Playing Games with the Job-Level Computation System

Chung-Chin Shih[†1] Ting han Wei[†1] Zheng-Yuan Lee[†1] I-Chen Wu [†1]

Abstract: This paper describes how the job-level computation system is used as a game playing agent, with a 90% win rate for Connect6 tournaments on the website Little Golem between the years 2009 to 2018. In addition, we also construct a win/loss database as an add-on to the existing job-level computation system. This database helps save precious computing resources when encountering previously solved game positions. In the experiments, a benchmark of 32 Connect6 openings is used. The results show that there are about 94.2% and 96.5% of the jobs can be conserved in JL-PNS and JL-UCT respectively for the benchmark when applying the win/loss database. Moreover, we discuss the issues about Graph Interaction Problem and how we generate hash keys of win/loss database.

Keywords: Computer games, Job-level computing, Distributed computing, Connect6.

1. Introduction

Job-level computing (JL), proposed by Wu *et al*. [1], is a distributed computing scheme that can be used to parallelize larger problems by breaking them down into smaller jobs. When used in the context of computer game analysis (e.g. solving game positions), the JL system works by encapsulating specific game positions and a corresponding game AI into a *job*. Each job will be dispatched to a *worker*, who will then perform calculations for the job (e.g., search for the best move for the job's given position) with a predefined budget. Upon completion, the worker returns the result back to the job-level search tree for updates. An actual implementation of a JL application involves the distributed computing environment, the algorithms used to parcel jobs, and defined behaviors for workers. It is collectively referred to as a *job-level computation system* (JLCS).

JL has been successfully used in analyzing opening positions for complex games such as Connect6 [1] [3], Chinese chess [2], and Hex [4]. The job-level computation system can be implemented with different search algorithms. *Job-level proof number search* (JL-PNS) was used to solve several Connect6 openings [1]. Chen *et al.* verified Chinese chess opening book positions using *job-level alpha-beta search* (JL-ABS). Another implementation, *job-level upper-confidence bound tree search* (JL-UCT), was used to solve Hex openings [4]. Lastly, Wei *et al.* compared JL-UCT and JL-PNS on their performance in solving Connect6 openings and constructing Connect6 opening books [3].

Collectively, the above-mentioned games can be prone to combinatorial explosion and memory restrictions. One of JL's main advantages is that it can utilize existing game AIs, while also benefiting from problem parallelization. This allows JL applications to be suitable for bigger problems, while simultaneously allowing faster speed up during analysis.

Meanwhile, JL systems can also be used as a game playing agent. This paper describes in detail how JL was applied to playing Connect6 on the website *Little Golem* (LG) [9]. JL was used to generate the best move to play for each turn, with the exception of the opening moves, which were determined manually. The results show that the JL playing agent was able to win 205 games out of a total of 228 games, and 12 out of 18 tournaments.

In the process of playing games with JL systems, we identified some potential improvements. Identical positions are often encountered across different games. Since each game is treated independently by the JL system, we may end up wasting precious computing resources on positions that have already been thoroughly analyzed, or in other cases, completely solved.

To avoid the above problem, we implemented a *win/loss database* that collects all previously solved positions. Upon encountering these previously solved positions in future games, the stored results can be used immediately, saving computing resources for new positions. This allows us to ultimately solve more complex openings and also improve the JL system's playing strength.

This paper is summarized as follows. Section 2 reviews background information, including the JLCS, Connect6 and our experiences on LG tournaments. In Section 3, we describe in more detail the implementation of the win/loss database in a JLCS. The experiments for the win/loss database in given in Section 4. In Section 5, some implementation issues for the win/loss database are discussed. Lastly, we conclude in Section 6.

2. Background

In this section, an introduction of a JLCS will be given in Subsection 2.1, including the components of a JLCS, and

†1 National Chiao Tung University, Hsinchu, Taiwan.

© 2018 Information Processing Society of Japan

the operations involved in a job-level search. Subsection 2.2 introduces Connect6 and the program NCTU6. Subsection 2.3 introduces the website Little Golem, and discusses our experiences using JLCS as a game playing agent on it. In Subsection 2.4, we review related work.

