V_2. Similarly, the version is denoted by V_4 after incorporating only the improved DMUS algorithm into V_3. Then, we made additional tunings on Phase 2 of version V_4, such as reordering the sequences of MUSs and cells in MUSs during search, and the version is denoted by V_5. All the MUSs used in the versions V_1 to V_5 were generated by the original Checker. The last version, denoted by V_6, was the same as V_5, except that all the MUSs were generated in Phase 1 by our modified program. Since this subsection focuses on the performances in Phase 2, the version V_6 will be discussed in the next subsection, not in this subsection.

# of MUSs	128	192	256	320	384	448	512	The fastest	Speed-up
V_1	2093.41	**1754.89**	1811.61	1926.37				1754.89	1.00
V_2	1210.80	586.75	**576.87**	617.95				576.87	3.04
V_3	818.03	93.65	52.77	**44.51**	47.65	50.86	52.75	44.51	39.43
V_4	704.57	64.76	25.68	19.95	**19.54**	20.83	21.74	19.54	89.81
V_5	705.98	59.51	18.85	13.00	**12.45**	12.71	13.07	12.45	140.96
V_6	730.69	56.01	19.06	13.28	**12.84**	12.93	13.42	12.84	136.67

Table 3: The averaged time of solving one primitive grid in Phase 2 for each version

Table 3 shows the averaged time of solving one primitive grid in Phase 2 in each version. In this table, we also tried different numbers of MUSs, such as 128, 192, 256, 320, 384, 448 and 512. As described above, all the MUSs used in the versions V_1 to V_5 were generated by the original Checker. According to our experiments, about 358.4 MUSs were generated on average for a primitive grid. The versions V_1 and V_2 did not run the cases for 384 MUSs or more simply because the original Checker did not support them.

In general, the more MUSs we used, the smaller search tree. Assume that more MUSs are available in Phase 2. Then, it is more likely to choose Substep 1.a to stop calling recursively. Thus, it makes the search tree smaller. Besides, more active MUSs may also help prune the search tree in our DMUS algorithm.

Searching smaller trees usually tends to raise performance, but on the other hand more MUSs may incur extra overhead. From Table 2, we observe the following: The version V_1 reached the best performance for 192 MUSs, version V_2 for 256, version V_3 for 320, and version V_4 and V_5 for 384. When the numbers of MUSs decreased from the above numbers (for the best performances), the corresponding performances went down. On the other hand, when the numbers of MUSs increased from the above values, the performances went down due to the overhead incurred by the large set of MUSs.

Comparing all versions by their best performances, we obtained that the speedups with

respect to the version V_1 were 3.04, 39.43, 89.81 and 140.96 respectively for versions V_2 to V_5. More specifically, the DMUS algorithm improved significantly the performance by a factor of 29.54 (through V_2 to V_4), especially that the basic DMUS algorithm improved by a factor of 12.97 (through V_2 to V_3). Except the DMUS algorithm, the other tunings (through V_1 to V_2 and through V_4 to V_5) improved by a factor of 4.77.

2.4.2 The Results in Phase 1

In this subsection, we want to discuss the experimental results in Phase 1. As described in Subsection 2.2.2.1, Phase 1 of the original Checker used both remove-region and brute-force approaches to find MUSs.

According to our experiments, for the remove-region approach, the original Checker found the MUSs in the designated regions and kept the MUSs with sizes 14 or less. The program with this approach ran very fast in about 0.5 seconds in Phase 1, and it was able to find only about 222.54 MUSs on average for a complete grid, among which 139.41 have sizes 12 or less.

In fact, the program also used the brute-force approach to search the MUSs with size 12 or less and was able to find about 358.4 MUSs on average for a complete grid, but it took much longer time, about 37.4 seconds, to find MUSs for a complete grid. Since the original Checker took a much longer time in Phase 2 (about 1754.89 as shown in the previous subsection), the computation time, 37.4 seconds, is negligible. Thus, it is more important for the program to use the above approach to find higher quality MUSs.

However, since our DMUS algorithm improves the performance significantly in Phase 2 as described in the previous subsection, the computation time for Phase 1 also becomes critical. Thus, we need to improve the performance in Phase 1. Our approach is to investigate the remove-region approach instead of the brute-force approach.

We improved the remove-region approach by magnifying the removed regions. In addition to removing three distinct digits, the third kind of regions described in Subsection 2.2.2.1, we also removed all combinations of regions with four boxes, and some more combinations to ensure to find all the MUSs with size 10 or less. After the tuning, we successfully reduced the averaged computation time in Phase 1 for each primitive grid from about 37.42 seconds down to about 1.09 seconds, while still obtaining high quality MUSs for Phase 2.

For the quality of MUSs, let us simply compare the performances of both versions V_5 and V_6 in Phase 2, since both versions were the same except for the used MUSs. From Table 2, most performances in V_6 in Phase 2 were, in general, slightly worse than those in V_5. In the case of 384 MUSs, where both versions reached the best performance, the performance in V_6 was reduced by only about 3% in Phase 2 when compared with that in V_5. The averaged computation time in Phase 2 for each primitive grid in V_6 was only 12.84 seconds. Thus, the quality and quantity of the MUSs generated by our new approach were nearly equivalent to those by the brute-force. However, in Phase 1, the performance in V_6 was much better than that in V_5.

Sizes of MUSs	≤ 11	12	13	14	≥15	Total
Original Checker	140.17	135.10	22.44	60.69	0	358.40
Modified program	140.17	106.22	61.58	156.48	283.54	747.99

Table 4: The number of MUSs for each size found by the programs

Table 4 shows the numbers of MUSs for each size found by the original Checker and our modified program. On average, we were able to find about 747.99 MUSs, which generally included more MUSs than those by the original. More specifically, for the 747.99 MUSs found by the new approach, the number of MUSs with each size less than 12 was the same as that by the brute-force, the number of MUSs with size 12 was slightly smaller, and

the number of MUSs with each size larger than 12 was much higher.

2.4.3 Overall Performances

By adding the computation time in Phase 1, the averaged computation time for each version is shown in Table 5. The original version, V_1, took about 1792.31 seconds for each primitive grid. We estimate that it would take about 311,000 years to check all 5,472,730,538 primitive grids to solve the minimum Sudoku problem. But the averaged computation time for V_6 was greatly reduced to 13.93 seconds for each primitive grid. Thus, we estimate that it would take only 2417 years on one core to solve the minimum Sudoku problem. The total speedup was about 128.67.

# of MUSs	128	192	256	320	384	448	512	The fastest	Speed-up
V_1	2130.83	**1792.31**	1849.03	1963.79				1792.31	1.00
V_2	1248.22	624.17	**614.29**	655.37				614.29	2.92
V_3	855.45	131.07	90.19	81.93	**85.07**	88.28	90.17	81.93	21.88
V_4	741.99	102.18	63.10	57.37	**56.96**	58.25	59.16	56.96	31.47
V_5	743.40	96.93	56.27	50.42	**49.87**	50.13	50.49	49.87	35.94
V_6	731.78	57.10	20.15	14.37	**13.93**	14.02	14.51	13.93	128.67

Table 5: The averaged time of solving one primitive grid for each version

In order to have more confident in the result, we also randomly chose another set of 100 primitive grids and ran them again using V_1 for 192 MUSs and with V_6 for 384 MUSs, and the times for them were 1704.25 seconds and 10.99 seconds, respectively. The speedup was about 155.07, more than the above result, 128.67. Furthermore, we randomly chose another set of 10000 primitive grids and ran them using V_6 for 384 MUSs. For the 10000 primitive grids, each was solved in about 12.72 seconds on average, close to the above results for 100 grids. We did not try V_1 or other versions since they would take a large amount of time. These chosen primitive grids are also listed in the webpage of the Sudoku

project [52].

2.4.4 Different Sequences of Shrinks in the Improved DMUS Algorithm

For the improved DMUS algorithm described in Subsection 2.3.2, we may choose different sequences of *Shrinks*. In our experiments, we considered the following six sequences.

1. Perform *Shrink*(*1*) only.
2. Perform *Shrink*(*1*), *Shrink*(*2*), …, *Shrink*(*r*).
3. Perform *Shrink*(*r*), *Shrink*(*r-1*), …, *Shrink*(*1*).
4. Perform *Shrink*(*1*), *Shrink*(*2*), …, *Shrink*(*r*), *Shrink*(*r-1*), …, *Shrink*(*1*).
5. Perform the second sequence (above) twice.
6. Perform the second sequence (above) repeatedly until no more clues could be pruned.

Method	1	2	3	4	5	6
Average Time (sec)	18.18	13.97	13.93	15.62	16.88	17.16

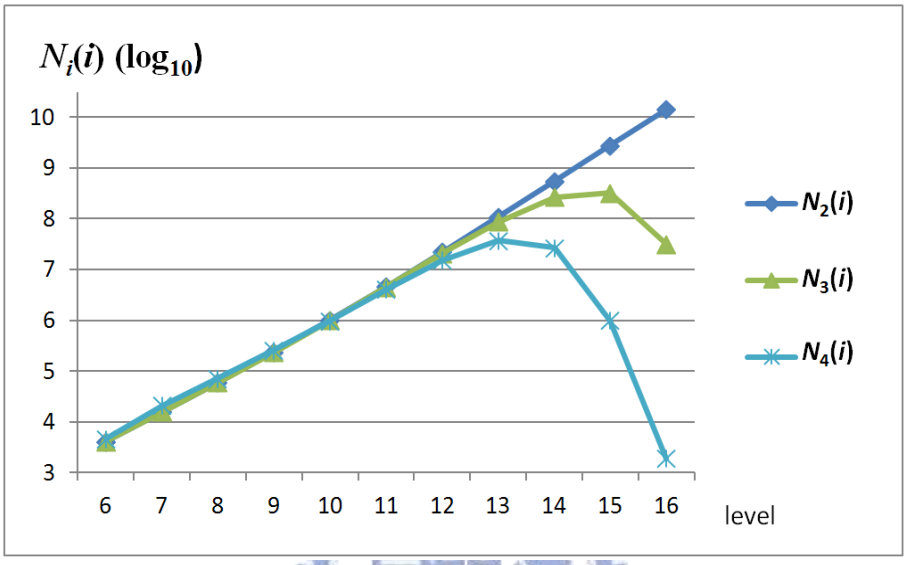
Table 6: The average solving times of using different sequences

Table 6 shows the performances of version V_6 using the above sequences respectively. This result indicates that the version performed best by using the second and third sequences and that the third performed slightly better than the second. For version V_4, we also obtained a similar result. Therefore, we simply chose the third in our experiments in Table 3 and Table 5 (above).

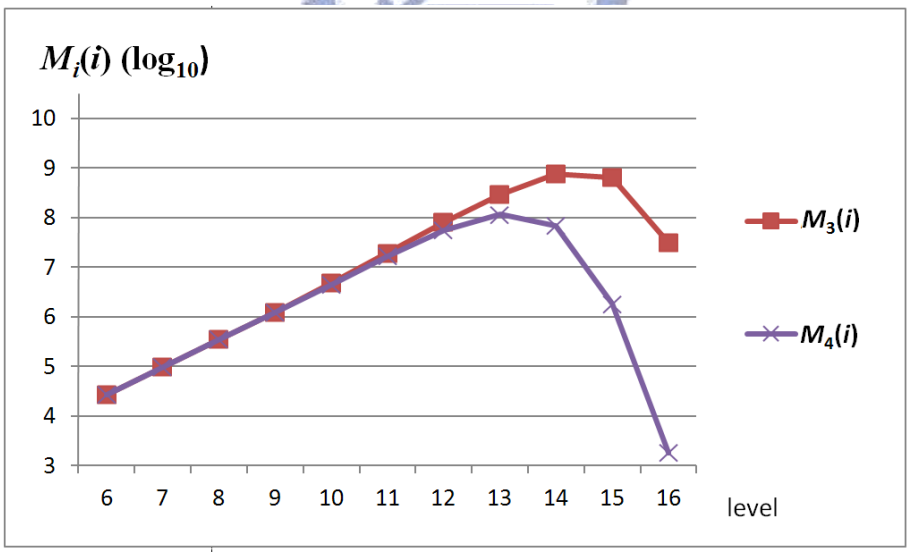
2.4.5 Node Counts in Phase 2

The key of the DMUS algorithm is to reduce greatly the number of nodes by paying the price of finding disjoint MUSs. This subsection investigates the number of visited nodes

and the number of disjoint MUSs in Phase 2 in versions V_2, V_3 and V_4. Let $N_2(i)$, $N_3(i)$ and $N_4(i)$ denote respectively the total numbers of nodes at level i of the search tree in V_2, V_3 and V_4, and $M_3(i)$ and $M_4(i)$ denote respectively the total numbers of disjoint MUSs generated from the nodes at level i of the search tree in both V_3 and V_4.



