SOFTWARE DEVELOPMENT ARCHITECTURE FOR JOB-LEVEL ALGORITHMS

Ting han Wei[1], *Chao-Chin Liang*[1], *I-Chen Wu*[1] *and Lung-Pin Chen*

Hsinchu, Taiwan

ABSTRACT

Recently, Wu et al. introduced a general distributed computing approach, named Job-Level (JL) Computing. In JL computing, a search tree is maintained by a client process, while search tree nodes are evaluated, expanded, or generated by leveraging game-playing programs. These node operations are encapsulated as coarse-grain jobs, each requiring tens of seconds or more of running the programs. This article presents an abstraction of the JL computing approach and develops a general JL search framework so that common modules may be reused for various JL applications, making JL development easier. We describe in detail the implementation of the JL Proof-number Search (JL-PNS) and JL Upper Confidence Bound Search (JL-UCT) as case studies in the application of the JL search framework. In our case studies, only hundreds of lines of code are required for new JL applications, while the code for the JL framework consists of more than ten thousand lines of code. It demonstrates that this framework can be used to greatly reduce new JL application development and software maintenance efforts.

1. INTRODUCTION

In computer game research, many efforts have been made to design game-playing programs, solve game positions, build opening books, etc. for a growing collection of games. During this pursuit, many search techniques have been proposed and applied to game-playing and solving programs. We list here three search techniques which have, since their inception, become highly influential and widely used. First, *Alpha-Beta Search* (ABS) (Knuth and Moore, 1975) is a well-known search algorithm applied to minimax search trees that prunes unnecessary branches, thereby greatly reducing the amount of search computation while still obtaining the same value as a normal minimax search. Second, *Proof Number Search* (PNS), proposed by Allis et al. (Allis, 1994; Allis, van der Meulen and Van den Herik, 1994), was successfully used to prove game-theoretical values of game positions (Van den Herik, Uiterwijk and Van Rijswijck, 2002) for Connect-Four (Allis, 1994), Gomoku (Allis, 1994; Allis *et al.*, 1994; Allis, van den Herik and Huntjens, 1996), Renju (Van den Herik, Uiterwijk and Van Rijswijck, 2002), Checkers (Schaeffer, Burch, Björnsson *et al.*, 2007), Lines of Action (Winands, Uiterwijk and van den Herik, 2003; van den Herik and Winands, 2008; Saito, Winands and van den Herik, 2010), Go (Kishimoto and Mueller, 2005), and Shogi (Nagai, 2002). Third, *Monte-Carlo Tree Search* (MCTS) (Coulom, 2006; Kocsis and Szepesvári, 2006; Gelly, Wang, Munos *et al.*, 2006) is a best-first search algorithm using Monte-Carlo simulations as state evaluations. It has been successfully applied to Go (Gelly, Wang, Munos *et al.*, 2006; Enzenberger, Müller, Arneson *et al.*, 2010; Gelly and Silver, 2011), Hex (Huang, Arneson, Hayward *et al.*, 2014), General Game Playing (Björnsson and Finnsson, 2009), Backgammon (Van Lishout, Chaslot and Uiterwijk, 2007) and Phantom-Go (Borsboom, Saito, Chaslot *et al.*, 2007).

Since game-playing programs usually require a large amount of computation, a challenging issue is the parallelization of game-playing programs on shared-memory systems and distributed-memory systems (abbr. distributed systems). For shared-memory systems, many parallelization techniques have been proposed based on multi-threading. For example, parallel ABS (Manohararajah, 2001), the lock-free multithreaded mechanism (Enzenberger and Müller, 2010), parallel dovetailing (Hoki, Kaneko, Kishimoto *et al.*, 2013), parallel depth-first PNS (Kaneko, 2010; Pawlewicz and Hayward, 2014), parallel MCTS (Chaslot, Winands and van den Herik, 2008), and so on. For distributed systems, parallelization is more difficult, and has been investigated through transposition-table driven scheduling (TDS) for parallel MCTS (Yoshizoe, Kishimoto, Kaneko *et al.*, 2011), the Young Brothers Wait Concept (Feldmann, 1993) for parallelizing ABS, and ParaPDS (Kishimoto and Kotani, 1999).

[1] Dept. of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu / Taiwan. Email: icwu@cs.nctu.edu.tw; tinghan.wei@gmail.com

There have also been attempts to generalize game tree search parallelization across game types and search algorithms. Multigame is a system that is capable of solving a variety of games using several distributed search methods (Romein, 2001). While games are defined via the Multigame language, the search methods need to be implemented separately. Since the two goals of Multigame are to allow application programmers to solve games easily, and researchers to experiment on different search methods, there is a strong emphasis on the separation of these two roles, and of what they are capable by using the system. The Asynchronous Parallel Hierarchical Iterative Deepening (APHID) algorithm also tries to apply game tree search on a distributed system, where the work is partitioned into a master process and several slave processes (Brockington, Schaeffer, 2000). APHID's main concept was to show empirically that asynchronous parallelization of game tree search can outperform synchronous methods such as YBWC. It uses an version of iterative deepening alpha-beta search, and is designed so that parallelization can be increased with minimal effort. ZRAM is a software library that parallelizes combinatorial optimization and enumeration problems (Marzetta, 1998). Similar to the above two methods, it tries to simplify parallel application developer efforts by hiding the parallel programming requirements from developers. The parallel components of ZRAM consist of several search engines, service modules, and the machine level host system code. Application developers in turn, only need to use the library to take advantage of the parallelization benefits.

Recently, Wu et al. (2011, 2013) introduced a general approach which was named *Job-Level* (*JL*) *Computing*. In JL computing, a search tree, called *a JL search tree*, is maintained by the *client*. JL search tree nodes are evaluated, expanded, or generated by leveraging game-playing programs. In this approach, game-playing programs are encapsulated as coarse-grain jobs, which typically require tens of seconds or more.

JL computing is similar to the above three methods (Multigame, APHID, ZRAM) in that application developers only require minimal efforts to complete a scalable parallelized game search application. The most distinct difference between JL computing and the above three methods is that JL computing is designed to work with existing game-playing programs. There are other smaller differences; for example, unlike Multigame, JL computing does neither need to specify the game rules via a defined language, nor does it need to supply the system with an evaluation function. With respect to APHID, JL computing is capable of using a wider variety of search algorithms. JL computing is able to work with heterogeneous systems, which ZRAM is not capable of.

Based on JL computing, JL algorithms and applications have been developed in the past. In (Wu *et al.*, 2011), a *JL-PNS* algorithm was proposed by applying JL computing to PNS, which was then used to automatically solve several Connect6 opening positions. Saffidine et al. (2012) also used JL-PNS to solve Breakthrough positions. In (Chen, Wu, Tseng *et al.*, 2015), JL-ABS is proposed to construct a Chinese chess opening book. In (Wei, Wu, Liang *et al.,* 2014), the *JL-Upper Confidence Tree* (JL-UCT) algorithm was compared with JL-PNS in the analysis of opening positions for Connect6. In (Liang, Wei and Wu, 2015), several methods to improve JL-UCT were proposed to solve Hex efficiently.