2.1 Job-Level Computation System (JLCS)

A job-level computation system is mainly composed of two components: a JL system and a client that connect to it. When a JLCS wants to analyze a given game position, the client encapsulates the game position into a job, and submits it to the JL system. Once the job is completed, the JL system returns the job result back to the client so that it can update the JL search tree. The above process is illustrated in Fig 1.

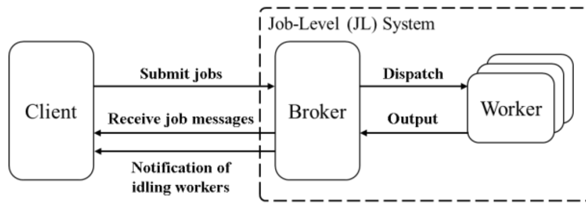

Fig 1. Illustration of a job-level computation system

A generic JL computation for tree search comprises four phases: selection, pre-update, execution, and update.

When the JL system notifies the client of idling workers, a generic JL computation starts. In the selection phase, it continues to select the most promising node (MPN) according to a specified algorithm from the root downwards, until it reaches a leaf. For example, if we use JL-UCT, UCB [20] applied to trees is used as the criterion to select the next child for visiting. The game position at the leaf is then encapsulated as a job with a predefined budget, and then submitted to the JL system, entering the execution phase.

Once the job-level system returns the job result, it enters the update phase. A node is expanded to store information from the job result such as the best move to play, and it updates data from the leaf upwards to the root using the information of the job result.

However, in the situation where there are multiple idling workers, we need to prevent the JLCS from selecting the same node for job submission. To do this, a pre-update phase is added after the selection phase. In the pre-update phase, the simplest policy is to flag the leaf node that has been selected for submitting the job. This means that the next selection will not select the same leaf. There are several other policies such as virtual win, virtual loss, and greedy, which are all discussed in depth in previous work [1], so we will not describe them here.

When the algorithm reaches its predefined computation budget (say, 30,000 jobs) or the root is already proved/disproved, the search is complete. We then use the JL search tree to analyze the game position at the root. The above process is illustrated in Fig 2.

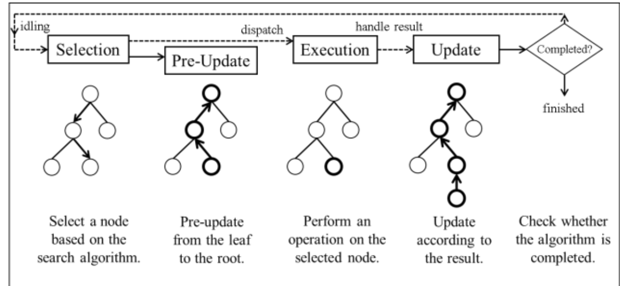

Fig 2. The four phases of a generic job-level computation

Following the mechanism described above, note that we only expand a child at a time after the execution phase because it is inefficient to expand all possible moves in the JLCS [1]. However, this implies that there would be no opportunity to expand other children for each of the selected nodes. Therefore, the *postponed sibling generation* method [1] is used. When a leaf node is selected according to the criteria in the selection phase, we submit not only the job on the selected leaf node but also another job on its parent node simultaneously. The job submitted on the parent node is given an extra restriction on the game AI so that it does not return the moves that the parent has already generated. For the case of NCTU6, the first layer of the game AI's search tree will then exclude computing the restricted moves. In other words, NCTU6 will ignore these moves in the first layer of its search tree completely.