(a)

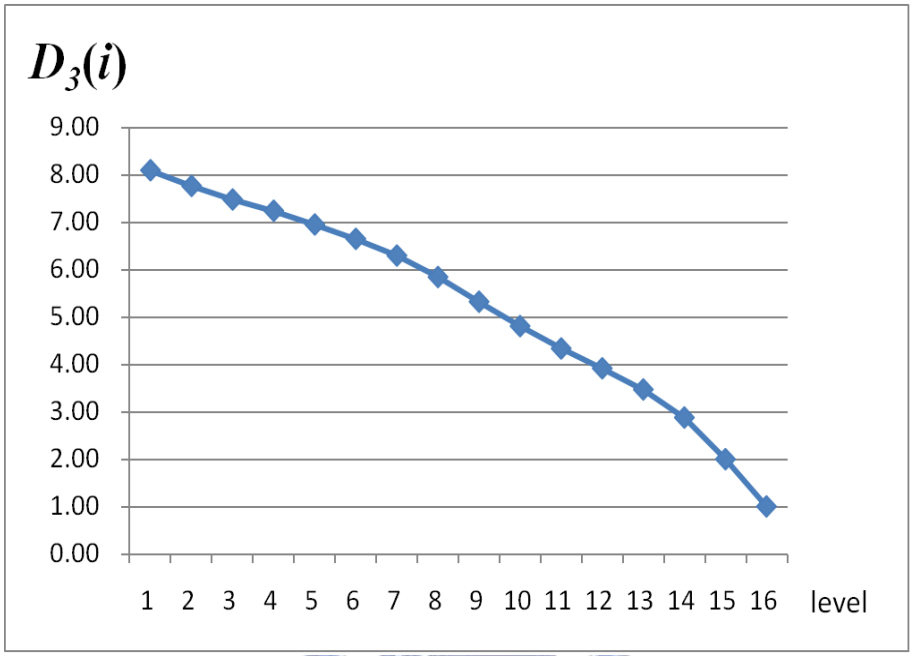


(b)

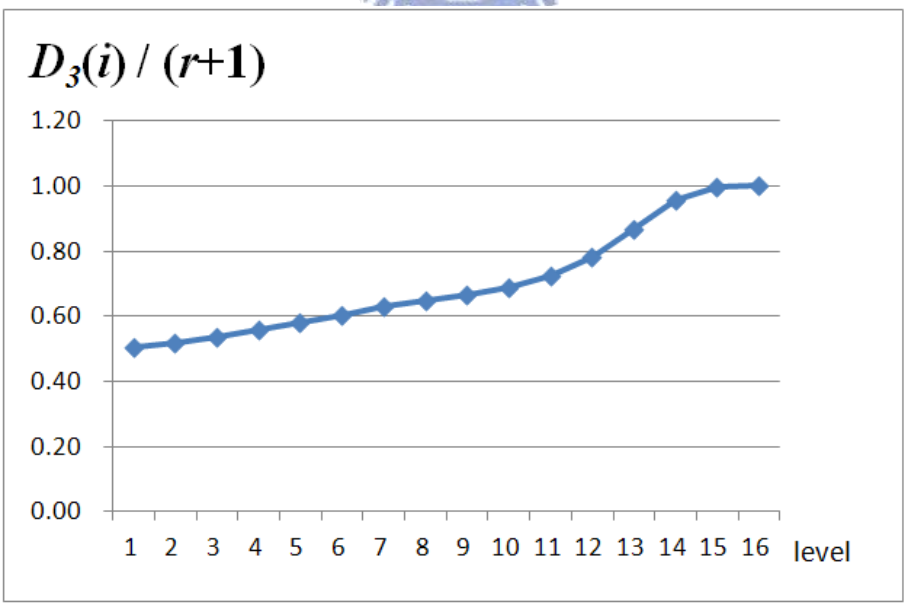
Figure 11: The numbers of (a) visited nodes and (b) disjoint MUSs

Figure 11 shows these numbers (in $\log_{10}$) at each level. For $N_2(i)$, it is clear that the maximum is at level 16 and is up to 14 billion. The maximum for $N_3(i)$ is shifted to level 15

and is up to 322 million, while the maximum for $M_3(i)$ is to level 14 and is up to 745 million. Again, the maximum for both $N_4(i)$ and $M_4(i)$ are shifted to level 13 and are up to 37 million and 115 million respectively. This shows that the two DMUS algorithms are able to reduce the numbers of nodes at higher levels, which are normally enormous.



(a)



(b)

Figure 12: (a): D3(i) (b): the ratio D3(i)/(r+1)

Now we investigate the average number of disjoint MUSs generated from each node at level i, denoted by $D_3(i)$ in version V_3. Figure 12 shows $D_3(i)$ and the ratio of $D_3(i)/(r+1)$ at each level i. From the figure, the ratio is near 1 at levels 14 to 16. This also implies that it is highly likely to find more than r disjoint MUSs and therefore prune most of the subtrees rooted at levels 14 to 16. This explains why the DMUS algorithm performed well.

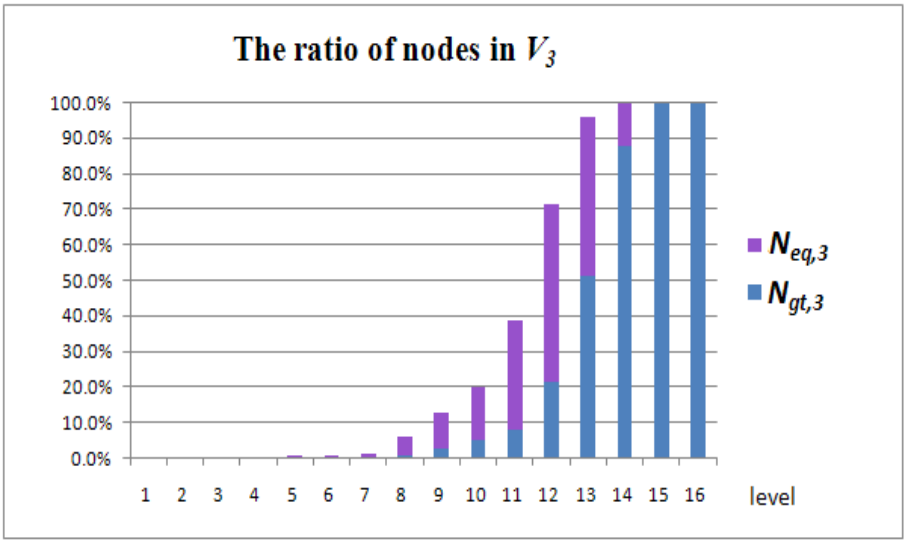


Figure 13: $N_{eq,3}(i)$ and $N_{gt,3}(i)$

Let us look into the value $N_3(i)$ more closely. Let $N_{eq,3}(i)$ denote the total number of nodes at level i, which generate exactly r disjoint MUSs, and $N_{gt,3}(i)$ denote the total number of nodes at level i, which generate greater than r disjoint MUSs. $N_{gt,3}(i)$ indicates that $N_{gt,3}(i)$ nodes at level i can be pruned, and $N_{eq,3}(i)$ indicates that $N_{eq,3}(i)$ nodes at level i may be further pruned by the improved DMUS algorithm. Figure 13 indicates that the ratio of $N_{eq,3}(i)$ to $N_3(i)$ is significant at levels 7 to 14. This motivated the improved DMUS algorithm.

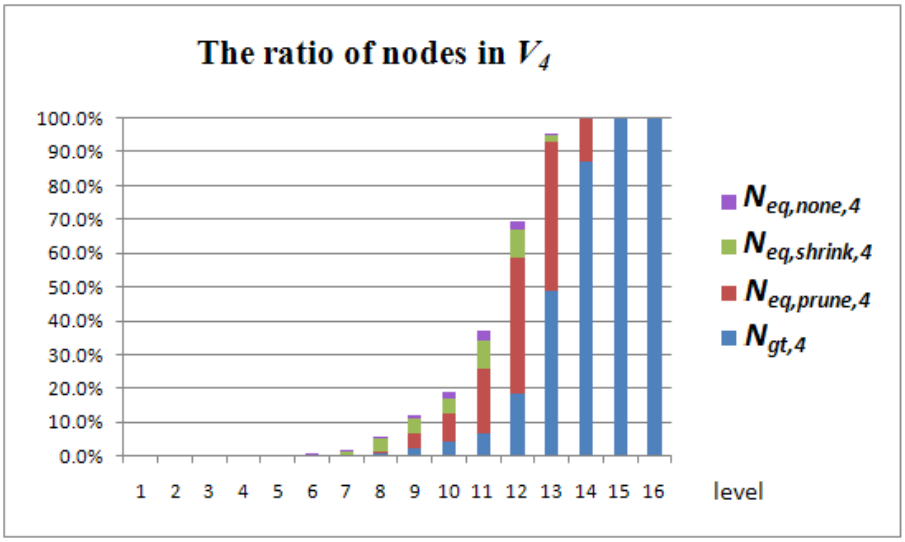


Figure 14: $N_{eq,none,4}(i)$, $N_{eq,shrink,4}(i)$, $N_{eq,prune,4}(i)$ and $N_{gt,4}(i)$

In version V_4, we further separate the value $N_{eq,4}(i)$ into $N_{eq,prune,4}(i)$, $N_{eq,shrink,4}(i)$ and $N_{eq,none,4}(i)$. $N_{eq,prune,4}(i)$ denotes the number of the $N_{eq,4}(i)$ nodes which can be all pruned by the improved DMUS algorithm, as described in Subsection 2.3.2. Similarly, $N_{eq,shrink,4}(i)$ denotes the number of the $N_{eq,4}(i)$ nodes which can be partially pruned, and $N_{eq,none,4}(i)$ denotes the number of the $N_{eq,4}(i)$ nodes which cannot be pruned at all. Figure 14 shows a large portion for $N_{eq,prune,4}(i)$ and a very small portion for $N_{eq,none,4}(i)$ in most $N_{eq,4}(i)$. This indicates that the improved DMUS algorithm is effective.

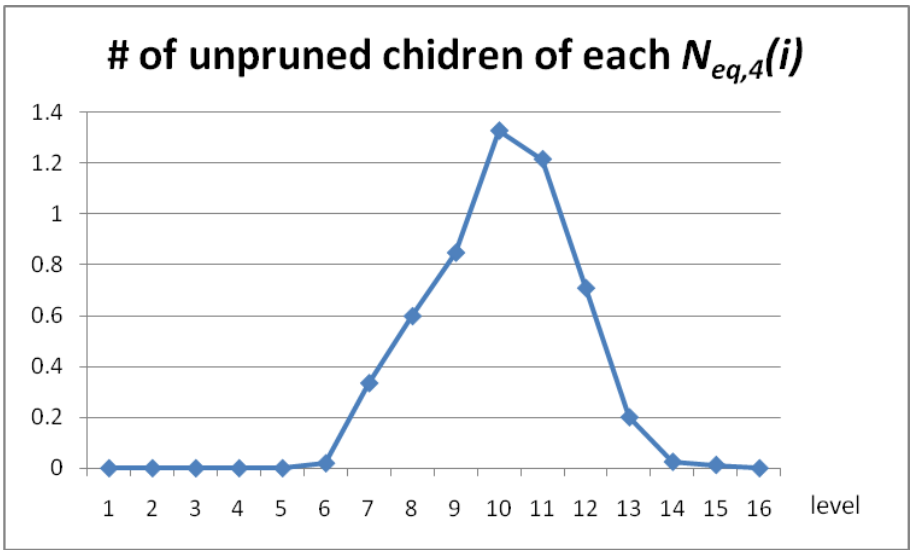


Figure 15: The average number of children generated from each of the $N_{eq,4}(i)$ nodes

Figure 15 shows the average number of children generated from each of the $N_{eq,4}(i)$ nodes, which cannot be pruned. In this figure, we observe that only the $N_{eq,4}(i)$ nodes at both levels 10 and 11 generated more than one child on the average in the tree search, and the remainder generated less than one child on the average when using the improved DMUS algorithm. This shows that $N_{eq,4}(i)$ nodes do not generally grow exponentially. This demonstrates the advantage of the improved DMUS in another aspect.

2.4.6 The Analysis of primitive grids

All primitive grids will be checked by the modified Checker when solving the minimum Sudoku problem. The average time of checking each primitive grid is about 13.93 seconds. However, the time for each primitive grid can be very different and we want to investigate it.

We divided all the primitive grids into 52 parts and analyzed the average computation time for each part. Each part contains 1024 * 1024 jobs (1024 * 1024 * 100 grids), except for the last part. For each part, we randomly chose 1000 grids to analyze the computation time.

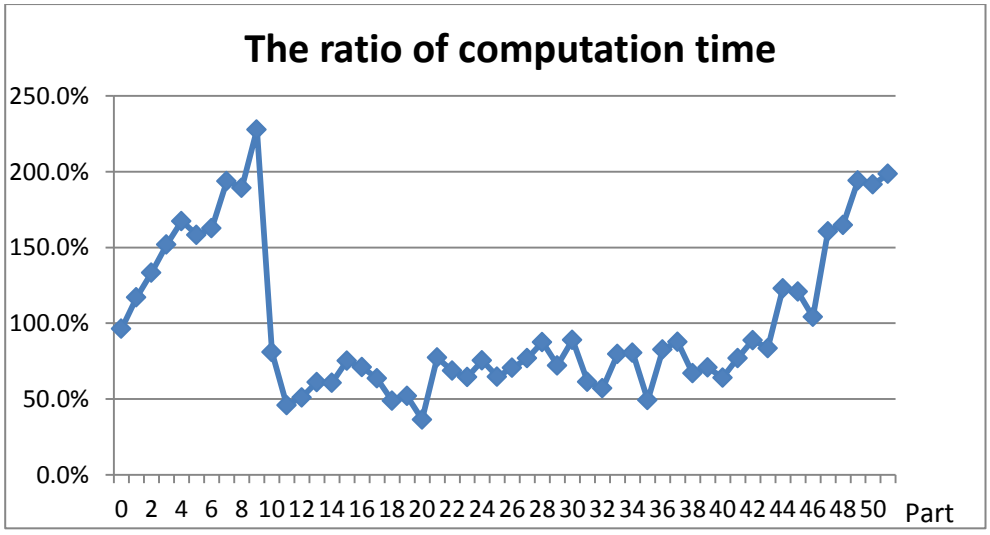


Figure 16: The ratio of computation time for each part compared to the average time

Figure 16 shows the ratio of computation time for each part when compared to the average time of all primitive grids. The parts with more than 100% take more time rather than the average. We can observe that some of the parts take 2 times of the averaged time, and some of the parts take less than half of that.

This experiment is helpful when solving the minimum Sudoku problem. Since the problem takes about several years to solve, we need to make sure the progress is on schedule when solving, and this experiment can help us.

2.5 Conclusion

We propose a new approach to solve the minimum Sudoku problem more efficiently. This thesis presents a more efficient algorithm, named DMUS, incorporates it into the program Checker, and makes some more modifications including tuning the program to greatly reduce the computation times for finding 16-clue puzzles from primitive grids.

According to our experiments, it took about 1792.31 seconds for the original Checker to solve one primitive grid on average. In contrast, our improved program presented above was able to solve one in 13.93 seconds on average. Thus, it is estimated that it takes only about 2417 years on one core to check all 5,472,730,538 primitive grids to solve the minimum Sudoku problem, while it would take the original about 311,000 years. If we had 10,000 cores, then we would solve it within three months, but more than 30 years by the original Checker.

Using the modified program, it becomes more feasible to solve the problem on top of BOINC [5]. Thus, we initiated a Sudoku project [52] on top of BOINC [5] to solve the minimum Sudoku problem by using the modified program on October 2010. Since each primitive grid can be checked within about 13.93 seconds, which is very fast, we

encapsulated 100 primitive grids as one job in this project, which may take about 1393 seconds.