This article presents an abstraction of the JL computing model and designs a general JL search framework so JL applications can be designed more easily. It is motivated from the observation that most of the code of JL-PNS for Connect6 can be easily reused for that of JL-UCT; furthermore, most of the code of JL-UCT for Connect6 can also be easily reused for Hex and Go. We give a quick review of the JL computing model in Section 2. Based on the model, a general JL framework is proposed and described in order to greatly reduce the efforts of JL application development in Section 3. In Section 4, our case study demonstrates that this framework simplifies development efforts significantly. Only hundreds of lines of code are required for JL application development, such as JL-PNS for Connect6 and JL-UCT for Go and Connect6. In contrast, the code for the JL framework requires above ten thousand lines of code. In (Liang *et al.*, 2015), a JL-UCT Hex application was also developed on top of this framework. Lastly, we make concluding remarks in Section 5.

2. BACKGROUND

This section summarizes the JL computing model in Subsection 2.1. The JL search, which is based on the JL computing model, is described in Subsection 2.2. Two JL search algorithms, JL-PNS and JL-UCT, are reviewed in Subsection 0.

2.1 Job-Level Computing

In the JL computing model, a client[2] dynamically creates jobs and chooses available computing units, called *workers*, to perform these jobs. For example, in a computer game application, the client creates one job to generate or expand a move from a position; typically, the job also entails the evaluation of the generated move as well. The *JL system* consists of a set of workers and a *broker*, which is used to help manage job dispatch to workers.

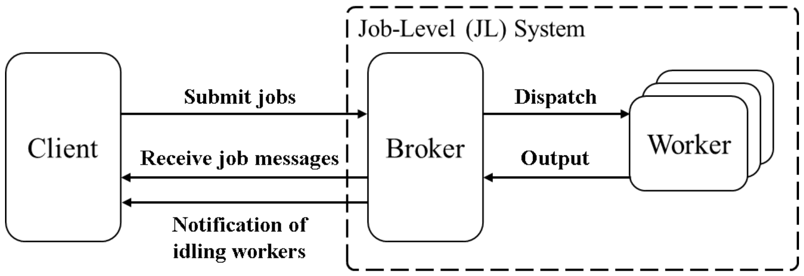
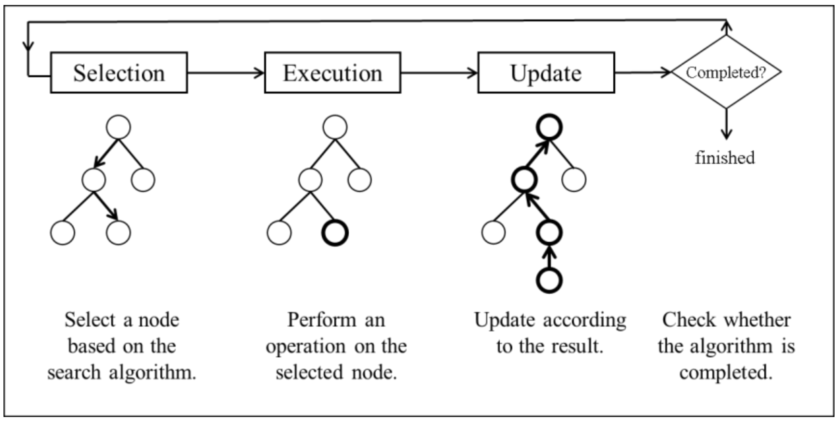


Figure 1: The job-level computing.

As shown in Figure 1, the communication between the client and the JL system mainly includes the following three messages: job submission, notification of idle worker, and job result. While job submission messages are sent from the client to the JL system, the other two are received by the client from the JL system. In JL computing, the programming in the client is based on event handling. Namely, the client waits passively for an event that indicates available workers to submit jobs. Whenever such an event is received from the broker, the client creates or chooses one job, if any, and submits it to available workers via the JL system. In practice, more messages can also be supported to abort jobs, request and receive worker information and job statuses in the system, and so on.

2.2 Job-Level Search

Generic best-first search, or simply called *BFS*, fits many search techniques, such as PNS and MCTS. BFS is usually associated with a search tree, where each node represents a game position and each edge represents a move. The process of BFS usually repeats the following three phases, selection, execution, and update, as shown in Figure 2 below.


Selection
Execution
Update
Completed?
finished
Select a node based on the search algorithm.
Perform an operation on the selected node.
Update according to the result.
Check whether the algorithm is completed.
Figure 2: Phases of BFS.

First, in the selection phase, a node, usually a leaf, is selected based on the search algorithm. For example, PNS selects the most proving node, while MCTS uses the upper confidence bound (UCB) function for selection. Second, in the execution phase, an operation $J(n)$ is performed on the selected node n without changing the search tree. For example, the operation may involve finding the best move from a node n and evaluating its score. Third, in the update phase, the search tree is updated according to the result of $J(n)$. For the above example, if the result is the best move, a node corresponding to the new position after playing

[2] While the terms *master*/*slave* may suit the situation better, we choose to use this term for consistency with previous publications.

the move is added to the search tree, and the status is updated on the path up to the root. After these three phases, we check to see whether BFS is complete. If so, the search process ends; otherwise, the three phases are iterated. For example, completion criteria may state that the BFS stops when the root is proven or disproven, or when the number of iterations exceeds a threshold.

Since the operation $J(n)$ does not change the search tree, the operation can be done as a job by another worker remotely. The job submission may include some data required by $J(n)$, such as game positions. When the job is done, the worker sends the result back to the client. Upon receiving the result, the client runs the update phase to reflect the results to the search tree accordingly. Through this process, BFS becomes a *BFS-based JL search*, which we will refer to as a *BF-JL search* in this article.

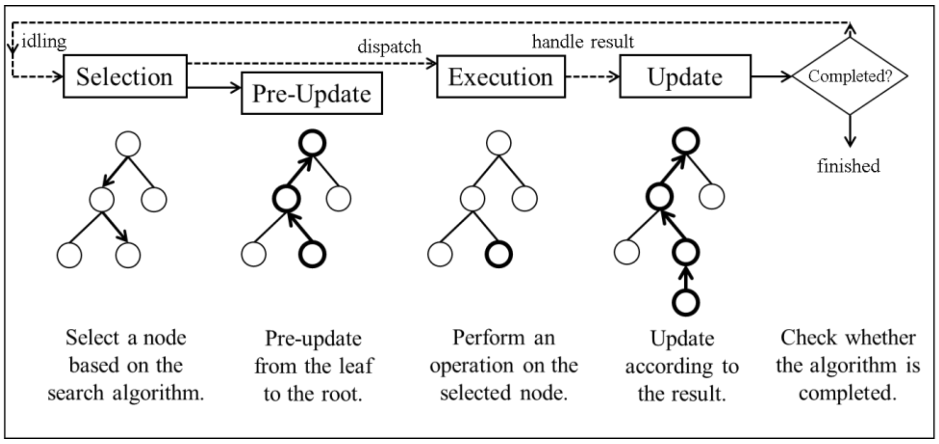

idling
dispatch
handle result
Selection
Pre-Update
Execution
Update
Completed?
finished
Select a node based on the search algorithm.
Pre-update from the leaf to the root.
Perform an operation on the selected node.
Update according to the result.
Check whether the algorithm is completed.

Figure 3: Phases of BF-JL search.

One issue for BF-JL search is that the client may select the same node in response to multiple idle worker notifications if the JL search tree is not updated between these notifications. For this issue, an additional phase, called the *pre-update phase*, is added as shown in Figure 3. This phase can be used to prevent selection of the same node repeatedly by using various policies. One intuitive approach, for example, would be to simply flag the selected nodes to prevent selecting them again later (Saffidine *et al.*, 2012; Wu *et al.*, 2013).