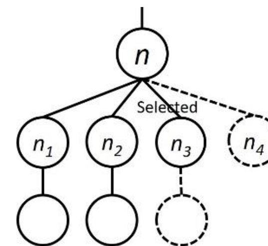

Fig 3. The postponed sibling generation method

To state more formally, the postponed sibling generation method is illustrated in Fig 3. The node n_3 is the leaf node selected now, which we submit a job for. Its parent n has currently three children. Simultaneously, another job is submitted for n with the restriction that the game AI is not to consider the three existing children moves n_1, n_2, and n_3. Once the client receives this job with the restrictions, the returned move is now the 4_{th} best move, assuming that the game AI will always return the best move in descending order of move quality, and subsequently n_4 is created. And so, one sibling of the

© 2018 Information Processing Society of Japan

selected node is expanded. As a result, all possible child nodes of n will be eventually explored, given an infinite amount of computation budget. For this reason, the ability to ignore specific moves is a prerequisite for a game AI if it is to be used in a JLCS without modifications.

In the JL system, the broker is responsible for receiving jobs from the client and managing the workers. If a worker has idle computation resources, it notifies the broker, and the broker will dispatch a job to it for computation.

We deploy our JLCS on the Computer Game Desktop Grid (CGDG) [11]. CGDG is a volunteer computing system developed by Wu *et al.* and aims to take advantage of idle computing resources. With CGDG, the volunteers can choose the number of CPU cores that they wish to donate to the grid. During the LG competitions, which we will discuss in more detail in Subsection 2.3, dozens to hundreds of CPU cores were available via CGDG for the JL player.

2.2 Connect6 and NCTU6

Connect6 is a two-player board game introduced by Wu *et al*. [12] [13]. Both players play on a 19x19 board, where the game starts with the black player playing one stone in her first move. Both players then take turns to play two stones for each move. The winner of a Connect6 game is the first player to connect six consecutive stones of her own horizontally, vertically, or diagonally.

NCTU6 is a Connect6 program developed by a team led by Wu, winning several Connect6 tournaments and man-machine championships [14] [15] [16] [17] [18]. It mainly uses relevance-zone based proof search [19] to solve Connect6 game positions effectively. NCTU6 is used as the game AI in the JLCS in this paper.

2.3 Playing on Little Golem (LG)

LG is a website that holds tournaments for games such as Chinese chess, Go, Gomoku, Connect6 etc., all of which is open to the public. Since 2009, we have been part of the Connect6 tournament, playing as the players Happy6 and Lomaben. For each game of the Connect6 tournament, both players will be given a total of 240 hours to play. Whenever a move is played, an additional 36 hours will be added to the player's total thinking time, up to a maximum of 240 hours. From our experience of playing on LG, it often takes weeks or even months to complete a game. Given the longer timeframe, JL is more suitable as a playing agent than exploiting a game AI purely for two reasons. First, JL allows for a much larger-scale analysis, for which conventional game AI cannot handle due to memory constraints. Second, unlike the case of a real-time competition, performance is less important for longer games on LG. The overhead involved in a JL system is significant, but at the same time JLCS can fully utilize the long thinking time allowed on LG, whereas a conventional game AI might not. Note that the JLCS player can take up to several hours to generate the best move to play, during which time the search tree is explored more thoroughly than a conventional game AI.

JL-PNS and JL-UCT have both been applied to solve Connect6 openings [3]. JL-PNS was developed first, so it was used exclusively in the earlier tournaments. While JL-PNS is often suitable to solving games, UCT was designed to choose the best moves to play. Wei *et al*. [3] showed that, as expected, JL-UCT is able to choose the better move to play for openings, and can be comparable to JL-PNS at solving positions. For the more recent tournaments, JL-UCT has been used to great effect.

For each turn, the JL system computes about 15,000 to 30,000 jobs using workers on the CGDG. Each job consists of a worker invoking NCTU6 and computing the best move to play given a certain position in the overall JL search tree. When using JL-PNS to play, the worker's returned candidate moves are classified into 13 ordered game statuses according to NCTU6's evaluation function. After 15,000 to 30,000 jobs are computed, we choose the next move by sorting them according to their games status; in the case of ties, the moves that belong to the best game status is further sorted by its proof number/disproof number ratio. That is, the move with the smallest proof number/disproof number ratio of the best game status is chosen. This method tended to be overly complicated and in many cases, produced unexpected results. As an improvement, for JL-UCT, we simply choose the move that has the largest subtree to play like with MCTS.

