Vol.2018-AL-168 No.2
2018/5/25

Othello Font

Amanj Khorramian[1,a)] Tomoko Taniguchi[3,b)] Takeaki Uno[2,c)] Ryuhei Uehara[3,d)]

Abstract: Othello, also known as Reversi, is a quite well known strategy board game for two players on a board of size 8×8. The set of all reachable patterns is not yet known for this game. In this paper, we finally obtain all reachable patterns on 5×6 board by developing nontrivial algorithm on a supercomputer. We observe the scale of complete search-tree is big even for a board of size 5×6 and parallelize a distributed frontier search using shared memory to reach the final depth of the tree. To reduce the memory requirement, the tree is horizontally compressed using a novel method, and the frontiers are maintained in a novel data-structure. Moreover, an efficient number system is proposed and utilized for representing the states of the game, and a symmetry of the states is applied during the search. We assume that the board and pattern are rotation symmetry, but we assume that the mirror symmetry gives the different pattern. Eventually, the whole tree is traversed in 2 hours by visiting 257,387,474,170 different states using random access memory shared among 576 processing cores. We aim to find specific font patterns among the states of the final depth. However, 83,175,694 of the states are located at the final depth, at which we start looking for font patterns. Before that, a set of 96 characters of size 6×6 is binarized, and their (5×6)-compatible patterns are taken for lookup by considering all possible symmetries. In this way, a font of 96 characters is designed.

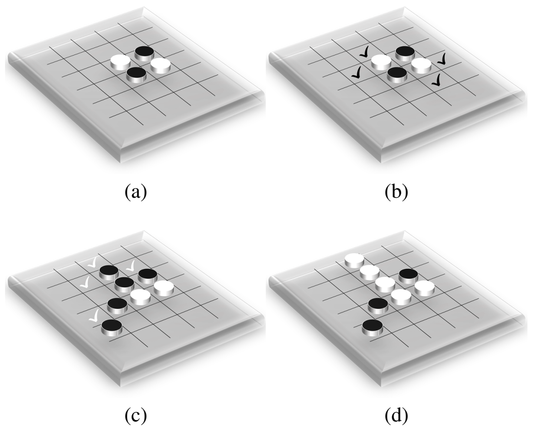

Fig. 1 Some examples of states for a 5×6 board of Othello. (a) An initial state of the board. (b) Choices of the player with black color in a state. (c) Choices of the player with white color in a state. (d) A state generated from (c), when the white player plays in such a way that maximizes the number of white disks.

1. Introduction

Othello (also known as Reversi) is a game on an $M \times N$ board, played by two players with $M \times N$ disks. The disks are colored diversely on each side using two colors, and each player is assigned one of the colors. We consider the colors of black and white with binary representations of 1 and 0, respectively. Figure 1 shows some states of a board of 5×6 for this game.

Four of the disks are initially located at a center region of the board, as shown in Figure 1(a). Each player puts a new disk with his/her color faced up, on an empty spot such that a straight line of another color should be bounded by the new color in both endpoints. The checkmarks of Figure 1(b) show the choices of putting a new disk for the player with black color, and Figure 1(c) shows the choices of player with white color. The players act in turns, but if one player has no choice, then s/he skips her/his turn. As for the example of Figure 1(c), if the player selects the choice that maximizes the number of flips from black to white, then the state of the board would change to Figure 1(d). At the end of the game, the player who has more disks of her/his color upwards on the board wins the game.

This game at this form with the original size of 8×8 was reinvented less than half a century ago. It has been popular so that some international tournaments are actively being held in several mainlands [10], [16]. There exist studies for a win-loss decision of the game [4], [16] as well as some experimental tricks and computational strategies [5], [12] of playing. We do not know any studies looking for specific patterns of the board, but it appeared of interest to design typefaces based on open problems [2], [13], resulting in several puzzle fonts published on the web [3], [14]. Following this context as well as inspiring by a talk entitled "FUN with FONTS" [2] at the 7th FUN, 2014, we are inclined to a cooperative approach of the Othello game so that a complete breadth-first search is conducted towards reaching the final depth of the search tree. The original size of the game has a very large state space [1]. Our selection is a 5×6 board of the game, since it is somehow approaching the size of 6×6 which remains challenging and possesses digital distributions in trending markets.