A second issue is determining when to grow multiple children. Wu et al. (2013) pointed out that it is inefficient to expand all possible moves for a position in the job-level model. Therefore, child nodes are generated according to the *postponed sibling generation* method as follows. The job $J(n)$ is redefined as $J(n, C(n))$, where $C(n)$ is a list of prohibited moves which are now excluded from consideration when attempting to find the best move among all possible candidates. Thus, for a node 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. Whenever a leaf is selected, the client will now also submit another job $J(p, C(p))$, where p is the parent of the leaf. In order to prove or disprove positions, the job $J(n, C(n))$ also needs to satisfy the following property: If the result of $J(n, C(n))$ is disproven, this implies that all moves not in $C(n)$ are all disproven as well. In brief, to fit in BF-JL search, game-playing programs need to provide the following two extra functionalities.

1. Support $J(n, C(n))$.
2. For each job $J(n, C(n))$, if the job result indicates a sure loss, all the moves other than the prohibited moves must also be losses.

The BF-JL search has the following advantages:

- Game-playing programs (jobs) and the BF-JL search may be developed independently.
- The BF-JL search can take advantage of distributed computing environments naturally.
- The JL search tree can be monitored easily. Since it usually takes tens of seconds to perform a job which usually generates a new node, the client will be idle most of the time. Users may then use various means, such as a *graphic user interface* (*GUI*), to observe the growth of the JL search tree. This is especially important when enlisting the help of field experts.

2.3 Job-Level Prove Number Search (JL-PNS) and Job-Level Upper Confidence Tree (JL-UCT)

JL-PNS and JL-UCT are two BF-JL searches that have been described in detail in (Wei *et al.*, 2014). We will only give a quick review to these two methods in this article. PNS attempts to minimize the number of node expansions during the process of solving the root of a game search tree. It does this by keeping track of each node's proof and disproof numbers (PN/DN). The proof number of a node signifies the minimum number of leaf nodes that must be expanded for the node to be proven as a winning position, and vice versa for the disproof number. During the selection phase, the search begins at the root of the search tree, and descends through the tree. We see the following: (1) at each OR node, the child with the smallest PN is chosen, and (2) at each AND node, the child with the smallest DN is chosen. Once the search arrives at a leaf node, called the *most proving node* (MPN), the selection process is complete, and the MPN is expanded.

Initially, UCT was initially proposed for the so-called *multi-armed bandit problem* (Auer, Cesa-Bianchi and Fischer, 2002), where a player decides on the best move to make among a list of possible candidates without requiring prior knowledge of the performance of each move. The aforementioned MCTS makes use of UCT, with Monte-Carlo simulations taking place of node evaluations. By taking into consideration each node's win rate WR, visit count N, and the total visit count of the node's parent N_p, UCT is able to balance between exploration (searching widely) and exploitation (searching deep) through the UCB function:

$$WR + C\sqrt{\frac{\log(N_p)}{N}}$$

During selection, at each level of the tree, UCT chooses the node that has the largest UCB value among its siblings, which eventually leads to a leaf that requires expansion, which is similar to what PNS does with the MPN.

In order to apply PNS and UCT to BF-JL search, both methods will first need to support postponed sibling generation, as described in the previous subsection. In this respect, both BF-JL searches behave identically during the expansion phase, where each node expansion is sent as a job and computed by the workers. In contrast, the selection, pre-update, and update phases all differ. We place particular emphasis on the update phase, when the worker has returned a move to play and along with it, the evaluation of the move's resulting position. The move to play is used to generate a new child node, while the evaluation is used to set the initial value for it. For example, JL-PNS will initialize the new child node with a PN and DN, while JL-UCT will do so with a win rate.

For game-playing programs that utilize MCTS, the evaluation may be a win rate value, which can be used directly in the JL-UCT case. As this is relatively trivial, we use the Connect6 program NCTU6 (Wu and Lin, 2010) as an example instead. NCTU6 returns evaluations of the move to play in the form of *game statuses*, which range from B:W (winning move for Black), to B4 (highly favourable for Black) to, B1 (slightly favourable for Black), to W:W (winning move for White). This is illustrated in Table 1. To apply NCTU6 to JL-PNS and JL-UCT, a mapping is required to translate game statuses into PN/DN or win rate values. The details of the settings for PN/DN are described in (Wu *et al.*, 2011), and the settings for the win rates are described in (Wei *et al.*, 2014).

Status	B:W	B4	B3	B2	B1	Stable	Unstable 2	Unstable 1	W1	W2	W3	W4	W:W
PN/DN	0/∞	1/18	2/12	3/10	4/8	6/6	4/4	5/5	8/4	10/3	12/2	18/1	∞/0
Win Rate (%)	100	90	80	70	60	50	50	50	40	30	20	10	0

Table 1: Mapping of statuses to PN/DN values and win rates.

3. JOB-LEVEL FRAMEWORK

Based on the model of JL computing, development of JL applications can be greatly simplified by leveraging existing game-playing programs. However, the following efforts are still required to develop JL applications.

1. Design a middleware to communicate with the JL system, handling tasks such as dispatching jobs and receiving results, as described in Subsection 2.1. As all JL applications need to communicate with the JL system, nearly all the code of the middleware can be shared or reused.
2. Design JL applications to utilize the JL system via the middleware. More specifically, for BF-JL search applications (referred to thereafter as *BF-JL applications*), developers need to maintain and traverse a JL search tree during the four phases. By defining a generic JL search tree that can be used for all BF-JL applications, a large portion of code for tree maintenance and traversal can be shared or reused.
3. Optionally support a *Graphic User Interface* (GUI) in order to monitor the behaviour of JL applications. The GUI can be divided into two parts, game-specific and non-game-specific. While the game-specific part, which includes the board view and game rules, contains code that is unlikely to be shared, the non-game-specific part that includes JL search tree browsing is relatively easier to share. Since the GUI and the generic JL search trees were described in detail in (Liu, Wu, Liao *et al.*, 2013), we will not elaborate on them in this article.

From the above, we first develop a *JL framework* that encompasses the shareable parts in order to reduce the JL application development efforts. The JL framework includes the middleware from the first category listed above and some common modules for BF-JL search from the second category, so that developers can easily develop JL applications on top of this framework. The middleware is described in Subsection 3.1 and the common modules for BF-JL search are described in Subsection 3.2. Subsection 3.3 discusses issues such as transposition handling and program scalability during the implementation of the JL framework.

3.1 Middleware

Our JL system leverages the Computer Game Desktop Grid (CGDG), which was developed by Wu et al. (Wu, Chen, Lin *et al.*, 2009). A desktop grid is a network platform that can harvest unused resources over a large number of personal computers (Foster and Kesselman, 2003). In practice, a desktop grid can be generalized to include other kinds of computing machines. An important feature of desktop grids is that the computing resources can be even donated by volunteers. Consequently, a grid may often consist of heterogeneous computing resources, where many different types of machines are involved; moreover, connections/disconnections to the grid may also occur spontaneously.

CGDG is able to function as a JL system since it supports the connection-oriented push-mode streaming links between the broker and clients, as well as between the broker and workers. This enables the broker to push control messages (e.g., initiating new jobs, passing data as input) to designated workers in a timely manner. It can also allow the broker to push messages (e.g., requesting jobs, returning job results) back to the client. These two capabilities, with the above mentioned examples, make up a complete JL application. The communication operations between a client and the CGDG are described in Subsection 3.1.1. A simple JL application is illustrated in Subsection 3.1.2.