We analyzed 49 games which were recorded in detail in recent LG tournaments from 2015 to 2018. In these games, on average, about 10 moves (the least being 2, at most 25) are determined by JLCS. For each move determined by JLCS, there's a corresponding JL search tree produced. There are no more than 30,000 nodes in each search tree.

From these search trees, we observed that there are on average 6.7% identical positions in the JL search trees of each move in the same game. The low identical percentages of the JL search trees is because the opponent may not respond with the moves that JLCS considered, opting to play instead moves that were investigated less from the perspective of the JL player.

Furthermore, in these games, we also observed that in a single game there are on average 6.9% identical positions that other games have searched previously. As mentioned in the introduction, opening moves are determined manually. This is because we wish to compile data for a wide variety of openings, and is also a reason why identical positions between games is still low. Despite this, however, in extreme cases a game could have up to 80 percent repeated positions from other games (including positions

© 2018 Information Processing Society of Japan

that are outright proved/disproved). In this case, the JL player can benefit significantly by reusing the previous results, through the use of a win/loss database, which we will describe in Section 3.

2.4 Related Work

Schaeffer *et al*. solved checkers [6] using distributed computing, where their approach contains two main components: the proof-tree manager and the proof solver. The proof-tree manager performs the same role as the client in our JLCS, where it maintains a proof tree and dispatches positions to the proof-solver, similar to the jobs submitted to workers in JLCS. Chaslot *et al*. [7] proposed Meta-MCTS to construct Go openings. They used the UCT algorithm to select the next position, and then dispatch it to the worker for simulation, similar to JL-UCT in JLCS. The above approaches are limited to proof-number search and UCT and were only applied to specific games. JLCS is more general, and can apply different kinds of search algorithms like JL-PNS and JL-UCT, for various games. Saffidine *et al*. [8] has applied JL-PNS to solving breakthrough, and proposed using the simple flag policy to prevent starvation.

3. Win/Loss Database

The win/loss database is a kind of transposition table. The traditional transposition table mechanism often stores all positions that it has analyzed before, then reuses the information once it encounters the same position. In contrast, for our win/loss database, we are only interested in positions that have been proved to be winning or losing instead of storing every encountered position. This is because a tremendously large amount of memory space is required to store the wide variety of game positions we may encounter throughout different LG matches. By skipping the unsolved positions, we will be able to store the most important information with much less memory. Once a JL search has started, the worker can query the win/loss database to find whether the returned position has already been proved or disproved.

The win/loss database is constructed and used as follows. Once a position is proved by the JLCS, we store its entire solution tree into the database. More specifically, for each of the position's descendants with a solved *win/loss/draw* value, its hash key along with its theoretical value are stored into the win/loss database for future use.

In our implementation, when a worker is given a job to analyze a position *p*, it will return the best move to play *m*, leading to a new position *p'*. Let the hash key for *p'* be *h*, which is computed by the arrangements of the stones on the board and the player to play with Zobrist hashing [22]. The worker will query the win/loss database with *h* to find whether *p'* is proved/disproved or otherwise. If *h* is found in the database, the worker will then modify the job's result with the database's theoretical value, and then return the modified result to the JL search tree for update; if not, we update the JL search tree with just the original job result. The above process is illustrated in Fig 4.

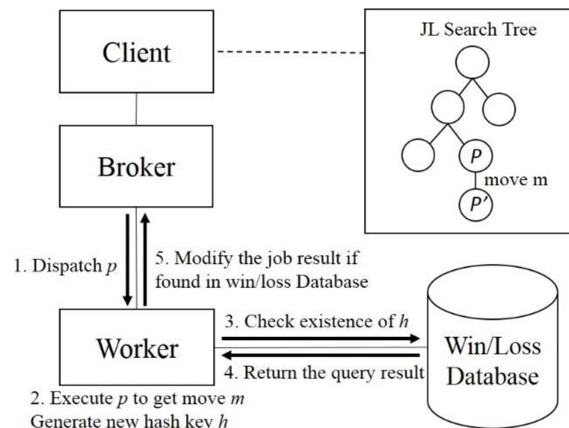

Fig 4. The process of querying the win/loss database

This added mechanism does not need extensive changes to the original software architecture [5] of a JLCS. The JLCS will treat the job result as if the given position *p'* is solved by the worker itself.