At the end of July 2013, the Sudoku project has been running more than 680 million credits on BOINC [5] and has completed the checking of more than 93% primitive grids, and no 16-clue grids have been found yet. We expect to finish soon, sometime in 2013.

Chapter 3 Job-Level Volunteer Computing

Since traditional volunteer computing cannot help solve some computer game problems efficiently as described in Section 1.2, this chapter introduces our job-level volunteer computing (JLVC) for solving computer games applications. The JLVC uses connection model, and thus the JLVC can solve computer game problems efficiently.

This chapter is organized as follows. The JLVC model is proposed in Section 3.1. The generic search is described in Section 3.2, while the generic job-level search is described in Section 3.3.

3.1 JLVC Model

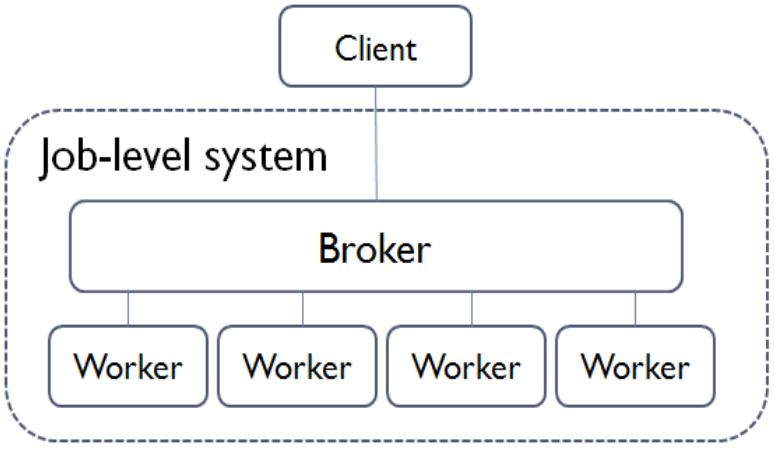


Figure 17: The job-level volunteer computing model

In the JLVC model, the computation is done by a *client* which dynamically creates *jobs* to do. For example, in a computer game application, the client creates one job for each move in a position, and each job evaluates the value of the corresponding move.

A *JLVC system*, or called *job-level system*, as shown in Figure 17 includes a set of

workers which helps perform jobs. In the system, jobs created by clients are dispatched to a *broker* which selects available workers to perform.

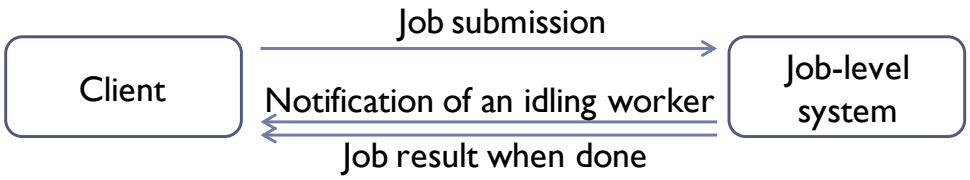


Figure 18: The messages between a client and the job-level system

As shown in Figure 18, messages between the client and the job-level system mainly include the following three things: job submission, notification of an idling worker, and job result. The first one is from the client to the system, while the next two are the other way around.

In the job-level model, the clients wait passively for the available workers to submit jobs. Whenever a worker is available for computing a job, it will notify the broker and the broker will in turn notify the client that one worker is available. Then, the client submits one job, if any, to the broker which in turn dispatches the job to the worker. When completing the job, the worker sends the job result back to the client, which then updates according to the result. During the update, more jobs may be generated for job dispatching.

In the model, the client usually does not actively submit a large number of jobs to the job-level system in advance. For example, for a position with 10 moves, assume that the client actively creates 10 jobs each per move in advance, and submits them to a job-level system with 2 workers only. In the case that one of these moves turns promising, say nearly winning, a good strategy is to shift the computing resources from other moves temporarily to this promising move and its descendants. However, in the case of submitting a large number of jobs in advance, the workers still work on other moves, unrelated to the promising move.

The job-level model was realized in a desktop system designed by Wu *et al.* in [59]. In practice, a job-level system may also support some other messages, such as abortion messages, ask-info messages, etc. Abortion messages can be used to abort running jobs which are no longer interesting. For example, if a move is found to be a sure win from a job, other jobs for its sibling moves are no longer interesting and therefore can be aborted immediately. Ask-info messages can be used to ask the job-level system to report the job status for monitoring.

3.2 Generic Search

This section describes generic best-first search, or simply called generic search, that fits many search techniques, like PNS and Monte-Carlo Tree Search (MCTS) [9]. Generic search is associated with a search tree, where each node represents a game position. The process of a generic search usually repeats the following three phases, selection, execution, and update, as shown in Figure 19 (below).

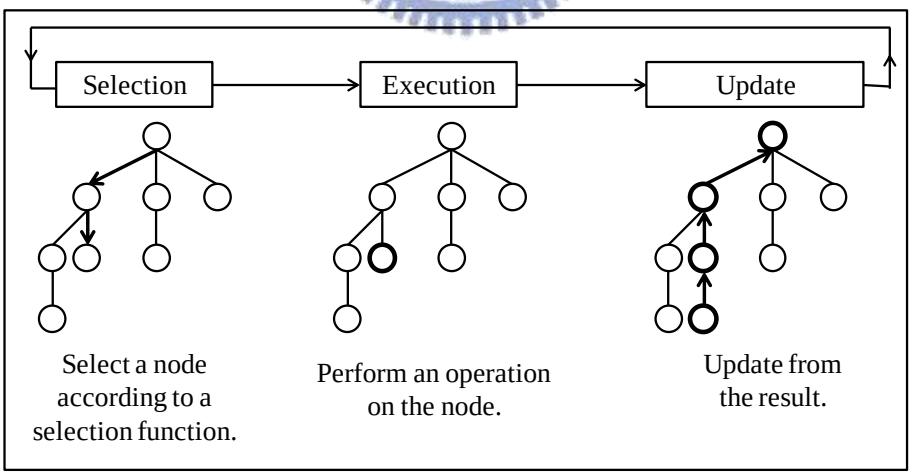


Figure 19: Outline of a job-level computation model for single core

First, in the selection phase, a node is selected according to a selection function based on some search technique. For example, PNS selects the most proving node (MPN, which will be described in more detail in Subsection 3.4.1.1); and MCTS selects a node based on

the so-called *tree policy* (defined in [9][20]). Note that the search tree is supposed to be unchanged in this phase.

Second, in the execution phase, an operation $J(n)$ is performed on the selected node n, but does not change the search tree yet. For example, find the best move from a node n, expand all moves of n, or run a simulation from n for MCTS. After performing the job $J(n)$, a result is obtained. For the above example, the result is the best move, all the expanded moves, or the result of a simulation, respectively.

Third, in the update phase, the search tree is updated according to the job result. For the above example, a node is generated for the best move, nodes are generated for all expanded moves, and the status is updated on the path to the root.

3.3 Generic Job-Level Search

From the previous section, the operation $J(n)$ on the selected node n does not change the search tree. Therefore, the operation $J(n)$ can be done as a job by another worker remotely in a job-level system. The job submission may include some data required by $J(n)$, such as the neighboring nodes or the path to the root. Thus, a generic search becomes a generic job-level search, run in a job-level with one worker only.

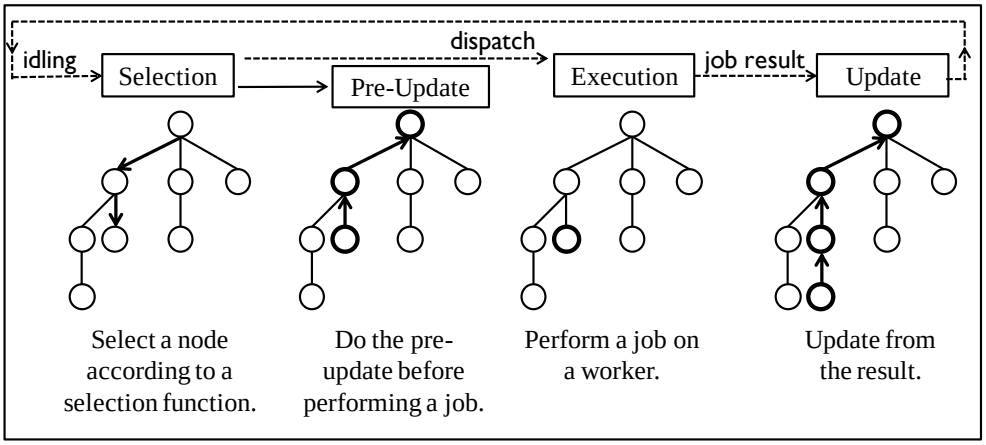


Figure 20: Outline of a job-level model

However, since a generic search repeats the three phases sequentially (as shown in Figure 19), the job-level system with multiple workers is not efficient. Thus, in generic job-level search, the computation model is changed to be run in parallel as shown in Figure 20. The details are described as follows.

As described in Section 3.1, the client waits passively for notification of idling workers. When receiving a notification, the client selects a node in the selection phase and then dispatches a job, if any, to the worker for execution. When the job is done, the worker sends the result back to the client. When receiving the result, the client runs the third phase to update the search tree from the result. These are all performed in an event-driven model.

In a job-level system with multiple workers, one issue is that in the above model the client will select the same node for multiple notifications of idling workers, if no other results are found and used to update the search tree in the interim.

In order to solve this issue, we modify the model as in Figure 20 by adding one phase, called the *pre-update phase*, after the selection phase and before job submission. In this phase, several policies can be used to update the search tree. For example, the flag policy sets a flag on the selected node, so that the flagged nodes will not be selected again.

Another issue deals with growth of the search tree, such as node expansion or generation from the search tree. Consider a case that a leaf node n is selected. If the job $J(n)$ is to expand all moves, all the child nodes (corresponding to these moves) are expanded from n. However, in many cases, it is inefficient to expand all moves in the job-level model.

If $J(n)$ is to find the best move, then, in the update phase, the node corresponding to the best move should be generated (usually by running a game-playing program for $J(n)$, such as NCTU6). However, the question is when and how to expand other nodes such as those for the second best node from n, the third best, etc. For this problem, we propose a

more general job, $J(n, C(n))$, which finds the best move among all the moves excluding those in the list $C(n)$, where $C(n)$ is a list of prohibited moves. Thus, for n, we can use $J(n, \emptyset)$ to find the best move n_1, and then use $J(n, \{n_1\})$ to find the second best move n_2, and so on.

Chapter 4 Solving Games Using JLVC

JLVC uses connection model to solve computer game problems efficiently. In this chapter, we demonstrate JLVC by solving Connect6 openings. In the selection, pre-update, and update phases, we follow the rule of proof number search to select a node and update nodes. In the execution phase, we use the Connect6 program, NCTU6 [31][64][68], to evaluate and expand a node. In this chapter, we will also introduce our method to expand nodes, called *postponed sibling generation*, and introduce many methods to be used in the pre-update phase.

This chapter is organized as follows. Section 4.1 reviews the background including PNS, Connect6, and the program NCTU6. Section 4.2 describes our algorithm. Section 4.3 does experiments. Section 4.4 discusses some related work and some miscellaneous issues for our algorithm, such as overhead. Section 4.5 makes concluding remarks.

4.1 Background

The proof number search (PNS) is reviewed in Subsection 4.1.1, and Connect6 and NCTU6 are described in Subsection 4.1.2.

4.1.1 Proof Number Search

Proof number search (PNS), proposed by Allis *et al*. [2][4], is a kind of best-first search algorithm that was successfully used to prove or solve theoretical values [21] of game positions for many games [2][3][4][22][44][46][47][58], such as Connect-Four, Gomoku, Renju, Checkers, Lines of Action, Go, and Shogi. Like most best-first searches,

PNS has a well-known disadvantage, the requirement of maintaining the whole search tree in memory. As a result, many variations [8][27][36][38][47][58] have been proposed to avoid this problem, such as PN2, DFPN, PN*, PDS, and parallel PNS [26][44]. For example, PN2 used two-level PNS to reduce the size of the maintained search tree.

As described in [2][4], PNS is based on an AND/OR search tree where each node *n* is associated with proof/disproof numbers, $p(n)$ and $d(n)$, which represent the minimum numbers of nodes to be expanded to prove/disprove n. Basically, all leaves' $p(n)/d(n)$ are initialized to $1/1$. The values $p(n)/d(n)$ are $0/\infty$ if the node n is proved, and $\infty/0$ if it is disproved. PNS repeatedly chooses a leaf called *the most-proving node* (*MPN*) to expand, until the root is proved or disproved. The details of choosing MPN and maintaining the proof/disproof numbers can be found in [2][4].

An important property related to MPN is: if the selected MPN is proved (disproved), the proof (disproof) number of the search tree decreases by one. The property, called *MPN Property* in this thesis, can be generalized as follows.

- If the selected MPN is proved (disproved), the proof (disproof) number of the node, whose subtree includes the MPN, decreases by one, and the disproof (proof) number of it remains the same or increases.

4.1.2 Connect6 and NCTU6

Connect6 [61][62] is a kind of six-in-a-row game that was introduced by Wu *et al*. Two players, named Black and White, alternately play one move by placing two black and white stones respectively on empty intersections of a Go board (a 19×19 board) in each turn. Black plays first and places one stone initially. The winner is the first to get six consecutive stones of his own horizontally, vertically or diagonally.

NCTU6 is a Connect6 program, written by Wu *et al.* This thesis reviews the results

from [56][65] as follows. NCTU6 included a solver that was able to find Victory by Continuous Four (VCF), a common term for winning strategies in the Renju community. More specifically, VCF for Connect6, also called VCST, wins by making continuous moves with at least one four (a threat which causes the opponent to defend) and ends with connecting up to six in all subsequent variations.

From the viewpoint of *lambda search*, VCF or VCST is a winning strategy in the second order of threats according to the definition in [65], that is, a Λ_a^2-tree (similar to a λ_a^2-tree in [56]) with value 1. Lambda search, as defined by Thomsen, is a kind of threat-based search method, formalized to express different orders of threats. Wu and Lin modified the definition to fit Connect6 as well as a family of k-in-a-row games and changed the notation from λ_a^i to Λ_a^i.

NCTU6-Verifier (abbr. Verifier) is a verifier modified from NCTU6 by incorporating a lambda-based threat-space search, and used to verify whether the player to move loses in the position, or to list all the defensive moves that may prevent the player from losing in the order Λ_a^2. If no moves are listed from a position, Verifier is able to prove that the position is a loss. If some moves are listed, Verifier is able to prove that those not listed are losses. In some extreme cases, Verifier may report up to tens of thousands of moves.