3.1.1 Communication with the JL System

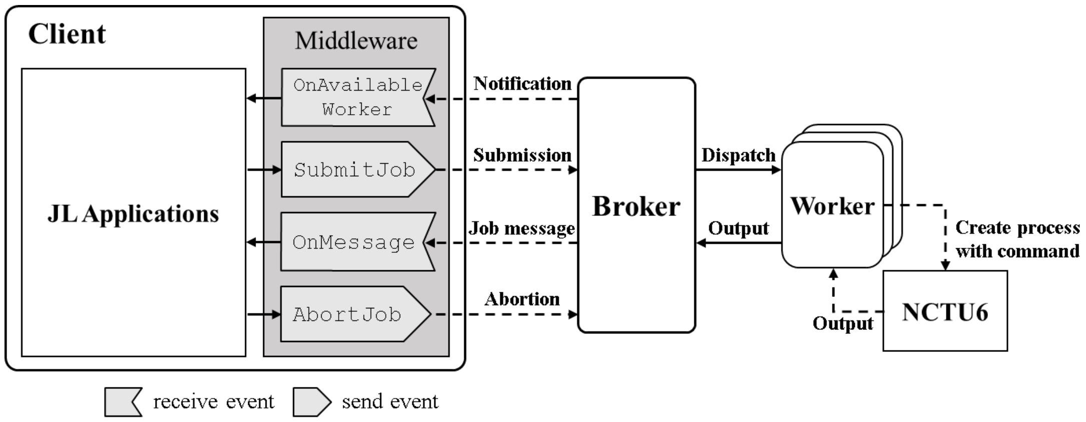


Figure 4: The middleware and the communication operations between the client and CGDG.

This subsection describes the communication protocol between clients and our JL system CGDG, based on the model of JL computing described in Subsection 2.1. The following four messages, also depicted in Figure 4, are most vital to the operation of a JL application.

- **Submission**: A client creates a job by formulating command, which may for example take the form "`program_name` *`args positions`*". The client then sends a job dispatch request for the job with an associated job ID to the JL system via the function **`SubmitJob`**, as illustrated in Figure 4 After receiving the request, the JL system dispatches it to a worker once any becomes available. The worker then creates a new process for the job by invoking the game-playing program via the command provided by the client.
- **Abortion**: Whenever a job is no longer interesting to the client, it can send a job abort request by providing a job ID to the JL system via the function **`AbortJob`**. For example, a proven move indicates that jobs exploring other candidate moves can be aborted. In (Chen *et al.*, 2015), job aborts are used to improve performance.
- **Job message**: In our JL system, the results of job processes are communicated via its standard output, which is always sent back to the client. The client can then extract results from this event message. The event message is handled by the function **`OnMessage`**.
- **Notification**: The JL system sends a notification to the client as soon as a worker becomes available for execution. The client will then select jobs to submit if possible. Note that with CGDG, the JL system will not send another notification to a client before it receives the next job submission from that particular client. The event message is handled by the function **`OnAvailableWorker`**.

In addition to the above, the communication protocol also supports other event messages such as job exceptions (receive), job standard input (send), job information and worker status requests (send), acknowledgements (send/receive), etc. For the remainder of this article, we mainly consider the above four when illustrating communications with the JL system.

3.1.2 A Simple JL Application

This subsection illustrates the operation of the middleware through a simple JL application called *AI competition*. Consider the situation where two different game-playing programs need to compete against each other, typically for strength analysis. We use a set of initial openings that are commonly played by experts as the benchmark. Let us consider exactly one opening in this benchmark for simplicity. One program will play first, while the other plays second. For each move to play, the client submits to a worker the following information as a job: the position to move from and the program that needs to play. Once a worker in the JL system receives this job, it invokes the corresponding program to generate the best move to play. When the worker is done with the job, the result will be sent back to the client via standard output, which the client parses. The client adds the newly played move to the game tree, and then submits the next move as the next job. Each program plays alternately until the game has concluded, where the outcome must be a win, loss, or draw. The two programs then switch roles, where the first program will now play second. Again, the game is played to completion, and the two game results are recorded. The two games played from this opening position are independent, and can be played in parallel. In fact, this is true for all openings in the set.

To implement AI competition on top of the middleware, we need to implement the two event handlers, **`OnAvailableWorker`** and **`OnMessage`**. The handler **`OnAvailableWorker`** first searches from the initial opening position down to a leaf, then tries to mark the leaf as running before submitting a job via the function **`SubmitJob`**, which generates a move on the position corresponding to the leaf. The handler is designed to select only leaves that are not running any jobs. After the job is finished and the best move is returned via standard output, the handler **`OnMessage`** unmarks the leaf, parses the returning message to determine the move to play, and adds to the tree a new leaf node corresponding to the move. If the status for the new leaf is a win, loss, or draw, AI competition records the result and no further jobs are generated for that particular game. Once all openings in the benchmark have been completed, AI competition finishes by tallying the results into a report.

3.2 BF-JL SEARCH

While development efforts can be reduced by facilitating JL system communications through a shared middleware, BF-JL applications still share many common behaviours such as tree traversal and position proof updates, which we will discuss in Subsection 3.2.1. In order to simplify BF-JL application development further, we describe common modules for BF-JL applications in Subsection 3.2.2, and then we abstract the specific behaviours for different BF-JL applications by using handlers and integrators in Subsection 3.2.3.

3.2.1 BF-JL Applications

All BF-JL applications require maintaining JL search trees, so a generic tree that is shared for different games and different BF-JL searches can be designed. Of the four phases, selection, pre-update, execution, and update, operations such as tree traversal and expansion can be reused entirely. In addition, postponed sibling generation, flag policies (Wu *et al.*, 2013) that are used in the pre-update phase, and AND-OR tree proof backups can also be reused.

Different BF-JL searches may require specific behaviour for some of the common operations listed above. For example, JL-PNS uses PN/DN values to locate MPNs to submit jobs; similarly, PN/DN values are modified in the pre-update and update phases. In addition, JL-UCT uses UCB values to select leaf nodes to submit jobs; visit counts and win rates are modified in the pre-update and update phases.

We now consider BF-JL applications in regards to the pairing between game-playing programs and BF-JL searches. For the first case, where different BF-JL applications share the same game-playing program while using different BF-JL searches, we need to specify the specific behaviour for each BF-JL search in regards to the game-playing program. For example, for a Connect6 BF-JL application that uses NCTU6, the returning game statuses, such as B2, W3, etc., need to be translated to specific PN/DN values for JL-PNS and specific win rates for JL-UCT as shown in Table 1. For the second case, where different BF-JL applications share the same BF-JL search among different game-playing programs, each program needs to specify the corresponding behaviour for the search. For example, the three programs, NCTU6 for Connect6, CGI (CGI-LAB, 2015a) for Go, and MoHex (Huang, Arneson, Hayward *et al.*, 2014) for Hex, can be paired with JL-UCT to create three separate BF-JL applications. Since JL-UCT uses win rates during the search, different programs need to provide JL-UCT with their own win rates. While the two MCTS-based programs, CGI and MoHex, return win rates directly, NCTU6 again needs to translate game statuses into win rates, as shown in Table 1.

Considering the above, we now separate the duties of BF-JL application developers into three different roles. Up to this point, all BF-JL applications are developed by a single party: the *application developer* (role 1). However, since most code can be reused in BF-JL applications, only the specific behaviour described above need to be defined. The *BFS-specific code*, say JL-UCT or JL-PNS, is the responsibility of *BF-JL search developers*, also called *BFS developers* (role 2). This code can be shared in BF-JL applications using the same BF-JL search with various game-playing programs. Finally, the game-specific code for game-playing programs is the responsibility of *game developers* (role 3). This code can be shared in BF-JL applications that share the same game-playing programs. Under this new division of development roles, the JL application developer only needs to integrate the game-specific code with the BFS-specific code.