It is worth noting that the original responsibility of the worker is only to invoke the specified game AI for computing the best move *m*, and does not include calculating the hash key *h* of the position *p*' after playing the move *m*. For this reason, to query the win/loss database for the theoretical value, the worker will need to know how to compute the new hash key *h* by itself for different games. That is, it will need to know the rules of the game (e.g. for Go, it should know basic rules for capturing) and play the moves from an empty board to the desired position *p*' for computing the hash key *h*. To be more specific, when the worker get the best move *m* from the game AI, the worker calls a subroutine to generate the next hash key with the arguments: game type *g*, position *p*, best move *m*, and then follows the relevant game rules to get the new position *p*' for computing the new hash key. This game-specific code is the responsibility of the game developers, and is originally shared between different job-level applications [5]. To use the win/loss database, the game developer's work includes implementing this game-specific code on the worker when generating the hash keys.

For maintainability and robustness, we choose MariaDB [10], which is a community-developed fork of MySQL, to construct our win/loss database, allowing us to query the database using SQL commands through the network. By choosing to implement this way, the entire win/loss database can be treated as an add-on mechanism to our JLCS. We only need to implement interfaces on the

© 2018 Information Processing Society of Japan

worker to query the information that we need.

A total of 308,111 positions are stored into the win/loss database and cost about 808 MB memory space in MariaDB. It costs about 0.078 seconds to query the database for a given position. The win/loss database values consists of positions from 52 common Connect6 openings and 49 games played on LG (which consists of 480 search trees). The 52 common Connect6 openings have been both solved by JL-UCT and JL-PNS. Among these positions, JL-UCT can solve 23 of them using less jobs, and JL-PNS can solve the remaining 29 positons better. Every opening can be solved within 150,000 jobs.

4. Experiments

To test the functionality and performance of our win/loss database in the JLCS for solving games, we chose 32 Connect6 openings that have all been solved as winning for black previously. These openings are all less than five plies into the game, and are the parent positions of the 52 common Connect6 openings described in Section 3.

Both JL-PNS and JL-UCT are applied to compare node counts between the cases with and without the database. The experiments are tested on CGDG, which typically has about 250 cores. Each job costs about 30 seconds on average. The node counts for both algorithms are listed in Table 1.

	Without DB	With DB
JL-PNS	217502	12405
JL-UCT	249397	8510

Table 1. The total search nodes to solve the 32 Connect6 positions

The results show that about 94.2% of the nodes can be saved for JL-PNS, and 96.5% for JL-UCT, which is a significant improvement. Most of them can be proven in the first move, which is returned by the job that contains a win/loss value in the database.

We also list the number of nodes that are directly proved/disproved by win/loss database in Table 2. Most of the nodes are proved/disproved in the earlier plies, guiding the JL player to find the immediate-win node, or alternatively, to avoid searching immediate-losses to save precious computation resources.

However, as we mentioned in Section 2, the JLCS uses the postponed sibling generation method to expand branches. In other words, it does not expand all children after a single expansion phase. In the case where there is a winning position for a later branch, the game AI might not return it until several earlier branches have been expanded. Therefore, the JLCS cannot use the win/loss value until the winning/losing job in the later branch is returned.

	#Win nodes	#Loss nodes
JL-PNS	36	86
JL-UCT	36	96

Table 2. The number of jobs that used queried values from the win/loss database

5. Discussion

In this section, we discuss some implementation issues when we build the win/loss database to our JLCS. In Subsection 5.1, we discuss the Graph History Interaction (GHI) problem [21], which occurs in retrieving proof/disproof values from a transposition table when cycles are involved. In Subsection 5.2, we discuss the design issues of generating the new hash key on the worker side.