One issue for Connect6 is that the game lacks openings for players, since the game is still young when compared with other games such as chess, Chinese chess and Go. Hence, it is important for the Connect6 player community to investigate more openings quickly. For this issue, Wu *et al.* in [59] designed a desktop grid, like the job-level system, to help human experts build and solve openings.

In the earliest version of the grid, both NCTU6 and Verifier were the two jobs used, and a game record editor environment was utilized to allow users to select and dispatch jobs to free workers. NCTU6 was used to find the best move from the current game position,

while Verifier was used to expand all the nodes (namely for all the defensive moves). This environment helped human experts build and solve openings manually.

In this thesis, the system is modified to support a job-level system where job-level search can be used to create and perform jobs automatically. Both NCTU6 and Verifier are supported as jobs. NCTU6 jobs take tens of seconds on the average (statistics are given in Section 4.4.3), and Verifier jobs take a wide variety of times, from one minute up to one day, depending on the number of defensive moves. As above, in some extreme cases, Verifier may generate a large number of moves in Job-Level Search, which is resource-consuming for both computation and memory resources. Thus, Verifier is less feasible in practice.

In order to solve this problem, NCTU6 is modified to support the following two additional functionalities:

1. Support $J(n, C(n))$. Given a position n and a list of prohibited moves $C(n)$ as input, NCTU6 generates the best move among all the moves outside the list. As described in Section 3.3, this can be used to find the best move of a position, the second best, …, etc.
2. For each job $J(n, C(n))$, report a sure loss in the job result, if none of the non-prohibited moves can prevent a loss.

Supporting the first functionality, the modified NCTU6 can be used to find the best move of a position, the second best, …, etc., as described in Section 3.3. Supporting the second functionality, all the moves can be expanded like Verifier. Thus, NCTU6 is able to replace Verifier with Job-Level Search.

4.2 Job-Level Proof Number Search

This section presents job-level proof number search (JL-PNS) and demonstrates it by

using NCTU6, a Connect6 program, to solve Connect6 positions on JLVC automatically. JL-PNS uses PNS (described in Subsection 4.1.1) to maintain a search tree in the client, and runs in four phases following the generic job-level search, described in Section 3.3.

In the selection phase, MPN is selected, and jobs are created from MPN for execution on workers in the execution phase. In Subsection 4.2.1, we propose a method, called postponed sibling generation, to create jobs. In the update phase, the move in the job result is used to generate the corresponding new node, and the evaluated value of the move is used to initialize the proof/disproof numbers of the node and to update others in the search tree, described in Subsection 4.2.2. In the pre-update phase, several policies are proposed and described in Subsection 4.2.3.

4.2.1 Proof/Disproof Number Initialization

This subsection briefly describes how to apply the domain knowledge given by NCTU6 to initialization of the proof/disproof numbers. Since it normally takes tens of seconds to execute an NCTU6 job, it becomes critical to choose carefully a *good* MPN to expand, especially when there are many candidates with 1/1 as the standard initialization. In [2], Allis suggested several methods such as the use of the number of nodes to be expanded, the number of moves to the end of games, or the depth of a node.

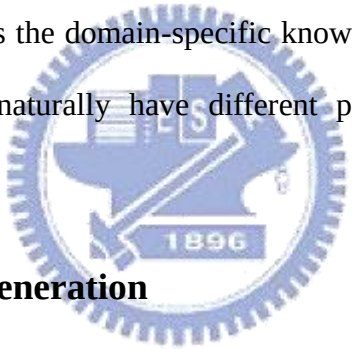
Status	Bw	B4	B3	B2	B1	stable	unstable2
p(n)/d(n)	0/∞	1/18	2/12	3/10	4/8	6/6	4/4
Status	Ww	W4	W3	W2	W1	unstable1	
p(n)/d(n)	∞/0	18/1	12/2	10/3	8/4	5/5	

Table 7: Game Status and the corresponding initializations.

Our approach is simply to trust NCTU6 and use its evaluations on nodes (positions) to initialize the proof/disproof numbers in JL-PNS as shown in Table 7. The status Bw indicates that Black has a sure win, so the proof/disproof numbers of a node with Bw are

$0/\infty$. For simplicity of discussion, this thesis looks to prove a game when Black wins, unless explicitly specified. The statuses B1 to B4 indicate that the game favors Black with different levels of win probability, where B1 indicates to favor Black least probability and B4 the most (i.e. Black has a very good chance of a win in B4) according to the evaluation by NCTU6. Similarly, the statuses W* are for White. The status 'stable' indicates that the game is stable for both players, while both 'unstable1' and 'unstable2' indicate unstable, where unstable2 is more unstable than unstable1. Proof/disproof numbers of these unstable statuses are smaller than those of "stable", since it is assumed to be more likely to prove or disprove "unstable" positions.

Of course, there are many different kinds of initializations other than those in Table 1. Our philosophy is simply to pass the domain-specific knowledge from NCTU6 to JL-PNS. Different programs or games naturally have different policies on initializations from practical experiences.



4.2.2 Postponed Sibling Generation

In this subsection, we describe how to create jobs after an MPN n is selected. Straightforwardly from PNS, the node n is expanded and all of its children are generated. Unfortunately, in Connect6, the number of children is up to tens of thousands of nodes usually. If we use Verifier to help remove some losing moves, it may still take a huge amount of computation time as described in Subsection 4.1.2. Thus, it becomes more efficient and effective to generate a node at a time. However, in PNS, the MPN is a leaf in the search tree. If we always generate the best move from the MPN, then there are no choices to generate the second best move, the third best, …, etc. In order to solve this problem, we propose a method called *postponed sibling generation* as follows.

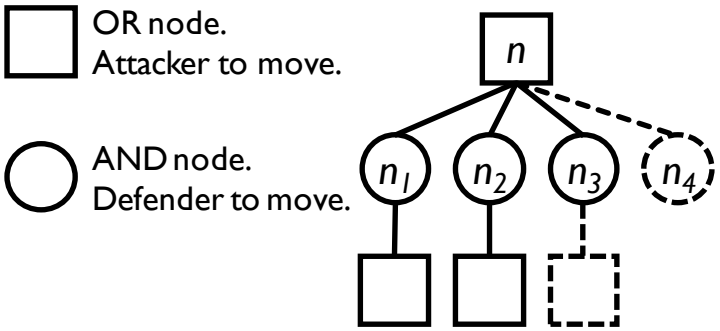


Figure 21: Expanding n_3 and n (to generate n_4) simultaneously.

- Assume that for a node n, the i-th move n_i is already generated, but the (i+1) n_{i+1} is not yet. When the node n_i is chosen as the MPN for expansion, generate the best move of n_i by $J(n_i, \emptyset)$ and generate n_{i+1} by $J(n, \{n_1, \ldots n_i\})$ simultaneously. The example in Figure 21 illustrates this. Assume that the node n_3 is chosen as the MPN. Then, generate the best move of n_3 by $J(n_3, \emptyset)$ and generate n_4 by $J(n, \{n_1, n_2, n_3\})$ simultaneously. On the other hand, if the branch n_1 or n_2 is chosen, do not generate n_4 as yet.
- In an Attacker to Move node, assume that a generated move is reported to be a sure loss to the Attacker. Then, generate no more moves from the node, since others are also sure losses as per the second functionality described in Subsection 4.1.2. For example, in Figure 21, assume that $J(n, \{n_1, n_2, n_3\})$ reports a sure loss when generating n_4. From the second functionality, all the moves except for n_1, n_2 and n_3 are sure losses. Thus, it is no longer necessary to expand the node n. In this case, all children of n are generated and n_4 behaves as a *stopper*. Note that it is similar in the case that the node n is an AND node.

Since NCTU6 supports $J(n, C(n))$ and is able to report a sure loss as described in Subsection 4.1.2, NCTU6 can support postponed sibling generation.

As shown in Figure 21, postponed sibling generation fits parallelism well, since

generating n_4 and expanding n_3 can both be performed simultaneously. Some further issues are described as follows.

One may ask, what if we choose to generate n_4 before expanding n_3? Assume that one player, say the Attacker, is to move in the OR node n. As per the first additional functionality described in Subsection 4.1.2, the move n_3 is assumed to be *better* for the Attacker than n_4 *according to the evaluation of NCTU6*. In this case, the condition $p(n_3) \leq p(n_4)$ holds. Thus, the node n_3 must be chosen as the MPN to expand earlier than n_4. It therefore becomes insignificant to generate n_4 before expanding n_3. In addition, the above condition also implies that the proof numbers of all the ancestors of node n remain unchanged. As for the disproof numbers of all the ancestors of n, these values are the same or higher. Unfortunately, higher disproof numbers discourage the JL-PNS from choosing n_3 as MPNs to expand. Thus, the behavior becomes awkward, especially if the node n_3 will be proved eventually.

One may also ask what if we expand n_3, but generate n_4 later? In such a case, it may make the proof number of n fluctuate. An extreme situation would be that the value becomes infinity when all nodes, n_1, n_2 and n_3, are disproved, but n is not disproved since n_4 is not disproved yet.

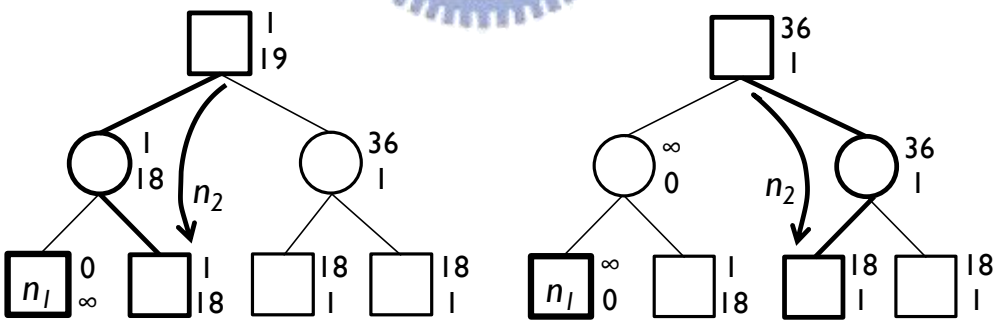
4.2.3 Policies in the Pre-Update Phase

In this subsection, several policies are proposed for the updates in the pre-update phase. As described in Section 3.3, when more workers in the job-level system are available, more MPNs will be selected for execution on these workers. If we do not change the proof/disproof numbers of the chosen MPNs being expanded, named the *active MPNs*, we would obviously choose the same node. Therefore, pre-updates are needed to select other nodes as MPNs.

An important goal of choosing multiple MPNs is that these chosen MPNs are also chosen eventually in the case there are no multiple workers, that is, when only one MPN is chosen at a time. Note that the policy without any pre-update is called the *native policy* in the rest of this thesis. Some new policies are introduced and proposed in the subsequent subsections.

4.2.3.1. Virtual-Win, Virtual-Loss, and Greedy

In this subsection, we introduce the simplest policies. One policy, used to prevent choosing the same node twice, named the *virtual-win policy* (abbr. *VW policy*), assumes a virtual win [12] on the active MPNs. The idea of the virtual-win policy is to assume that the active MPNs are all proved. Thus, their proof/disproof numbers are all set to $0/\infty$, as illustrated in Figure 22 (a). When the proof number of the root is zero, the choosing of more MPNs is stopped. The reason is that the root is already proved if the active MPNs are all proved.



(a) (b)

Figure 22: (a) Virtual win policy. (b) Virtual loss policy.

In contrast, another policy, named the *virtual-loss policy* (abbr. *VL policy*), is to assume a virtual loss on the active MPNs. Thus, the proof/disproof numbers of these nodes are set to $\infty/0$ as shown in Figure 22 (b). Similarly, when the disproof number of the root is zero,

we stop choosing more MPNs. Similarly, the root is disproved, if all the actives are disproved.

Another introduced policy, named a *greedy policy* (abbr. *GD policy*), chooses VW policy when the chosen nodes favor a win according to the evaluation of NCTU6, and chooses VL policy otherwise as we may not always be able to decide a winner in advance, as in cases such as the one in Figure 32 (f). The pseudo code for these policies is shown below. The function UpdateAncestors updates the proof/disproof numbers of all the ancestors of the given node n in PNS.

```
Policy VirtualWin(n)
  1:    n.pn = 0; n.dn = ∞;
  2:    UpdateAncestors(n);
end policy

Policy VirtualLoss(n)
  1:    n.pn = ∞; n.dn = 0;
  2:    UpdateAncestors(n);
end policy
```

```
Policy Greedy(n)
  1:    if n.pn ≤ n.dn then
  2:        n.pn = 0; n.dn = ∞;
  3:    else
  4:        n.pn = ∞; n.dn = 0;
  5:    end if
  6:    UpdateAncestors(n);
end policy
```

Figure 23: The pseudo code for VW, VL, and GD.

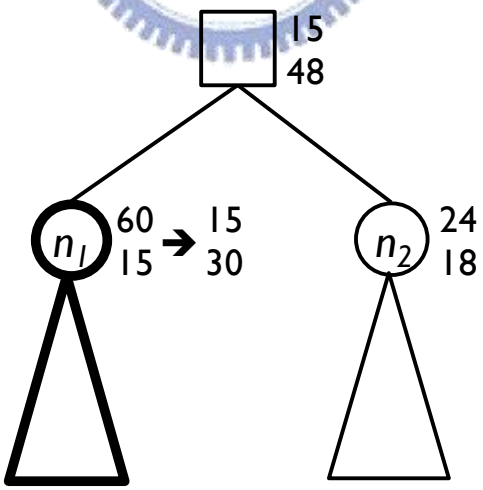


Figure 24: A starvation example for virtual-win policy.

These policies may cause possible fluctuation. From our observation, fluctuation for these policies may result in starvation, as illustrated by the example of VW policy, shown in Figure 24. In this example, workers are running the jobs for some nodes under the subtree

rooted at n_1. Let $p(n_1)/d(n_1)$ be 60/15 for the native policy, and 15/30 for the virtual-win policy. Also, let $p(n_2)/d(n_2)$ be 24/18 for both policies, since no jobs inside the subtree are rooted at n_2.