3.2.2 Common BF-JL Modules for BF-JL Applications

For BF-JL applications, we summarize eight common modules on top of the middleware. The eight modules are **Initialize**, **Select**, **Dispatch**, **Pre-update**, **Handle result**, **Update**, **Is completed** and **Finalize**. The diagram for these modules is shown in **Error! Reference source not found.**. We describe each of these modules, together with their game-specific and BFS-specific operations below.

Initialize

This module sets up all initial settings and the JL search tree. Two functions are included in this module: the BFS developers initialize BFS-specific data and operations in the function `InitializeBFS`, while game developers initialize game-specific data and operations in `InitializeGame`.

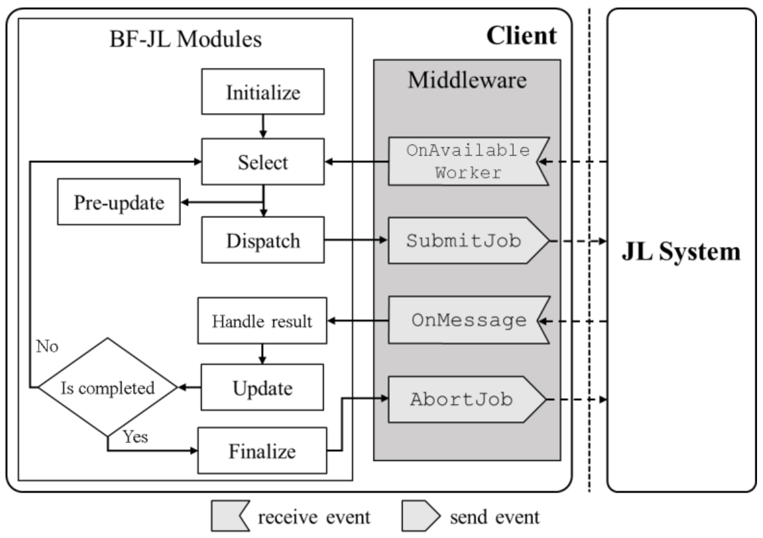


Figure 5: The modules of generic BFS algorithm.

Select

When the middleware receives a notification of idle workers, the event handler **`OnAvailableWorker`** invokes this module. The module traverses the JL search tree from the root to a leaf, based on selection policies, then selects the leaf node as a job, which is dispatched by the **Dispatch** module. Postponed sibling generation as mentioned in Subsection 2.2 is also done in this module. In the module, BFS developers (role 2) design their own selection policies in the function **`SelectBestChild`** to select the best child for each node on the path. For example, for JL-UCT the function selects the best child based on their UCB scores; for JL-PNS, this function selects the child with the smallest PN/DN values.

Dispatch

With the selected leaf node in the previous module, a job is created along with its corresponding command arguments, which is then submitted to the JL system via the middleware. Game developers (role 3) prepare job commands using the function **`PrepareJobCommand`**, which usually includes the position of the selected leaf node and a set of prohibited moves.

Pre-update

In order to prevent selecting the same node repeatedly, this module uses a pre-update policy to update BFS-specific data of the nodes along the path from the dispatched node to the root. This module implements the flag policies (Saffidine *et al.*, 2012; Wu *et al.*, 2013). In addition to the above, BFS developers (role 2) can add more policies in the function **`PreUpdateData`**. For example, the virtual-loss policy for JL-UCT (Wei *et al.*, 2014), and the virtual-equivalence policy for JL-PNS (Wu *et al.*, 2011).

Handle result

When the middleware receives job messages, the event handler **`OnMessage`** invokes this module. The module then calls a parser to handle the received messages. Since the message format depends entirely on the game-playing program, it is the responsibility of the game developer (role 3) to design game parsers so that game-specific data can be retrieved from the returning messages. A generic class named **`GameParser`** is provided so that specific game parsers can be implemented. If the parsed data includes a new move to play along with the values associated with this new node, then the module generates a node corresponding to this move using the function **`GenerateNode`**, which is game-specific, since the definition and format of a move depends on the game. After the new node is generated, this module also initializes the BFS-specific data of the node using the associated value in the returned message with the function **`InitializeNodeBFSData`**. For example, if the initialization values are mapped to the game statuses according to Table 1, and the NCTU6 worker returns the game status B3, then the PN/DN value 2/12 or a win rate of 80% will be initialized for the new node. Since the function converts game-specific data to BFS-specific data, both game developers (role 3) and BFS developers (role 2) are responsible. Alternatively, we may consider this implementation the responsibility of the JL application developer (role 1).

In order to clarify the responsibilities between developers, let us assume that the JL application developer (role 1) provides retrievers that convert the game-specific data into BFS-specific data. Thus, the BFS developer (role 2) can simply use this provided retriever to initialize the generated node without having to understand any game-specific information.

For example, assume that a BF-JL application uses JL-UCT with NCTU6. First, the BFS developer (role 2) defines the abstract class **`UctRetriever`** for JL-UCT to retrieve win rates from game parsers. Then, the JL application developer (role 1) implements a specific **`UctRetriever`**, in this case **`NCTU6UctRetriever`**, to retrieve win rates from **`NCTU6Parser`**, based on the mapping in Table 1. In practice, the JL application developer (role 1) also needs to implement an integrator to integrate the retriever with the game parser, based on the design pattern of abstract factory described in (Gamma, Helm, Johnson *et al.*, 1994). A sample segment of the above described code is shown in the Appendix.

In summary, game developers (role 3) are responsible for designing game parsers and the function **`GenerateNode`**. BFS developers (role 2) are responsible for defining BF-JL search retrievers and designing the function **`InitializeNodeBFSData`**. Finally, JL application developers (role 1) integrate them together by implementing the game and BF-JL search-specific retriever and the integrator.

Update

After new nodes are generated by the previous module, this module updates data of the nodes along the path from the new node to the root. For each visited node on the path, this module updates BFS-specific data via the function **`UpdateData`**, which is provided by BFS developers (role 2). For example, for JL-UCT, the function updates win rates, visit counts and UCB scores, while for JL-PNS, the function updates PN/DN values. In the case that a position is proven for a player, this module can automatically mark a win on the corresponding node for the player so that the opponent will avoid making that move.

Is completed

After updating all the data, this module checks whether the BF-JL search is complete. Typically, JL computing ends when the number of jobs exceeds a threshold or the root is proven. The BFS developer can define the criteria via the function **`IsCompleted`**.

Finalize

When the BF-JL search is complete, this module is invoked for finalization. For example, we may wish to abort all running jobs upon completion. In this module, the BFS developer can finalize via the function **`FinalizeBFS`**, while game developers can finalize via the function **`FinalizeGame`**.

3.2.3 Handlers and Integrators

The operations that need to be developed by the game developers, BFS developers and JL application developers are shown in Table 2. BFS developers are responsible for defining BF-JL search specific retrievers and implementing all functions in the *BFS handler*. Game developers implement game parsers and all functions in the *game handler*. JL application developers are responsible for integrators that integrate retrievers with game parsers. Other than the above handlers and integrators, common BF-JL modules can be shared by all BF-JL applications and therefore can be included in the JL framework as shown in Figure 6.

BFS Handler (role 2)	**Game Handler (role 3)**	**Integrator (role 1)**
`InitializeBFS`	`InitializeGame`	`Retriever`
`SelectBestChild`	`PrepareJobCommand`	
`PreUpdateData`	`GenerateNode`	
`UpdateData`	`FinalizeGame`	
`InitializeNodeBFSData`	`GameParser`	
`IsCompleted`		
`FinalizeBFS`		
`BfsRetriever`		

Table 2: Lists of functions and classes of handlers and integrators.