5.1 The GHI Issues of the Win/Loss Database

In our implementation, the GHI problem is prone to occur, resulting in incorrect proofs when reusing the results from a transposition table (i.e. the win/loss database). For the game of Connect6, cycles are not possible. Thus, there is no need to worry about the GHI problem when reusing the theoretical values of a position for Connect6.

However, for games such as Go and Chinese chess, certain rules exist that forbid players from recreating positions that have already been played previously in the same game. Go, for example, prohibits players from playing ko, which occurs frequently (often single ko). More specifically, two positions that have the same arrangement on the board with the same player to play, should, in fact be regarded as different positions in terms of the win/loss values in certain conditions.

To handle the GHI problem, we define columns for preconditions in the win/loss database to discriminate between these cases that share the same board positions, but do not have the same theoretical value. This issue is game-related, and the method to identify duplicates with different values depends on the game developers. We will investigate the issue in the future and apply it to solve game positions for Go.

5.2 The Design Issues of Generating the Hash Key

A potential question is whether the game AI should compute the hash key h instead of having the worker do it. In that case, there would be no need to call a subroutine in the worker to generate h. There are two main reasons why we do not design our mechanism this way.

First, one of the main benefits of the JL model is that the game AI can be developed independently, without the need for modification from the JL application developer.

© 2018 Information Processing Society of Japan

For this reason, it is not preferable to have to modify the source code of the game AI. In any case, the JL application developer may not even have access to the game AI code, or it may be difficult to understand and alter the game AI if the JL developer cannot contact or consult the original game AI authors.

Second, two different game AIs might share the same win/loss information of the same position when we want exploit their differing advantages. In this situation, each AI program will need to contain the same duplicate code to ensure the hashing mechanism is consistent between them. For a more flexible and maintainable JLCS framework, we keep the hashing mechanism sharable and reusable, while also removing the requirement to alter the original game AI.

6. Conclusion

This paper describes how the JLCS, which has been traditionally used to analyze game positions, can also be used as a game playing agent. Our JLCS has been used as a player on the website Little Golem, where it is the owner of the two handles Happy6 and Lomaben. Collectively, the JLCS has won 12 championships[†2] in 18 tournaments from 2009 to 2018, achieving about 90% (205 games) winrate in a total of 228 games. The more recent Lomaben has an Elo rating of 2591 currently, being the top Connect6 player on LG, while the second place player is rated at 2453.

Additionally, we constructed a win/loss database as an add-on to our JLCS. The added win/loss database saves precious computing resources when encountering game positions solved previously, allowing us to focus on new problems, while potentially improving the playing strength when using JLCS as a game playing agent.

Experiment results show that the win/loss database achieves 94.2% savings in jobs for JL-PNS and 96.5% for JL-UCT when solving Connect6 openings compared to not using the database.

Currently, the win/loss database is only designed for Connect6. For other games such as Go, the so-called GHI problem may occur, resulting in incorrect proofs. The win/loss database can be improved by adding pre-conditions with the given hash key, which we will investigate in a more generic way and apply to Go in the future.

References

[1] I-Chen Wu, Hung-Hsuan Lin, Der-Johng Sun, Kuo-Yuan Kao, Ping-Hung Lin, Yi-Chih Chan, and Po-Ting Chen, "Job-Level Proof Number Search", the IEEE Transactions on Computational Intelligence and AI in Games (SCI), Vol. 5, No. 1, pp. 44-56, March 2013.

[2] Jr-Chang Chen, I-Chen Wu, Wen-Jie Tseng, Bo-Han Lin, and Chia-Hui Chang, "Job-Level Alpha-Beta Search", IEEE Transactions on Computational Intelligence and AI in Games (SCI), Vol. 7, No. 1, pp. 28-38, March 2015.

[3] Ting-han Wei, I-Chen Wu, Chao-Chin Liang, Bing-Tsung Chiang, WenJie Tseng, Shi-Jim Yen, and Chang-Shing Lee, Job-Level Algorithms for Connect6 Opening Book, ICGA Journal (SCI), Vol. 37(3), September 2015.