Now, when a new worker is available, an MPN is chosen for execution. To locate the MPN, the branch to node n_1 is chosen for the virtual-win policy, since $p(n_1) < p(n_2)$. However, for the virtual-win policy, the proof number $p(n_1)$ becomes smaller and the disproof number $d(n_1)$ remains the same or becomes higher according to MPN Property. Subsequently, available workers will continue to choose n_1, as long as jobs remain unfinished. Even if some jobs do finish, the subtree rooted at n_1 will still be chosen as long as $p(n_1)$ remains less than $p(n_2)$. Hence, the node n_2 may starve.

The phenomenon of starvation may also happen in both VL and GD policies. In the following subsections, further policies are proposed to avoid the above problem.

4.2.3.2. Flag

A simple policy [43] to avoid the above starvation problem, named the *flag policy* (abbr. *FG policy*), is to use a flag mechanism. In this policy, all the MPNs being chosen to generate the first child (like n_3 in Figure 21) are flagged. Let nodes be called *partially-flagged nodes*, if some of their children are flagged, but others are not, and called *fully-flagged nodes*, if all of their children are flagged. Fully-flagged nodes are also flagged recursively. The pseudo code for FG policy is as follows.

```
Policy Flag(n)
 1:    n.flag = 1;
 2:    if all siblings are flagged then
 3:        Flag( n.parent );
 4:    end if
end policy
```

Figure 25: The pseudo code for FG.

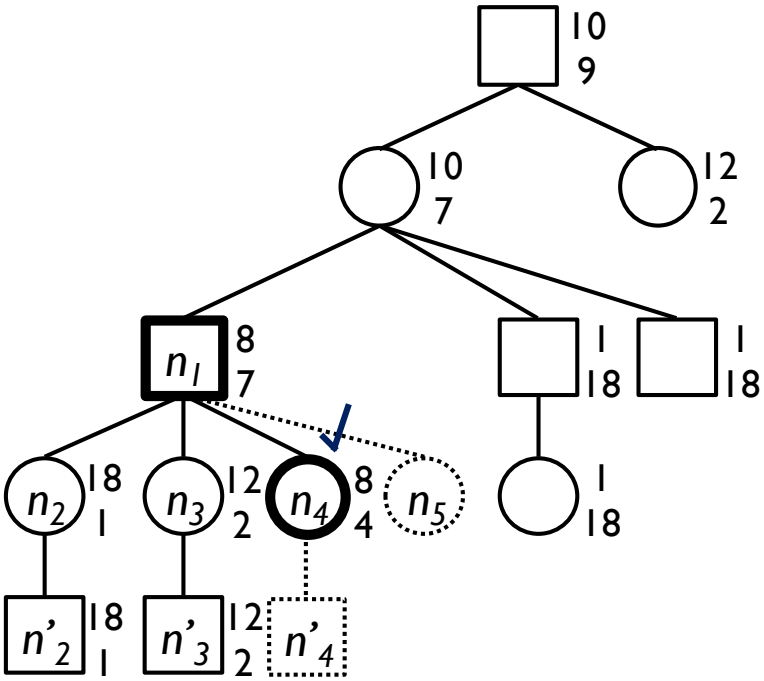


Figure 26: An example FG policy.

For choosing MPNs, the policy follows the native policy in principle, whilst avoiding choosing the nodes with flags. Namely, when a chosen node is flagged, another non-flagged node, with the smallest proof/disproof numbers, is chosen instead. An example is illustrated in Figure 26. In the policy, the next MPN to choose is n'_3, since the branch to go from n_1 is n_3.

4.2.3.3. Modified Flag

Although the FG policy can solve the problem of starvation, the example in Figure 26 shows another potential problem. The node n_1 and all of its ancestors *think* $p(n_1)$ as well as $p(n_3)$ should be 8. However, the actual value of $p(n_3)$ is 12. In the case that $p(n_3)$ is much larger, the problem is even more serious. Thus, the policy may lead to the choosing of the wrong MPNs, in this case, n_3.

To solve the above problem, we modify the above policy into a new one, named

Modified-Flag policy (abbr. *MF policy*). The MF policy is as follows. For a partially-flagged node, say it is an OR node for simplicity of discussion, its proof number is the minimal proof numbers of non-flagged children. And for a fully-flagged node, its proof number is the maximum proof number of all flagged children. The pseudo code for MF policy is as follows.

```
Policy ModifiedFlag(n)
 1:    Flag(n);
 2:    while (n.parent) do
 3:       n = n.parent;
 4:       if n is an OR node then
 5:          n.dn = sum(c.dn) for all children c;
 6:          if n has non-flagged children then
 7:             n.pn = min(c.pn) for all non-flagged children c;
 8:          else
 9:             n.pn = max(c.pn) for all flagged children c;
10:          end if
11:       else
12:          //omitted
13:       end if
14:    end while
end policy
```

Figure 27: The pseudo code for MF.

For a fully-flagged node, we set its proof number to the maximum proof number among children, instead of the minimum one. The reasoning behind this is illustrated by the example in Figure 26. Assume that for the node n_1, the two children, n_3 and n_4, are flagged and the child n_2 is not flagged yet. The value $p(n_1)$ is 18. Now, we look to select one more MPN from n_1. The node n'_2 is then selected. According to FG policy, where the proof number is set to the minimum, the value $p(n_1)$ then drops to 8. This implies that the next MPN selection will be attracted towards the node n_1. This is clearly awkward. In the case that we set the proof number to the maximum proof number among children, the value $p(n_1)$ remains 18. Thus, this policy does not wrongly direct the MPN selection.

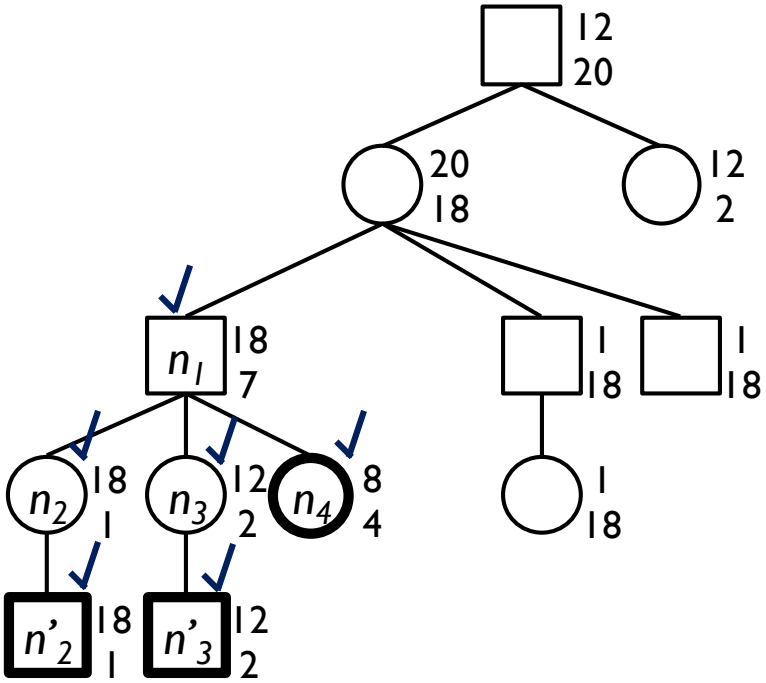


Figure 28: Assign the maximal proof numbers of children for fully-flagged nodes.

Figure 28 shows the proof/disproof numbers of search trees in Figure 26 in MF policy. As for disproof numbers (in OR nodes) for the above case, we still follow the PNS to sum up the disproof numbers of all the children, regardless of whether they are flagged or not. For example, in both Figure 26 and Figure 28, $d(n_1)$ is 7.

4.2.3.4. Virtual-Equivalence

Virtual equivalence is an idea based on the assumption that the generated node is expected to have almost the same proof/disproof numbers as its parent, if the generated node is the eldest child, or as the youngest elder sibling, otherwise. The pseudo code for this policy is as follows.

```
Policy VirtualEquivalence(n)
 1:    if n has sibling then
 2:        set s to the youngest elder sibling;
 3:    else
 4:        set s to the parent;
 5:    end if
 6:    n.pn = s.pn;
 7:    n.dn = s.dn;
 8:    UpdateAncestors(n);
end policy
```

Figure 29: The pseudo code for VE.

The following two cases are discussed for this policy. First, assume that a node n has no child yet. Then, when a program like NCTU6 is used to generate from n (regardless of AND/OR node) the first node n_1, the best move from n, it is expected that the calculated value which is used to initialize the proof/disproof values of n_1 is the same as or close to that for n, based on the assumption that the program is accurate enough.

Second, assume that a node n has some children, say three, n_1, n_2, and n_3, generated based on the scheme of the postponed sibling generations. As per the argument in the postponed sibling generations, the three children stand for the best, the second best and the third best children of the node n, respectively. Now, when the program is to generate the fourth child n_4, that is, the fourth best child, it is expected that the calculated value for n_4 is the same as or close to that for n_3.

In fact, the FG policy can be viewed as a kind of the first case. For example, in Figure 26, the generation of a new node n'_4 is assumed, whose proof/disproof numbers of n'_4 are the same as those of n_4.

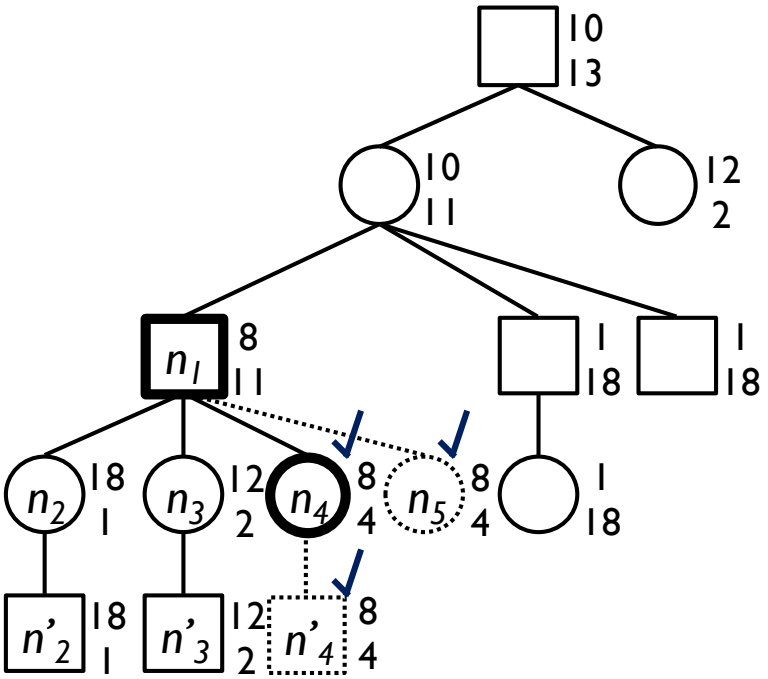


Figure 30: The search tree in Figure 28 with FG-VE policy.

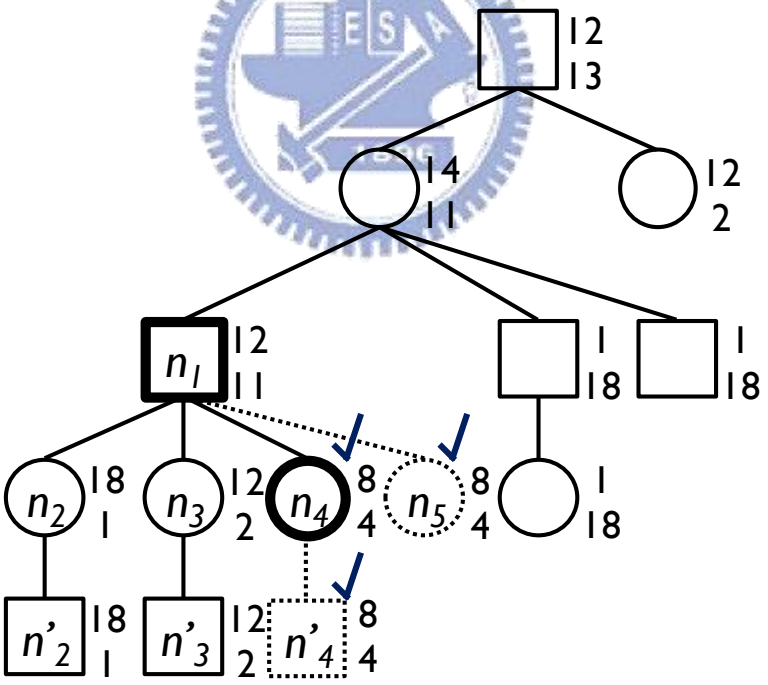


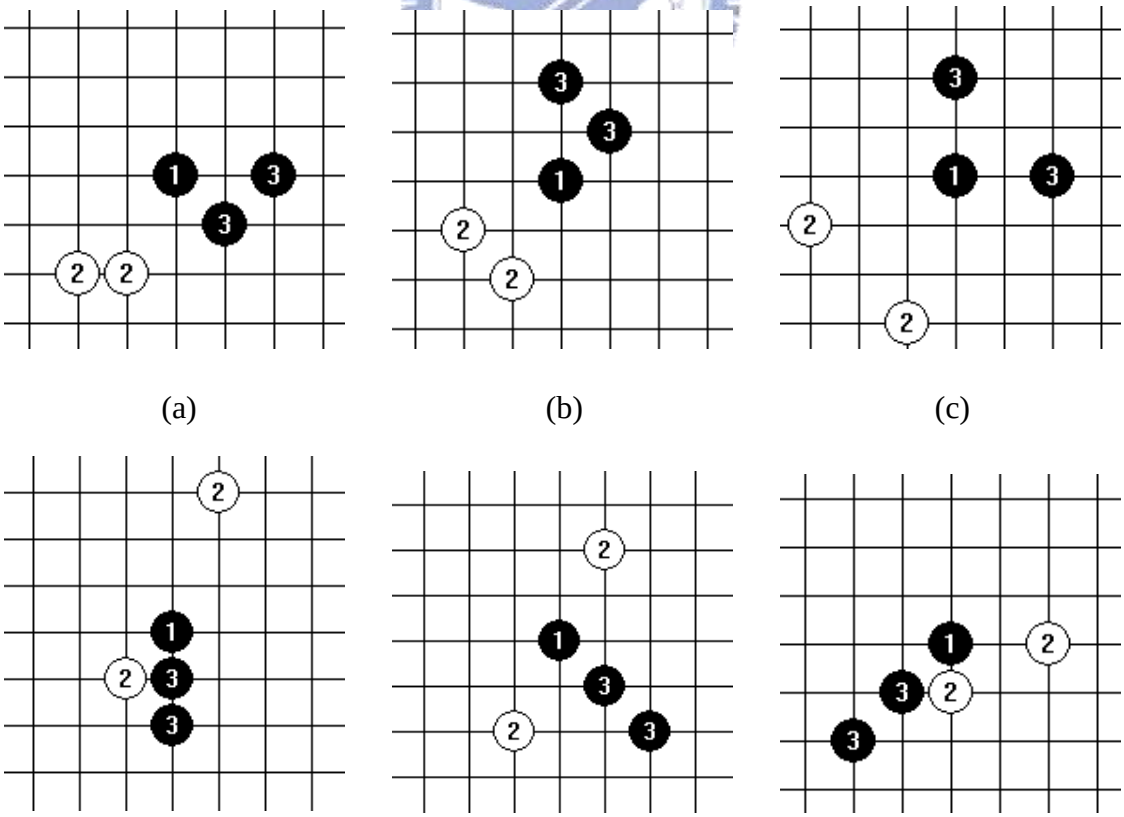
Figure 31: The search tree in Figure 28 with MF-VE policy.