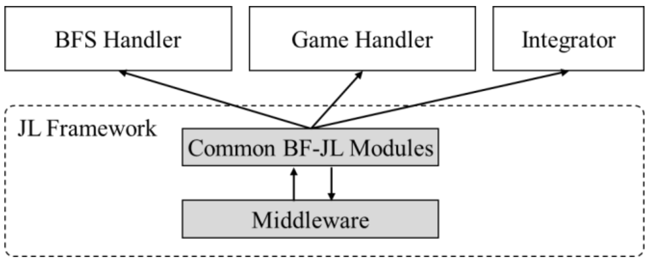


Figure 6: JL framework for BF-JL search.

3.3 Implementation Issues

This subsection discusses two important additional issues during the implementation of the JL framework, namely, how to deal with transpositions and how to design for scalability.

Transpositions

For game-playing programs, the game-tree search often encounters identical positions, which are referred to as *transpositions*, via different search paths. In order to prevent from searching the same transpositions, a common technique for depth-first search (DFS) is to save results of transpositions into transposition tables for later reuse. For BFS, the search trees are usually converted to search graphs by merging identical nodes (corresponding to the same transposition) into a single node, as mentioned in (Breuker, van den Herik, Uiterwijk *et al.*, 2001).

For JL applications, transpositions can be classified into three types: job-level, intra-job and inter-job transpositions. First, when a new node corresponding to a position is generated from the returning result of a job (from **Handle result**), the position is viewed as a *job-level transposition*. Second, inside a game-playing program or job, all transpositions which can be accessed only by the job itself are viewed as *intra-job transpositions*. Third, all transpositions which can be accessed by all other jobs are viewed as *inter-job transpositions*. The first is handled in our JL framework as described in the rest of this subsection. The second is completely handled by the game-playing programs, and is completely transparent to the BF-JL application, while the third should be carefully handled by JL application developers (role 1). A case for the third is described in (Liang *et al.*, 2015), where all inter-job transpositions are further divided into two sub-classes, according to whether the transpositions are shared among jobs in the same computation node or not.

It is critical to support job-level transpositions, since each job usually takes tens of seconds of execution. As a distributed BFS, BF-JL search can follow the use of search graphs, as described above. This can be implemented in the following two ways. The first way is to use JL search trees with links between transpositions, while the second way is to use transposition tables for all nodes and moves. An early version of our JL framework was implemented using the first way. The current version was implemented using the second by embedding the transposition tables into databases, which can greatly reduce the efforts of implementing transposition tables while also improving the scalability problem, discussed in the next subsection.

In order to further exploit job-level transpositions, we also consider positions that are identical after mirroring and rotating. However, mirroring and rotating are specific to games. For example, Go and Connect6 can rotate and mirror in eight ways, hex and Chinese chess only in two ways, and Havannah in 12 ways. In our framework, game developers can define the rotation and mirroring of transpositions on their own, but the details are omitted in this article.

Scalability

In the past, BF-JL search trees were all stored in the memory of the client process. Wu et al. (Wu *et al.*, 2011) mentioned that this method was sufficient enough to solve Connect6 opening positions which typically require hundreds of thousands of nodes. However, in the case of extremely large problems, such as solving relatively difficult 10x10 Hex openings and building 9x9 Go opening books, it is highly difficult to store all search nodes into one process.

In order to solve this problem, we propose the use of externally stored databases to maintain the whole JL search tree in our framework. We use MySQL to easily support up to billions of job-level transpositions. The whole JL search tree is stored in two tables, one for moves, and the other for positions. In this way, job-level transpositions can be easily accessed via query operations. Developers will need to modify the BFS handlers and game handlers to accommodate for databases.

In regard to efficiency, the database access overhead is relatively small compared to the job length, since each job typically takes tens of seconds. Nonetheless, there are two methods that allow more efficiency. First, we can greatly improve the database performance by using the database cache mechanism. Second, we can improve performance by using stored procedures. Stored procedures are user-defined functions that combine several database operations so that a single query may be used instead of a series of queries, which reduces the overhead of data transfer. However, this comes at a cost of higher framework software complexity and increases the effort of program maintenance.

4. CASE STUDY

This section studies cases of implementing JL applications on top of our JL framework. For simple JL applications, such as AI competition, we can directly develop on top of the middleware, as mentioned in Subsection 3.1.2. Similarly, applications with simple parallelization like finding Sudoku puzzles with the minimum number of clues (Lin, Wu and Wei, 2013) can also be applied directly. For BF-JL applications, handlers and integrators will need to be developed as described in Subsection 3.2. In the rest of this section, we will only discuss such applications.

We describe our development for the following three JL applications: one running JL-PNS with NCTU6, one running JL-UCT with NCTU6, and one running JL-UCT with CGI. Following the description in Subsection 3.2, BFS developers (role 2) simply need to implement two BFS handlers for JL-PNS and JL-UCT, called the *PNS Handler* and the *UCT Handler* respectively; game developers (role 3) simply need two game handlers for NCTU6 and CGI, called the *NCTU6 Handler* and the *CGI Handler* respectively. Once these four handlers are implemented, the application developer (role 1) simply needs to design an integrator to complete a JL application.

These handlers and integrators are briefly described as follows. First, we describe three common functions that are used in both BFS handlers.

- **`InitializeBFS`**: Initialize settings, create database tables, define database store procedures, etc.
- **`IsCompleted`**: Check whether the root of the JL search tree is proven/disproven or whether the number of jobs exceeds a user-defined limit.
- **`FinalizeBFS`**: Abort all running jobs.

The PNS handler implements a generic BFS handler with five functions.

- **`SelectBestChild`**: Given a parent, select the child with the minimum proof-number (PN) for OR nodes and the minimum disproof-number (DN) for AND nodes.

- **`PreUpdateData`**: For the selected leaf node, its PN/DN values are set to ∞/0, if virtual-equivalence policy is used. For each node on the path to the root, update PN/DN values according to the rule of PNS.
- **`UpdateData`**: For all nodes on the path to the root, update PN/DN values accordingly.
- **`PnsRetriever`**: Define the interface with two functions, **`RetrievePn`** and **`RetrieveDn`**, to retrieve PN and DN values respectively.
- **`InitializeNodeBFSData`**: Given a PNS retriever and a newly generated node, use the retriever to retrieve PN/DN values and then initialize these values for the node.

The UCT handler implements a generic BFS Handler with five functions.

- **`SelectBestChild`**: Given a parent, select the child with maximum UCB score.
- **`PreUpdateData`**: For each node on the path to the root, increase the move count by one if virtual-lost policy is used. Update UCB scores according to the rules of UCT.
- **`UpdateData`**: For all nodes on the path to the root, update win rates with the win rate of the leaf, and update UCB scores accordingly.
- **`UctRetriever`**: Define the interface with a function, **`RetrieveWinRate`**, to retrieve the win rate from a game parser.
- **`InitializeNodeBFSData`**: Given a UCT retriever and a newly generated node, use the retriever to retrieve the win rate and then initialize it for the node.

Next, we describe the only common game handler function.

- **`InitializeGame`**: Initialize game program settings and execute any additional operations that the game program may require.

The NCTU6 handler implements a generic Game Handler with three functions.