[4] Xi Liang, Ting-han Wei, and I-Chen Wu, Solving Hex Openings Using Job-Level UCT Search, ICGA Journal (SCI), Vol. 37(3), September 2015.

[5] Ting-han Wei, Chao-Chin Liang, I-Chen Wu, and Lung-Pin Chen, Software Development Architecture for Job-Level Algorithms, ICGA Journal (SCI), Vol. 37(3), September 2015.

[6] J. Schaeffer, N. Burch, Y. N. Björnsson, A. Kishimoto, M. Müller, R.Lake, P. Lu, and S. Sutphen, "Checkers is solved," Science, vol. 5844, no. 317, pp. 1518–1552, 2007.

[7] 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 Proc. Workshop General Intell. Game Playing Agents, 2009, pp. 7–12.

[8] A. Saffidine, N. Jouandeau, and T. Cazenave, "Solving breakthrough with race patterns and job-level proof number search," in *Proc. 13th Adv. Comput. Games Conf.*, Tilburg, The Netherlands, 2011, pp. 196–207.

[9] Little Golem, http://www.littlegolem.net/jsp/main

[10] MariaDB, https://mariadb.com/

[11] I.-C. Wu, C.-P. Chen, P.-H. Lin, K.-C. Huang, L.-P. Chen, D.-J. Sun,Y.-C. Chan, and H.-Y. Hsou, "A volunteer-computing-based grid environment for Connect6 applications," in *Proc. 12th IEEE Int. Conf. Comput. Sci. Eng.*, Vancouver, BC, Canada, Aug. 29–31, 2009, pp. 110–117.

[12] I.-C. Wu, D.-Y. Huang, and H.-C. Chang, "Connect6," Int. Comput. Games Assoc. J., vol. 28, no. 4, pp. 234–242, 2006.

[13] I.-C. Wu and D.-Y. Huang, "A new family of k-in-a-row games," in Proc. 11th Adv. Comput. Games Conf., Taipei, Taiwan, 2005, pp. 180–194.

[14] I-Chen Wu and Shi-Jim, Yen, "NCTU6 Wins Connect6 Tournament", ICGA Journal (SCI), Vol.29, No.3, September 2006.

[15] Wu, I.-C. and Lin, P. (2008). Nctu6-Lite Wins Connect6 Tournament. ICGA Journal, Vol. 31, No. 4, pp. 240-243.

[16] Lin, P.-H. and Wu, I. (2009). Nctu6 Wins Man-Machine Connect6 Championship 2009. ICGA Journal, Vol. 32, No. 4, pp. 230.

[17] Wu, I., Lin, Y.-S., Tsai, H.-T. and Lin, P.-H. (2011). The Man-Machine Connect6 Championship 2011.ICGA Journal, Vol. 34, No. 2, pp. 103.

[18] Wei, T.-H., Tseng, W.-J., Wu, I. and Yen, S.-J. (2013). Mobile6 Wins Connect6 Tournament. ICGA Journal, Vol. 36, No. 3, pp. 178-179.

[19] I-Chen Wu and Ping-Hung Lin, "Relevance-Zone-Oriented Proof Search for Connect6", the IEEE Transactions on Computational Intelligence and AI in Games (SCI), Vol. 2, No. 3, pp. 191-207, September 2010.

[20] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M.,

†2 The 2nd Connect6 tournament in 2018 on LG is not counted because it is still ongoing as of this writing, but we are guaranteed to win according to the current score.

© 2018 Information Processing Society of Japan

Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. Computational Intelligence and AI in Games, IEEE Transactions on, Vol. 4, No. 1, pp. 1-43.

[21] Palay, A. J. 1983. Searching with Probabilities. Ph.D. Dissertation, Carnegie Mellon University.

[22] Albert Lindsey Zobrist, A New Hashing Method with Application for Game Playing, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1969.

© 2018 Information Processing Society of Japan