Now, let us investigate the second case. For this argument, we look to generate a new child whose proof/disproof numbers are the same as those of its youngest elder sibling, i.e., for the example in Figure 26, we set the proof/disproof numbers of n_5 to those of node n_4.

Based on the discussion above, both FG and MF policies can be modified into *Flag-with-Virtual-Equivalence policy* (abbr. *FG-VE policy*) and *Modified-Flag-with-Virtual-Equivalence policy* (abbr. *MF-VE policy*), respectively. Both Figure 30 and Figure 31 show proof/disproof numbers of the PNS tree in Figure 28 for both FG-VE and MF-VE policies respectively.

4.3 Experiments

In our experiments, our job-level system is maintained on a desktop grid [59] with 8 workers, Intel Core2 Duo 3.33 GHz machine. Since each worker has two cores, the desktop has actually 16 cores in total. And, the client was located on another host. Note that the time for maintaining the JL-PNS tree in the client is negligible, since it is relatively low when compared with that for NCTU6.



(a) (b) (c)

(d) (e) (f)

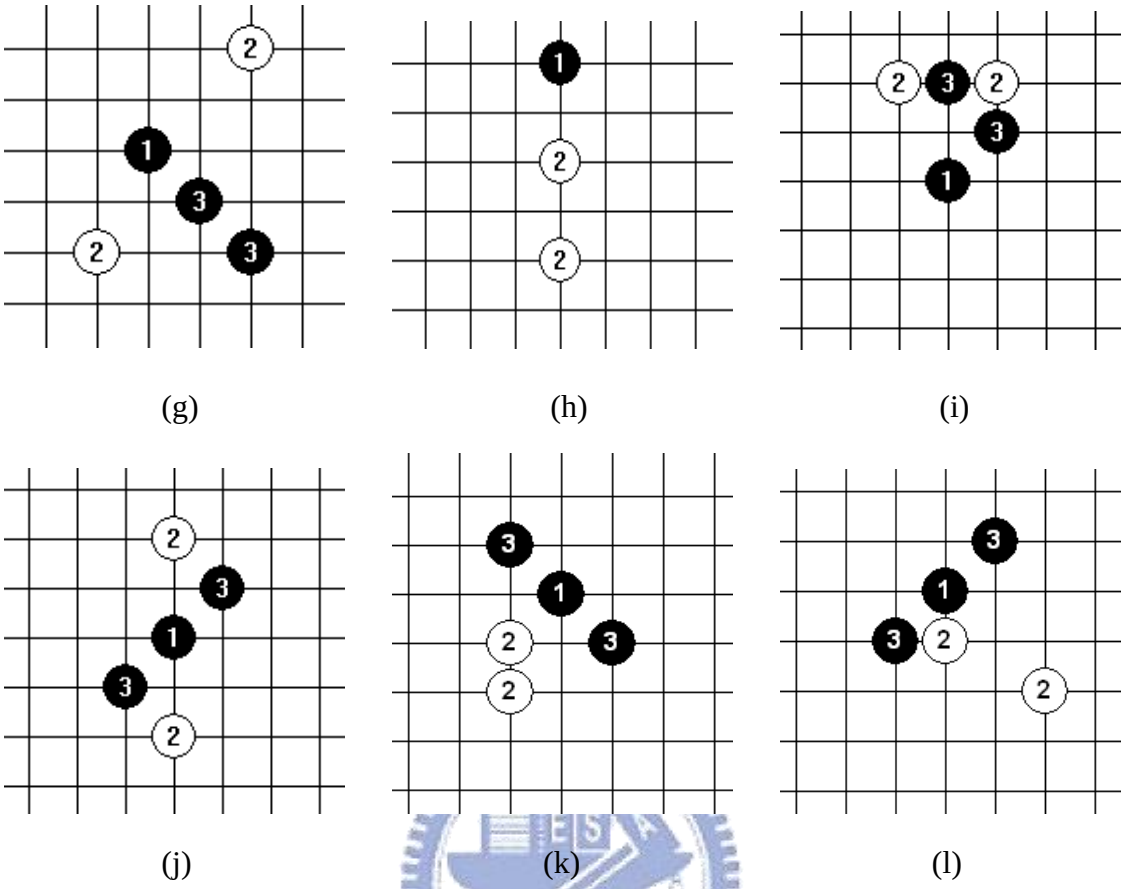


Figure 32: The twelve solved openings.

In our experiments of JL-PNS, the benchmark included 35 Connect6 positions, among which the last 15 positions are won by the player to move, while the first 20 are won by the other. The first 20 and the last 15 are ordered according to their computation times on one core.

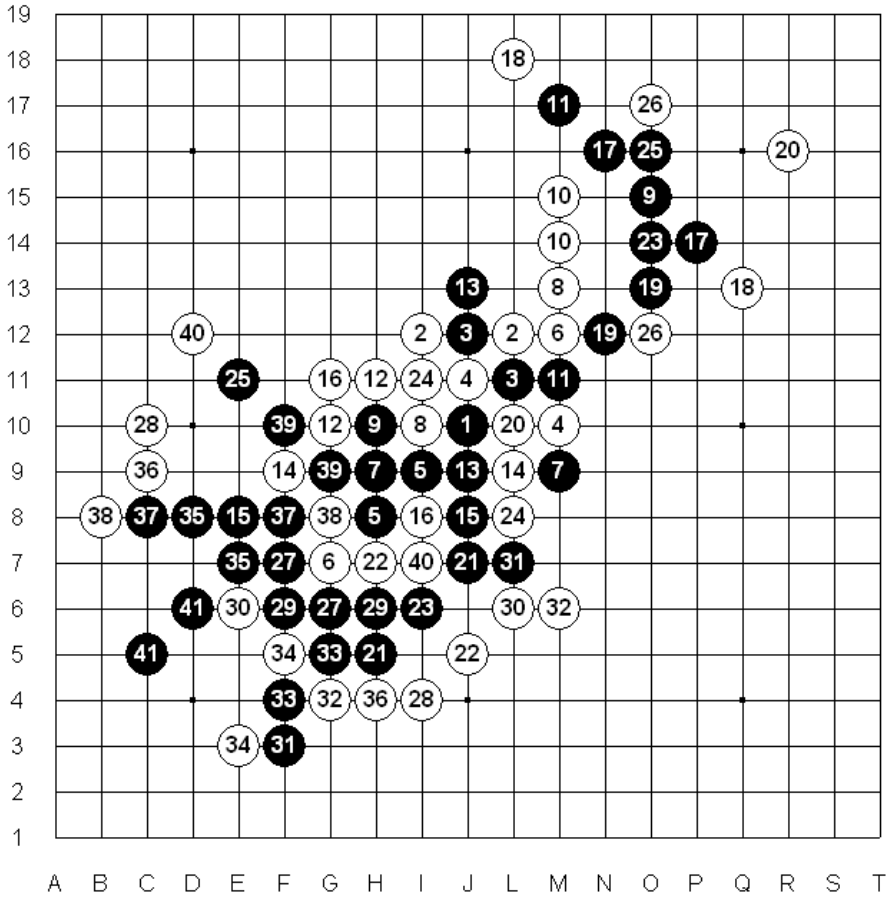


Figure 33: A path in the winning tree of the Mickey-Mouse Opening.

Among the 35 positions, ten are 3-move openings shown in Figure 32 (a) to (j). For many of them, their winning strategies had not been found before our work. In particular, the Mickey-Mouse Opening (Figure 32 (i)) had been one of the most popular openings before we solved it. Figure 33 shows a path in the winning tree. And the tenth one (Figure 32 (j)), also called Straight Opening, is another difficult one.

According to our statistics on running the 35 positions, each NCTU6 job takes about 37.45 seconds on average. About 21.10% of the jobs are run over one minute. About 14.99% of jobs are returned with wins/losses, and these jobs are usually run quickly. If these jobs are not counted, each NCTU6 job takes about 41.38 seconds on average. In addition, 11.19% extra jobs are aborted.

In this section, for performance analysis, let speedup S_k be T_1/T_k, where T_k is the computation time for solving a position with k cores. Also, let efficiency E_k be S_k/k. The efficiencies are one for ideal linear speedups.

This section is organized as follows. Subsection 4.3.1 details the experiments for our benchmark, comparing all the polices mentioned in Subsection 4.2.3. The results show that the four policies, FG, MF, FG-VE and MF-VE, are clearly better than the other three. Subsection 4.3.2 discusses the accuracy of VE (virtual-equivalence) by showing status correlations between nodes and their parents or sibling nodes. Then, we further analyze the experimental results of the four policies in Subsections 4.3.3. In Subsection 4.3.4, we analyze the performances for the positions requiring more computation times.

4.3.1 Experiments for Benchmark

In this subsection, the experiments were done for our benchmark to investigate all the policies mentioned in Subsection 4.2.3. For each of these policies, the computation times with 1, 2, 4, 8, and 16 cores for each Connect6 position are measured.

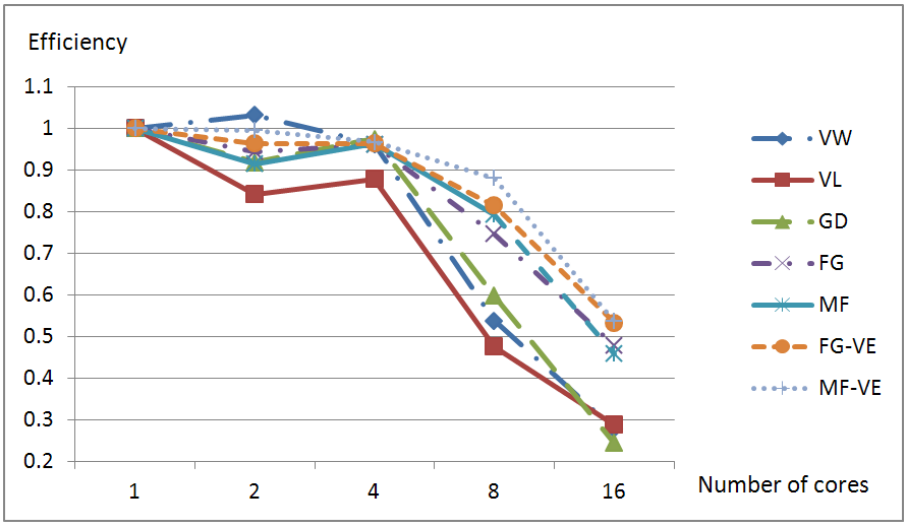


Figure 34: The efficiencies for all 35 positions for each policy.

In order to have a quick comparison and performance analysis, we compared the

efficiencies of the 35 Connect6 positions, for each policy and for each number of cores, as shown in Figure 34. Note that all the one-core performance results for the different policies are the same since there are no differences in choosing nodes on a single core for different policies.

From Figure 34, the four policies with flag mechanism, FG, MF, FG-VE and MF-VE, all outperformed the other three without flag mechanism, VW, VL and GD. For example, the computation times for the VW, VL, or GD with 16 cores were about 80% longer than those of MF-VE. From our observation, we did find some cases with starvation phenomenon, as mentioned in Subsection 4.

The performances for the policies with flag mechanism are close and will be discussed in more detail in Subsection 4.3.3. Before discussing these, we give an analysis of virtual-equivalence in Subsection 4.3.2.

4.3.2 The Analysis for Virtual-Equivalence

In Subsection 4.2.3.4, the concept of virtual-equivalence is: the generated node is expected to have almost the same proof/disproof numbers as its parent, and as the youngest elder sibling. In this subsection, our experiments are designed to test how close they are. For example, how close are the $p(n)/d(n)$ of the two generating nodes, n_4 and n'_4, and/or n_4 and n_5 in Figure 31? To assess this, we measure the distances between n_4 and n'_4, and between n_4 and n_5.

Status	B:w	B4	B3	B2	B1	stable
v(status)	0	1	2	3	4	5
Status	W:w	W4	W3	W2	W1	unstable
v(status)	10	9	8	7	6	*

Table 8: Assign a value for each status.

For the measurement, we assign a value to each status as shown in Table 8 and

calculate the distances. For example, in Figure 31, the distance between n_3 and n_4 is 8 minus 6, since W3 is 8 and W1 is 6. Notably, for "unstable" positions, since they are hard to locate, we simply ignore the distances with the unstable status. The procedure of counting the distance is as follows.

```
Procedure Count Distance(n)
 1:    if n is eldest child then
 2:        p = parent of n;
 3:        par_dist += v(n.status)-v(p.status);
 4:    else
 5:        s = youngest elder sibling of n;
 6:        sib_dist += v(n.status)-v(s.status);
 7:    end if
end
```

Figure 35: The pseudo code of counting the distance.