- **`PrepareJobCommand`**: Prepare the job commands, which in this case are the arguments for NCTU6. Includes information such as the position and a set of prohibited moves**.**
- **`NCTU6Parser`**: Define a game parser to parse the job messages returned from NCTU6, which contains a move and its game status.
- **`GenerateNode`**: Given a parent node and a NCTU6 parser, generate a new child node in the JL search tree and initialize game data of the node using the returned game status from NCTU6.

The CGI handler implements a generic Game Handler with three functions.

- **`PrepareJobCommand`**: Prepare the job commands, which includes the position and a set of prohibited moves**.**
- **`CGIParser`**: Define a game parser to parse the job messages returned from CGI, which contains a move and its win rate.
- **`GenerateNode`**: Given a parent node and a CGI parser, generate a new child node in the JL search tree and initialize game data of the node using the returned win rate from CGI.

For the three applications, we only describe their integrators' retrievers for simplicity by three functions.

- **`NCTU6PnsRetriever`**: Implement the retriever, **`PnsRetriever`**. Given a NCTU6 parser, both **`getPn`** and **`getDn`** convert games statuses to PN and DN values as described in Table 1.
- **`NCTU6UctRetriever`**: Implement the retriever, **`UctRetriever`**. Given a NCTU6 parser, **`getWinRate`** converts games statuses to win rates as described in Table 1.
- **`CGIUctRetriever`**: Implement the retriever, **`PnsRetriever`**. Given a CGI parser, **`getWinRate`** directly return the win rate from the parser.

Component	Lines of Code
JL Framework	13068
Middleware	6369
Common BF-JL Modules	6699
PNS Handler	654
UCT Handler	744
NCTU6 Handler	361
CGI Handler	282
NCTU6-PNS Integrator	227
NCTU6-UCT Integrator	179
CGI-UCT Integrator	135

Table 3: The number of lines of code for each component in the JL client.

The numbers of lines of code for the above JL application components, as well as the JL framework are listed in Table 3. The handlers and integrators only take hundreds of lines of code, which is much smaller than that for the JL framework. This shows that the JL framework greatly reduces the efforts of developing JL applications.

The BF-JL searches and games that we have developed are shown in Table 4. Using the framework, we developed a JL application with JL-UCT for Hex in (Liang *et al.*, 2015) by implementing a MoHex handler and using an integrator to integrate both UCT and MoHex handlers together. From this, we can easily develop JL applications with JL-PNS for Go and Hex by simply using integrators to integrate PNS handlers with CGI and MoHex handlers together. One of our ongoing projects is to port the original JL-ABS application, marked as a star in Table 4, to the new framework.

	Tic-Tac-Toe	**Connect6**	**Go**	**Hex**	**Chinese chess**
JL-UCT	V	V	V	V	
JL-PNS	V	V			
JL-ABS					*
AI competition	V	V	V	V	V

Table 4: JL applications. Completed applications are marked with 'V', while * is work in progress.

A console only version of the JL framework is available for download at (CGI-LAB, 2015b).

5. CONCLUSION

This article abstracts and develops a general JL search framework so that JL application developers can easily develop JL applications. The framework includes two parts, the middleware and common BF-JL modules. First, we develop the middleware by providing functions for send events and by defining handlers for receive events, so that JL applications can be easily built on top of the middleware. Second, we extract common BF-JL modules and define handlers and integrators so that developers can easily develop BF-JL applications using these common modules. Furthermore, we classify the handlers into BFS handlers (role 2) and game handlers (role 3). BFS developers who are responsible for the development of BF-JL searches, such as JL-PNS or JL-UCT, develop all BFS-specific operations into BFS handlers. Game developers who are responsible for the operations specific for games or game-playing programs, such as Connect6 and NCTU6, develop all game-specific operations into game handlers. Since the responsibilities of developers are made clear and separate, BFS and game developers can work independently. The application developer (role 1) completes the BF-JL application by integrating the BFS handler with the game handler. This includes the mapping of game program outputs with search data (e.g., game status to PN/DN pairings).

Our case studies demonstrate that efforts were greatly reduced by using this framework. In the case studies, only hundreds of lines of code are required for developments of JL applications, such as JL-PNS for Connect6, JL-UCT for Go and Connect6. In contrast, the code for JL framework requires above ten thousand lines of code.

Some ongoing projects make full use of the JL framework, including: the construction of a large scale opening book for 9x9 Go; solving 6x6 Go; solving 7x7 killall-Go. For these projects, we will investigate more challenging issues such as scalability, efficiency, the graph-history-interaction problem.

ACKNOWLEDGEMENTS

The authors would like to thank the National Center for High-performance Computing (NCHC) for computer time and facilities, and thank the Ministry of Science and Technology of the Republic of China (Taiwan) for financial support of this research under contract numbers MOST 104-2221-E-009-127-MY2, and MOST 104-2221-E-009-074-MY2.

6. REFERENCES

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands.

Allis, L. V., van den Herik, H. J. and Huntjens, M. P. H. (1996). Go-Moku Solved by New Search Techniques. *Computational Intelligence*, Vol. 12, No. 1, pp. 7-23.

Allis, L. V., van der Meulen, M. and Van Den Herik, H. J. (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91-124.

Auer, P., Cesa-Bianchi, N. and Fischer, P. (2002). Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, No. 2-3, pp. 235-256.

Björnsson, Y. and Finnsson, H. (2009). Cadiaplayer: A Simulation-Based General Game Player. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 1, No. 1, pp. 4-15.

Borsboom, J., Saito, J.-T., Chaslot, G. and Uiterwijk, J. W. H. M. (2007). A Comparison of Monte-Carlo Methods for Phantom Go. *Proc. 19th Belgian–Dutch Conference on Artificial Intelligence–BNAIC, Utrecht, The Netherlands*.

Breuker, D. M., van den Herik, H. J., Uiterwijk, J. W. H. M. and Allis, L. V. (2001). A Solution to the GHI Problem for Best-First Search. *Theoretical Computer Science*, Vol. 252, No. 1, pp. 121-149.

Brockington, M., Schaeffer, J. (2000). APHID: Asynchronous Parallel Game-Tree Search. *Journal of Parallel and Distributed Computing*, Vol. 60, pp. 247-273.

CGI-LAB. (2015a). "Introduction to Go Programs Developed at CGI Lab." 2015, from http://aigames.nctu.edu.tw/~icwu/aigames/CGI.html.

CGI-LAB. (2015b). "Github: Cgi-Lab/Job-Level Framework." 2015, from https://github.com/CGI-LAB/Job-Level-Framework/.

Chaslot, G. J. B., Winands, M. H. M. and van den Herik, H. J. (2008). Parallel Monte-Carlo Tree Search. *Computers and Games*. (eds. H. J. van den Herik, X. Xu, Z. Ma and M. H. M. Winands), Vol. 5131, of *Lecture Notes in Computer Science*, pp. 60-71, Springer Berlin Heidelberg.

Chen, J.-C., Wu, I.-C., Tseng, W.-J., Lin, B.-H. and Chang, C.-H. (2015). Job-Level Alpha-Beta Search. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 7, No. 1, pp. 28-38.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *5th International Conference on Computers and Games*, Turin, Italy, Springer Berlin Heidelberg.

Enzenberger, M. and Müller, M. (2010). A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. *Advances in Computer Games*. (eds. H. J. van den Herik and P. Spronck), Vol. 6048, of *Lecture Notes in Computer Science*, pp. 14-20, Springer Berlin Heidelberg.

Enzenberger, M., Müller, M., Arneson, B. and Segal, R. (2010). Fuego—an Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 2, No. 4, pp. 259-270.