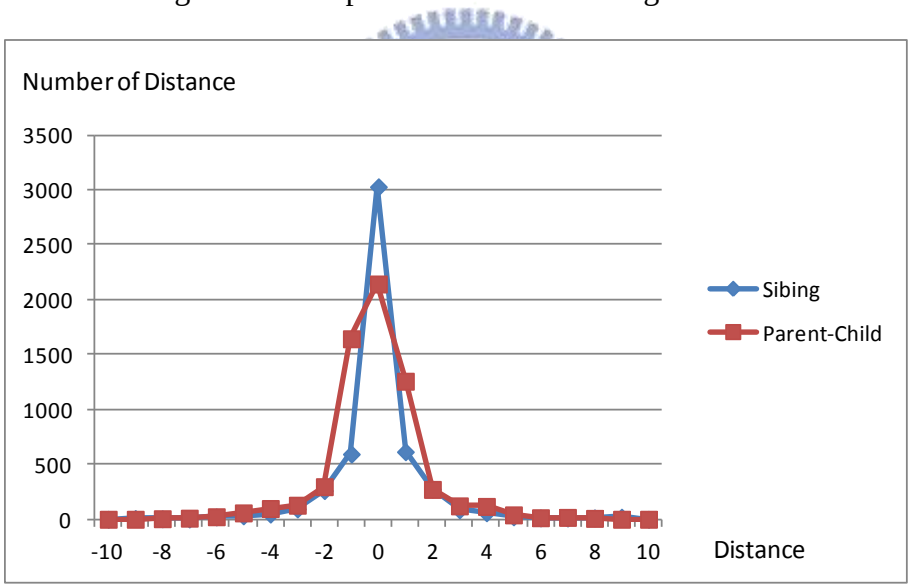


Figure 36: Illustrates the measurement of each distance.

For all nodes generated in solving the 35 positions, we show the statistics for the distances between neighboring siblings and between parents and the eldest children in Figure 36. As seen in the figure, most generated nodes have the same status (the distances are 0) as the eldest child and as the eldest younger sibling. According to this result, it is expected that the proof/disproof numbers will be less fluctuated. Thus, it becomes more

likely that the chosen MPNs will also be chosen in the single core version.

From Figure 36, we also observe that the distances between parents and the eldest children are, in general, larger than those between siblings. The reason for this is similar to the two-ply update issue, mentioned in [67]. Since a parent has two less stones than its children in Connect6, it is harder to evaluate them consistently.

4.3.3 Flag Mechanism

This subsection analyzes the performances of the policies with flag mechanism in more detail. Figure 37 (below) shows the ratios of the performances of different versions with respect to those of the FG. In this figure, we observe that the MF-VE policy performed best and outperformed FG by about 17.8% and 12.3% for 8 cores and 16 cores, respectively.

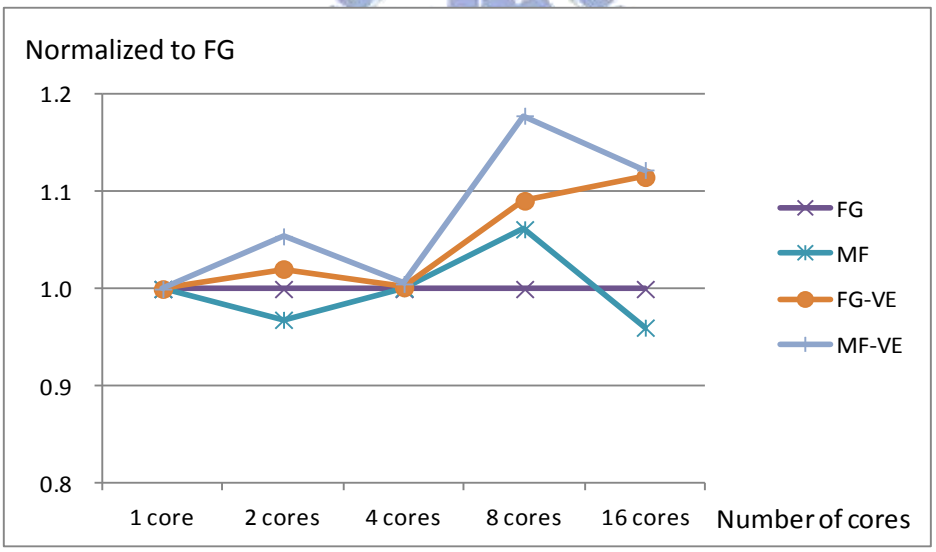


Figure 37: The speedups relative to FG policy for solving 35 Connect6 positions with different policies.

Figure 37 also shows that the performances of both FG-VE and MF-VE are better than those with MF and FG. Both FG-VE and MF-VE use sibling VE while the other two do not. This indicates that the policies with sibling VE perform better.

4.3.4 Experiments for Difficult Positions

In this subsection, we analyze performance for the positions requiring more computation. For this purpose, we chose 15 of the most difficult positions amongst the 35, and analyzed the improvements from FG to MF-VE using 16 cores, as shown in Figure 38 (below). Note that the positions in Figure 38 are ordered according to the computation time, with the rightmost one requiring the most time. This is about 2.75 hours for 16 cores. From this figure, we observe that MF-VE generally performed slightly better than FG.

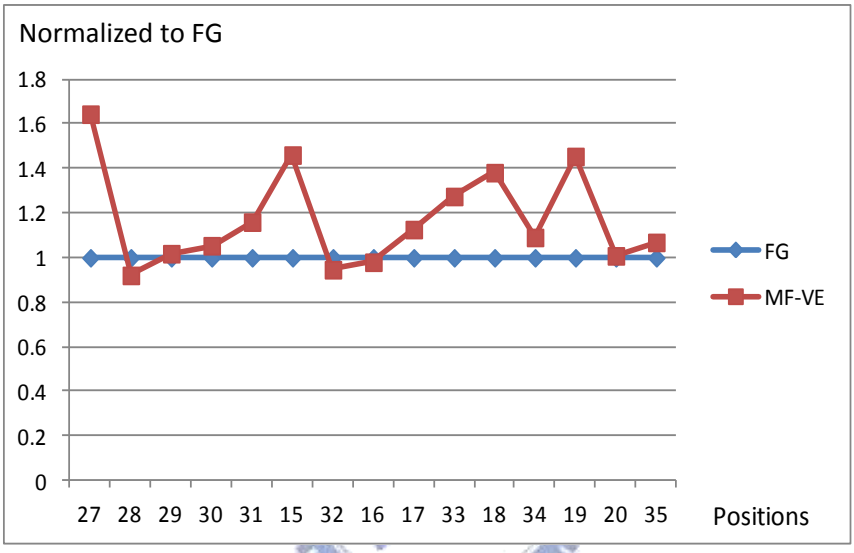


Figure 38: The improvement of speedup for the most difficult 15 positions from FG to MF-VE.

Thereafter, we investigated some other positions requiring even more computation times. We solved two more openings, as shown in Figure 32 (k) and (l). These required significantly more computation time, about 7.03 and 35.11 hours for 16 cores, respectively. Since much more time was spent in solving the two openings, we only compared three policies, VL, FG, and MF-VE by running them on 16 cores. The computation times are shown in Figure 39.

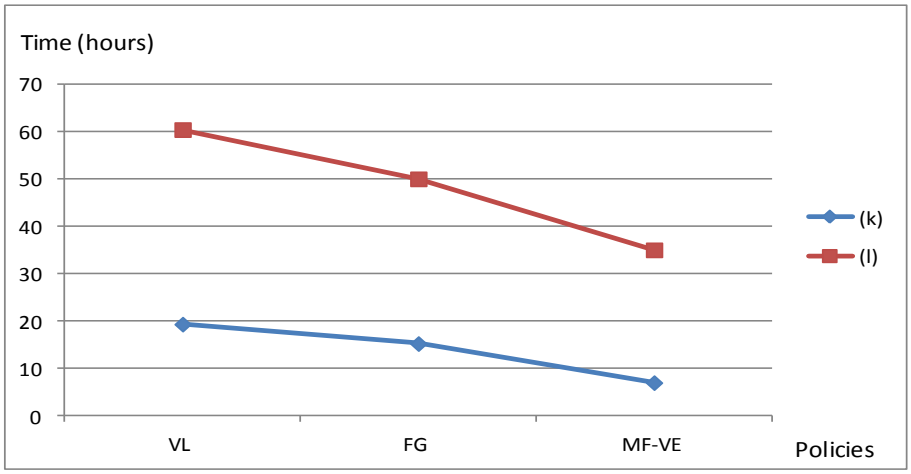


Figure 39: The solving times for the three versions of each position on 16 cores.

As seen in Figure 39 MF-VE performed better than FG by factors of 2.18 and 1.43 for the positions in Figure 32 (k) and (l), respectively. Moreover, MF-VE performed better than VL by 2.76 and 1.72 for Figure 32 (k) and (l), respectively. The results demonstrate that MF-VE also outperforms FG in bigger cases.

4.4 Discussion

In this section, we first discuss some past job-level-like research work, and then some issues about job-level computation in the implementations.

4.4.1 Past Job-level-like Work

The research into solving Checkers [46] was separated into two parts: the proof-tree manager and the proof solvers. The proof-tree manager, like the client in our model, used the PNS to identify a prioritized list of positions to be examined by the proof solvers, like jobs in our model. Their manager generated several hundred positions at a time to keep workers busy, and they did not consider pre-update.

The research [11] proposed a meta MCTS to build opening s in the book of Go. In their method, a tree policy was used to select a node in the UCT tree, while an MCTS program

was used to generate moves in the simulation. The program maintaining the UCT tree acts as the client, while the MCTS program used in the simulation acts as the job.

We believe that the job-level computation model can also be easily applied to many other search techniques. In addition to JL-PNS, our ongoing projects are seeking to apply the job-level model to other game search applications [60], such as job-level Monte-Carlo Tree Search (JL-MCTS) for Go, job-level alpha-beta search (JL-ABS) for Chinese chess opening, and job-level learning (JL-Learning) for tuning the weights or scores in games such as Connect6 and Chinese chess.

Another example is solving triangular nim, which can also be applied on job-level computation model. The 8 layer and 9 layer triangular nim are both solved [49][50]. As solving 8 layer triangular nim an example, all the positions are categories into 16 blocks, and we need to solve all the blocks. However, these blocks are dependent; namely, we cannot solve them in parallel. We can start to solve a block only when all of the former blocks are solved. Note that the papers [49][50] proposed some methods to reduce the dependency and made a block can be solved when only some former blocks are solved. The details are omitted here. Since the jobs for solving triangular nim are dynamic generated, the jobs are suitable to be solved on job-level computation model.

4.4.2 Miscellaneous Issues

The first issue for discussion is that of overhead of job dispatching. In the job-level model, the client must wait passively for notification of idling workers. Thus, overhead is incurred for the round trip of notification. In practice, in the job-level system [59], one or more jobs are dispatched to the broker in advance, to keep all workers busy. Note that we do not dispatch a large number of jobs in advance for the reason mentioned in Section 1.

The second issue is that of distributed versus shared-memory. One key for our

job-level model is to leverage a game-playing program which can be encapsulated as a job to be dispatched to a worker remotely in a distributed computing environment. Distribution, however, means that some data, such as transposition tables, cannot be shared by different jobs. However, if the job supports several threads and the worker offers several cores, then the job can still be run with several threads on the worker in the job-level system.

The third discussion is that of the quality of game-playing programs. In our experiences of using JL-PNS, we observed that the quality of game-playing programs affects the total computation time significantly. In our earlier versions of NCTU6, we could not solve the Straight opening after a hundred thousand jobs, and solved the Mickey Mouse opening with many more than that. After we improved NCTU6 in later versions, the Straight opening, as well as many other positions, was solved, and the Mickey Mouse opening was solved with fewer jobs (almost half). On the other hand, JL-PNS or job-level search can be used to indicate the quality of game-playing programs.

The last issue is how to use the result of JL-PNS to build the openings. PNS is a best-first search algorithm mainly designed for proving or disproving positions. The problem is that if the positions cannot be solved by PNS, the result of PNS is hard to tell which move is the best move to play, since the pn/dn are designed to indicate how fast the position is to be proved/disproved. Thus, when we use JL-PNS to build the Connect6 opening, the positions which are proved or disproved can be easily added to the openings database. However, for the positions which are not proved or disproved yet, one of possible ways is to choose the move with the smallest ratio of pn and dn for black and the move with the biggest ratio of pn and dn for white. According to our experiences, this method is not perfect. This leaves as an open problem.

4.5 Conclusion

Generic job-level search can be used to leverage game-playing programs which are already written and encapsulated as jobs. In this chapter, we present and focus on a Job-Level Proof Number Search (JL-PNS), a kind of generic job-level search, and apply JL-PNS to solving automatically several Connect6 positions including some difficult openings. The advantages are as follows.

- This thesis proposes the job-level computation model. The benefits of job-level includes the following: develop clients and jobs independently, run jobs in parallel, maintain the generic search in the client, and monitor the search tree easily. The first also implies that it is easy to develop job-level search without extra development cost to the game-playing programs (like NCTU6).
- This thesis proposes a new approach, JL-PNS (job-level proof number search), to help solve the openings of Connect6.
- This thesis successfully uses JL-PNS to solve several positions of Connect6 automatically, including several 3-move openings in Figure 32. No Connect6 human experts were able to solve them. From the results, we expect to solve and develop more Connect6 openings.
- For JL-PNS, this thesis proposes some techniques, such as the method of postponed sibling generation and the policies of choosing MPNs.
- Our experiments demonstrated that the MF-VE policy performs best. Thus, it is recommended to use this policy to solve positions.
- Our experiments demonstrated an average speedup of 8.58 on 16 cores.

In addition to JL-PNS, we can apply the job-level model to other applications [60], such as JL-MCTS for Go and JL-ABS for Chinese chess.

Chapter 5 Conclusions

The purpose of this thesis is to use volunteer computing to solve computer game problems, including the minimum Sudoku problem and the Connect6 game openings. This thesis uses traditional volunteer computing, BOINC, to help solve the minimum Sudoku problem. However, the Connect6 game openings are not suitable to be solved using the traditional volunteer computing. Thus, this thesis proposes job-level volunteer computing to help solve the Connect6 game openings efficiently.

For solving the minimum Sudoku problem, this thesis presents a more efficient algorithm, named disjoint minimum unavoidable set (DMUS), and incorporates it into the program Checker written by McGuire in 2006. In total we speedup the program by a factor of about 128.

We use traditional volunteer computing, BOINC, to help solve the minimum Sudoku problem. The Sudoku project started on October 2010. At the end of July 2013, the project has been running more than 680 million credits on BOINC [5] and completed the checking of more than 93% primitive grids, and no 16-clue grids have been found yet. We expect to complete the result soon.