Feldmann, R. (1993). Game Tree Search on Massively Parallel Systems. PhD Thesis, University of Paderborn.

Foster, I. and Kesselman, C. (2003). The Grid 2: Blueprint for a New Computing Infrastructure, Elsevier.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education.

Gelly, S. and Silver, D. (2011). Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, Vol. 175, No. 11, pp. 1856-1875.

Gelly, S., Wang, Y., Munos, R. and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical Report RR-6062m Vol. 32, pp. 30-56.

Hoki, K., Kaneko, T., Kishimoto, A. and Ito, T. (2013). Parallel Dovetailing and Its Application to Depth-First Proof-Number Search. *ICGA Journal*, Vol. 36, No. 1, pp. 22-36.

Huang, S.-C., Arneson, B., Hayward, R. B., Müller, M. and Pawlewicz, J. (2014). Mohex 2.0: A Pattern-Based MCTS Hex Player. *Computers and Games*, pp. 60-71, Springer, Heidelberg, Germany.

Kaneko, T. (2010). Parallel Depth First Proof Number Search. *AAAI*, pp. 95-100.

Kishimoto, A. and Kotani, Y. (1999). Parallel and/or Tree Search Based on Proof and Disproof Numbers. *5th Games Programming Workshop*.

Kishimoto, A. and Mueller, M. (2005). Search Versus Knowledge for Solving Life and Death Problem in Go, *AAAI*, pp. 1374-1379.

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293-326.

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: Ecml 2006*, pp. 282-293, Springer.

Liang, X., Wei, T. and Wu, I.-C. (2015). Solving Hex Openings Using Job-Level Uct Search. *ICGA Journal*, Vol. 38, No. 3, pp. 149-164.

Lin, H.-H., Wu, I. and Wei, T. (2013). On Specific 17-Clue Sudoku Puzzles. *ICGA Journal*, Vol. 36, No. 3, pp. 131-138.

Liu, H.-Y., Wu, I.-C., Liao, T.-F., Kang, H.-H. and Chen, L.-P. (2013). Software Framework for Generic Game Development in Cgdg. *Advances in Intelligent Systems and Applications-Volume 1*, pp. 219-229, Springer, Heidelberg, Germany.

Manohararajah, V. (2001). Parallel Alpha-Beta Search on Shared Memory Multiprocessors. Master's thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, Canada.

Marzetta, A. (1998). ZRAM: A Library of Parallel Search Algorithms and Its Use in Enumeration and Combinatorial Optimization. PhD Thesis, ETH Zurich.

Nagai, A. (2002). Df-pn Algorithm for Searching AND/OR Trees and its Applications, PhD Thesis, University of Tokyo.

Pawlewicz, J. and Hayward, R. B. (2014). Scalable Parallel Dfpn Search. *Computers and Games*, pp. 138-150, Springer, Heidelberg, Germany.

Romein, J. (2001). Multigame - An Environment for Distributed Game-Tree Search, PhD Thesis, Vrije Universiteit.

Saffidine, A., Jouandeau, N. and Cazenave, T. (2012). Solving Breakthrough with Race Patterns and Job-Level Proof Number Search. *Advances in Computer Games*, pp. 196-207, Springer, Heidelberg, Germany.

Saito, J.-T., Winands, M. H. M. and van den Herik, H. J. (2010). Randomized Parallel Proof-Number Search. *Advances in Computer Games*, pp. 75-87, Springer, Heidelberg, Germany.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. and Sutphen, S. (2007). Checkers Is Solved. *Science*, Vol. 317, No. 5844, pp. 1518-1522.

Van den Herik, H. J., Uiterwijk, J. W. H. M. and Van Rijswijck, J. (2002). Games Solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, No. 1, pp. 277-311.

Van den Herik, H. J. and Winands, M. H. M. (2008). Proof-Number Search and Its Variants. *Oppositional Concepts in Computational Intelligence*, pp. 91-118, Springer, Heidelberg, Germany.

Van Lishout, F., Chaslot, G. and Uiterwijk, J. W. H. M. (2007). Monte-Carlo Tree Search in Backgammon. *Computer Games Workshop*, pp. 175-184.

Wei, T.-H., Wu, I.-C., Liang, C.-C., Chiang, B.-T., Tseng, W.-J., Yen, S.-J. and Lee, C.-S. (2014). Job-Level Algorithms for Connect6 Opening Position Analysis. *Computer Games*, pp. 29-44, Springer.

Winands, M. H. M., Uiterwijk, J. W. H. M. and van den Herik, J. (2003). Pds-Pn: A New Proof-Number Search Algorithm. *Computers and Games*, pp. 61-74, Springer.

Wu, I.-C., Lin, H.-H., Lin, P.-H., Sun, D.-J., Chan, Y.-C. and Chen, B.-T. (2011). Job-Level Proof-Number Search for Connect6. *Computers and Games*, pp. 11-22, Springer.

Wu, I.-C., Lin, H.-H., Sun, D.-J., Kao, K.-Y., Lin, P.-H., Chan, Y.-C. and Chen, P.-T. (2013). Job-Level Proof Number Search. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 5, No. 1, pp. 44-56.

Wu, I.-C. and Lin, P.-H. (2010). Relevance-Zone-Oriented Proof Search for Connect6. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 2, No. 3, pp. 191-207.

Wu, I., Chen, C., Lin, P.-H., Huang, G.-C., Chen, L.-P., Sun, D.-J., Chan, Y.-C. and Tsou, H.-Y. (2009). A Volunteer-Computing-Based Grid Environment for Connect6 Applications. *Computational Science and Engineering, 2009. CSE'09. International Conference on*, IEEE.

Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H. and Ishikawa, Y. (2011). Scalable Distributed Monte-Carlo Tree Search. *Fourth Annual Symposium on Combinatorial Search*.

7. APPENDIX

The following segments of code (in C++) illustrate the way that integrator for PNS and NCTU6 combines the `PnsRetriever` with `NCTU6Parser`. And the usage of retrievers for function `InitializeNodeBFSData`.

```
class PnsRetriever: public BfsRetriever // defined by JL-PNS
developer
{
public:
  virtual double RetrievePn() = 0;
  virtual double RetrieveDn() = 0;
};
// PNS handler
InitializeNodeBFSData(Node* node, BfsRetriever *bfsRetr) {
  PnsRetriever *pnsRetr= (PnsRetriever*)bfsRetr;
 ...
}

class UctRetriever: public BfsRetriever // defined by JL-UCT
developer
{
public:
  virtual double RetrieveWinRate() = 0;
};
// UCT handler
InitializeNodeBFSData(Node* node, BfsRetriever *bfsRetr) {
  UctRetriever *uctRetr= (UctRetriever*)bfsRetr;
 ...
}

class NCTU6Parser: public GameParser // defined by NCTU6 developer
{
public:
  string GetGameStatus() { ... };
  ...
};

class NCTU6PnsRetriever: public PnsRetriever
{
private:
  NCTU6Parser* m_nctu6Parser;
  ...
public:
  NCTU6PnsRetriever(NCTU6Parser* nctu6Parser);
  ...
  double RetrievePn() { ... }
  double RetrieveDn() { ... }
};
// NCTU6PnsRetrieverFactory
class NCTU6PnsIntegrator: public Integrator
{
public:
  BfsRetriever* Retriever(GameParser* gameParser) {
    return new NCTU6PnsRetriever((NCTU6Parser*)gameParser);
  }
}
```

Table 5: A segment of program of parsers, retrievers and the integrator.