For solving the Connect6 game openings, this thesis proposes job-level volunteer computing. The benefits of job-level include the following: develop clients and jobs independently, run jobs in parallel, maintain the generic search in the client, and monitor the search tree easily. The first also implies that it is easy to develop job-level search without extra development cost to the game-playing programs (like NCTU6).

Many search algorithms can be incorporated into job-level volunteer computing, such

as proof number search, monte-carlo tree search, and A*. This thesis propose generic (best-first) search, and modified it into generic job-level search, which can be run in parallel. This thesis incorporates proof number search into job-level volunteer computing, which is called JL-PNS, and use it to solve the Connect6 game openings. This thesis also proposes some methods to improve job-level search, such as the initialization of proof number and disproof number, postponed sibling generation, and polices for pre-update phase. Finally, this thesis successfully solves several positions of Connect6, including several 3-move openings shown in Figure 32, especially the difficult openings like mickey mouse opening and straight opening, and no Connect6 human experts were able to solve them. From the results, we expect to solve and develop more Connect6 openings.

References

[1] L.V. Allis, *A knowledge-based approach of Connect Four: The game is over, white to move wins*, M.Sc. Thesis, Vrije Universiteit Report No. IR-163, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1988.

[2] L.V. Allis, *Searching for solutions in games and artificial intelligence*, Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[3] L.V. Allis, H.J. van den Herik and M.P.H. Huntjens, "Go-Moku Solved by New Search Techniques," *Computational Intelligence*, vol. 12, pp. 7–23, 1996.

[4] L.V. Allis, M. van der Meulen and H.J. van den Herik, "Proof-number Search," *Artificial Intelligence*, vol. 66 (1), pp. 91–124, 1994.

[5] D.P. Anderson, "Boinc: A System for Public-resource Computing and Storage," *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing* (*GRID'04*), IEEE CS Press, Pittsburgh, USA, pp. 4-10, 2004.

[6] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An Experiment in Public-Resource Computing." *Communications of the ACM*, vol. 45(11), pp. 56–61, 2002.

[7] P. Berman, and G. Schnitger, "On the Complexity of Approximating the Independent Set Problem," Springer-Verlag, *Lecture Notes in Computer Science*, vol. 349, pp. 256–267, 1989.

[8] D.M. Breuker, J. Uiterwijk and H. J. van den Herik, "The PN2-search Algorithm," In H. J. van den Herik, B. Monien (Eds.), *Advances in Computer Games*, vol. 9, IKAT, Universiteit Maastricht, Maastricht, The Netherlands, pp. 115–132, 2001.

[9] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *the IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4(1), Forthcoming, 2012.

[10] M. Cambell, AJ Hoane Jr., F.-H. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, pp. 47–83, 2002.

[11] G. M. Chaslot, J-B. Hoock, J. Perez, A. Rimmel, O. Teytaud, and M. H. M. Winands,

"Meta Monte-Carlo Tree Search for Automatic Opening Book Generation," *In Proceedings of the IJCAI'09 Workshop on General Intelligence in Game Playing Agents*, pp. 7–12, 2009.

[12] G. M. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," *the 6th International Conference on Computers and Games* (*CG2008*), Beijing, China, 2008.

[13] J.-P. Delahaye, "The Science Behind Sudoku," *Scientific American*, vol. 294(6), pp. 80–87, 2006.

[14] Einstein@home, Available: http://einstein.phys.uwm.edu/.

[15] H.-R. Fang, T.-S. Hsu, S.-C. Hsu, "Construction of Chinese Chess Endgame Databases by Retrograde Analysis," *Revised Papers from the Second International Conference on Computers and Games*, pp.96–114, 2000.

[16] G. Fedak, C. Germain, V. Neri and F. Cappello, "Xtremweb: A Generic Global Computing System," *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid* (*CCGRID2001*): *Workshop on Global Computing on Personal Devices*, IEEE CS Press, Brisbane, Australia, pp. 582–587, 2001.

[17] B. Felgenhauer, and F. Jarvis, "Mathematics of Sudoku I," *Math. Spectrum*, vol. 39, pp. 15–22, 2006.

[18] G. Fowler, Fowler's sudoku solver, Available: http://www2.research.att.com/~gsf/sudoku/sudoku.html, 2007.

[19] R.U. Gasser, "Solving Nine Men's Morris," In R.J. Nowakowski, *Games of No Chance*, *MSRI Publications*, vol. 29, Cambridge University Press, Cambridge, MA, pp. 101–113, 1996.

[20] S. Gelly, and D. Silver, "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go," *Artificial Intelligence*, vol. 175, pp. 1856–1875, July 2011.

[21] H.J. van den Herik, J.W.H.M. Uiterwijk and J.V. Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134 (1-2), pp. 277–311, 2002.

[22] H.J. van den Herik and M.H.M. Winands, "Proof-Number Search and its Variants," *Oppositional Concepts in Computational Intelligence*, pp. 91-118, 2008.

[23] Y.-L Huang, *The Study of Minimum Sudoku*, Master's thesis (in Chinese), Graduate Department of Compute Science, National Chiao Tung University, Taiwan, 2009.

[24] ICGA Tournaments, Available: http://www.grappa.univ-lille3.fr/icga/

[25] H. Iida, M. Sakuta, and J. Rollason, "Computer Shogi," *Artificial Intelligence*, vol. 134(1–2), pp. 121–144, 2002.

[26] A. Kishimoto and Y. Kotani, "Parallel AND/OR Tree Search Based on Proof and Disproof Numbers," *Fifth Games Programming Workshop*, vol. 99(14) of IPSJ Symposium Series, pp. 24–30, 1999.

[27] A. Kishimoto, and M. Müller, "DF-PN in Go: Application to the One-Eye Problem." In H.J. van den Herik, H. Iida, and E. A. Heinz, editors, *Advances in Computer Games Conference* (*ACG'10*), pp. 125–141. Kluwer Academic, 2003.

[28] H.-H. Lin, D.-J. Sun, I.-C. Wu and S.-J. Yen, "The 2010 TAAI Computer-Game Tournaments", ICGA Journal (SCI), vol. 34(1), March 2011.

[29] H.-H. Lin and I.-C. Wu, "An Efficient Approach to Solving the Minimum Sudoku Problem", ICGA Journal (SCI), vol. 34(4), pp. 191–208, December 2011.

[30] H.-H. Lin, and I.-C. Wu, "Solving the Minimum Sudoku Problem", *The International Workshop on Computer Games* (*IWCG 2010*), Hsinchu, Taiwan, November 2010.

[31] P.-H. Lin and I.-C. Wu, "NCTU6 Wins Man-Machine Connect6 Championship 2009," *ICGA Journal*, vol. 32(4), pp. 230–232, 2009.

[32] Y.-K. Lin, "*Research on Minimum Sudoku Generator*", Master's thesis (in Chinese), Graduate Department of Computer Science and Information Engineering, National Taiwan Normal University, Taiwan, 2007.

[33] G. Mailer, "*A Guess-Free Sudoku Solver*," Master's thesis, Graduate Department of Computer Science, the University of Sheffield, 2008.

[34] G. McGuire, Sudoku checker and the minimum number of clues problem, Available: http://www.math.ie/checkerold.html, 2006.

[35] G. McGuire, B. Tugemann, and G. Civario, "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem," Available: http://www.math.ie/McGuire_V1.pdf, January, 2012.

[36] A. Nagai, *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*, Ph.D. thesis, University of Tokyo, Japan, 2002.

[37] O. Patashnik, "Qubic: 4×4×4 Tic-Tac-Toe," *Mathematical Magazine*, vol. 53, pp. 202–216, 1980.

[38] J. Pawlewicz and L. Lew, "Improving Depth-first Pn-search: 1+ε Trick," In H. J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Fifth International Conference on*

Computers and Games, vol. 4630 of LNCS, pp. 160–170, Computers and Games, Springer, Heidelberg, 2006.

[39] PrimeGrid, Available: http://www.primegrid.com/.

[40] J. Robson, The complexity of Go, in: *Proc. IFIP* (*International Federation of Information Processing*), pp. 413–417, 1983.

[41] G. Royle, Minimum Sudoku. Available: http://people.csse.uwa.edu.au/gordon/sudokumin.php, 2007.

[42] E. Russell, and F. Jarvis, "Mathematics of Sudoku II," *Math. Spectrum*, vol. 39, pp. 54–58, 2006.

[43] A. Saffidine, N. Jouandeau, and T. Cazenave, "Solving Breakthrough with Race Patterns and Job-Level Proof Number Search," *the 13th Advances in Computer Games Conference* (*ACG'13*), Tilburg, The Netherlands, 2011.

[44] J.T. Saito, M.H.M. Winands and H.J. van den Herik, "Randomized Parallel Proof-Number Search," *Advances in Computer Games Conference* (*ACG2009*), *Lecture Notes in Computer Science (LNCS 6048)*, pp. 75–87, Palacio del Condestable, Pamplona, Spain, 2009.

[45] J. Schaeffer and H. J. van den Herik, "Games, computers, and artificial intelligence," *Artificial Intelligence*, vol. 134(1–2), pp. 1–7, 2002.

[46] J. Schaeffer, N. Burch, Y.N. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu and S. Sutphen, "Checkers is Solved," *Science*, vol. 5844(317), pp. 1518–1552, 2007.

[47] M. Seo, H. Iida and J. Uiterwijk, "The PN*-search Algorithm: Application to Tsumeshogi," *Artificial Intelligence*, vol. 129(1-2), pp. 253–277, 2001.

[48] SETI@home, Available: http://setiathome.ssl.berkeley.edu.

[49] Y.-C. Shan, I.-C. Wu, H.-H. Lin, and K.-Y. Kao, "Solving Nine Layer Triangular Nim," *Journal of Information Science and Engineering* (*SCI*), vol. 28(1), pp. 99–113, January 2012.

[50] Y.-C. Shan, I.-C. Wu, H. –H. Lin, and K.-Y. Kao, "Solving 9 Layer Triangular Nim," *The International Workshop on Computer Games* (*IWCG 2010*), Hsinchu, Taiwan, November 2010.

[51] C. E. Shannon, "Programming a computer for playing chess," *Philisophical Magazine*, vol. 41, pp. 256–275, 1950.

[52] Sudoku at VTaiwan Project on BOINC, Available: http://sudoku.nctu.edu.tw/, October, 2010.

[53] Sudoku Forum, Available: http://www.setbb.com/phpbb/index.php?mforum=sudoku, 2009.

[54] Taiwan Computer Game Association, Available: http://tcga.ndhu.edu.tw/

[55] Taiwan Connect6 Association, Connect6 homepage, Available: http://www.connect6.org/.

[56] T. Thomsen, "Lambda-Search in Game Trees - With Application to Go," *ICGA Journal*, vol. 23(4), pp. 203–217, 2000.

[57] J. Wagner and I. Virag, "Solving Renju," *ICGA Journal*, vol. 24(1), pp. 30–34, 2001.

[58] M.H.M. Winands, J.W.H.M. Uiterwijk and H.J. van den Herik, "PDS-PN: A New Proof-number Search Algorithm: Application to Lines of Action," In J. Schaeffer, M. Müller, and Y. Björnson, editors, *Computers and Games 2002*, vol. 2883 of LNCS, pp. 170–185. Computers and Games, Springer, Heidelberg, 2003.

[59] I.-C. Wu, C.-P. Chen, P.-H. Lin, K.-C. Huang, L.-P. Chen, D.-J. Sun, Y.-C. Chan and H.-Y. Tsou, "A Volunteer-Computing-Based Grid Environment for Connect6 Applications," *IEEE International Conference on Computational Science and Engineering* (*CSE2009*), vol. 1, pp. 110–117, 2009.

[60] I-C. Wu, S.-C. Hsu, S.-J. Yen, S.-S. Lin, K.-Y. Kao, J.-C. Chen, K.-C. Huang, H.-Y. Chang, and Y.-C. Chung, "A Volunteer Computing System for Computer Games and its Applications," an integrated project, National Science Council, Taiwan, 2010.

[61] I.-C. Wu, D.-Y. Huang and H.-C. Chang, "Connect6," *ICGA Journal*, vol. 28(4), pp. 234–242, 2006.

[62] I.-C. Wu and D.-Y. Huang, "A New Family of K-in-a-row Games," *Advances in Computer Games Conference* (*ACG2005*), Taipei, Taiwan, 2005.

[63] I.-C. Wu, H.-H. Lin, D.-J. Sun, K.-Y. Kao, P.-H. Lin, Y.-C. Chan, and P.-T. Chen, "Job-Level Proof Number Search", *the IEEE Transactions on Computational Intelligence and AI in Games* (*SCI*), vol. 5(1), pp. 44–56, March 2013.

[64] I.-C. Wu and P.-H. Lin, "NCTU6-Lite Wins Connect6 Tournament," *ICGA Journal*, vol. 31(4), pp. 240–243, 2008.

[65] I.-C. Wu and P.-H. Lin, "Relevance-Zone-Oriented Proof Search for Connect6," *IEEE*

Transaction on Computational Intelligence and AI in Games, vol. 2(3), September 2010.

[66] I.-C. Wu, H.-H. Lin, P.-H. Lin, D.-J. Sun, Y.-C. Chan and B.-T. Chen, "Job-Level Proof-Number Search for Connect6," *International Conference on Computers and Games* (*CG2010*), Kanazawa, Japan, 2010.

[67] I.-C. Wu, H.-T. Tsai, H.-H. Lin, Y.-S. Lin, C.-M. Chang, P.-H. Lin, "Temporal Difference Learning for Connect6," *Advances in Computer Games* (*ACG 13*), Tilburg, The Netherlands, 20–22, November 2011.

[68] I.-C. Wu and S.-J. Yen, "NCTU6 Wins Connect6 Tournament," *ICGA Journal*, vol. 29(3), pp. 157–158, September 2006.

[69] S.-J. Yen, J.-C. Chen, T.-N. Yang, and S.-C. Hsu, "Computer Chinese Chess," *ICGA Journal*, vol. 27(1), pp. 3–18, March 2004.

[70] S.-J. Yen, C.-W. Chou, H.-H. Lin and I.-C. Wu, "TAAI 2010 Computer Go Tournaments," ICGA Journal, vol. 34(1), pp. 48-50, March 2011.

Vita

Hung-Hsuan Lin was born in Taichung, Taiwan in 1984. He received the B.S. and Ph.D. degree in Computer Science and Information Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 2007 and 2013, respectively. His research interests include artificial intelligence, computer game, volunteer computing and cloud computing.