1 University of Kurdistan
2 National Institute of Informatics
3 Japan Advanced Institute of Science and Technology
a) khorramian@gmail.com
b) tomoko-t@jaist.ac.jp
c) uno@nii.jp
d) uehara@jaist.ac.jp

From a theoretical point of view, given an arbitrary position of the Othello game played on an $n \times n$ board, from generalized geography played on bipartite graphs with maximum degree 3, the problem of determining the winner is shown to be PSPACE-complete [6]. Othello computer programs have easily beaten the humans in the 1980s, and the human champion was lost in 1997, beaten by Logistello, which may have presented the strongest Othello player program, focused on several techniques and approaches [11]. In a strong Othello program, the key idea is pruning of the search tree. Precisely, the algorithm estimates/evaluates each pattern and gets rid of it if it is not valuable anymore. Using this technique, 6×6 Othello was investigated, and it had been found that the second player wins on 6×6 Othello [15]. With this background, the number of possible states is approximately estimated, but not yet solved. This is our motivation for research.

The initial pattern is determined in the official Othello game, and in our 5×6 board, we have two different choices by considering rotation symmetry. Assuming that the mirror symmetry gives different patterns, we fix an initial pattern. By this assumption, the states in the first three depths of the corresponding search tree (i.e., depths 0, 1, and 2) are shown in the first, the second, and the third rows of Figure 2, respectively.

During the search, we maintain only the frontiers [8] in a data structure similar to a double-ended queue [7], which is being utilized in both ends. For reducing the memory usage, we sort and store the difference values rather than the actual values of state representatives. This method dramatically compresses the search tree. To represent each state of the board, we invent a number system of mixed-base digits. This number system suits the inherent properties of Othello board and better fits primitive data types in common programming languages. In addition, this system bounds the values of states from below and above, yielding a high compression ratio in overall. We keep unchanged, the compressed structure of data while distributing and combining the states using 576 processing cores. Note that there are states that appear several times if the symmetry of the board is not considered. Moreover, for finding the font patterns, we consider both characters of white color with black background and vice versa. These considerations of symmetry reduce the size of the tree and benefits to search more rapidly, though it does not disaffect the final pattern search. The complete search tree is traversed in one pass, and the number of states as a function of depth is reported. If nobody can move, the game is over, and 5,811,761 such states are found.

2. Shifted Queue

Frontier search [8] is originally proposed as a memory-efficient framework for breadth-first search. In this framework, the states which are already expanded do not need to be kept in memory anymore. The key point for efficient search is how to avoid generating redundant states. During the search procedure, after generating all possible states from a given state s by one operation, we say s is expanded. For example, after generating all four cases of the state shown in Figure 1(b), we say the state is expanded. In the Othello case of study, the representation of each state includes the necessary fields, thus there is no chance for a state in one depth to be regenerated in later depths. Therefore, we maintain only the generated nodes, called the frontiers in memory. For an efficient maintenance, we introduce a novel data structure named *shifted queue*.

The frontiers are stored in a simple array, called A. Inside the array, we consider a subarray S which holds only the frontiers of the current depth of the search. This means the rest of frontiers, i.e. the generated nodes corresponding to the next depth, are held outside S but still inside A. By an operation of *dequeue*(), the data items are taken and removed from the left side of S, which is kept as the rightmost subarray of A. Each generated node from the item would be inserted in the leftmost empty cell of A. See Figure 3.

When all items are removed from S, it means the search of current depth is finished. At this point, before starting the search process of next depth, all inserted nodes are assigned to S, then right after that, S is shifted to the rightmost side of A. The operation of $shift()$ is done as follows. In the order of right to left, the first item of S moves to the first empty cell of A. Then, the second item moves to the second place, and so on. See Figure 4. Note that the moving of data items must be done from right to left to avoid data collisions.

Initially, the search is started from the state shown in Figure 1(a). At this starting point, it is the only state in S, located at the leftmost side of A. After that, the search procedure runs in the following loop of three repetitive steps while S is not empty:

- The set S is shifted from the left to the right region of A. See Figure 4.
- For each state s in S from left to right, all generated nodes from s are stored at the leftmost empty entry of A, and s is removed from S. See Figure 3.
- Now S is empty, hence it is newly assigned to contain all states which are stored at the left side of A.

By applying this structure of data, we guarantee no waste of memory. That is because by assigning the maximum number of necessary states needed to be kept in memory as the size of A, the array is capable to contain all of them without losing any data. We suggest naming shifted queue for this data structure since S works as a simple queue, and it is obviously shifted in memory after each stage.

During the second step of the procedure mentioned above, we apply an additional idea to save more memory as follows. If the entry index of memory for the leftmost empty cell of A equals to the index of the leftmost state in S, it means there is no space for maintaining additional generated stated in A. In this case, we *relax* A by removing redundant states (as well as symmetric states) of A before the next node expansion. Our data structure of shifted queue has no advantage in memory over the data structure of double-ended stack. But later in this paper, we discuss how to compress shifted queue for saving even more memory, so that our method of compression is not applicable to the double-ended stack. In addition, we would distribute shifted queue for the purpose of parallelism to save also the time.

3. Symmetry

We suppose the initial state of the game to be either in the

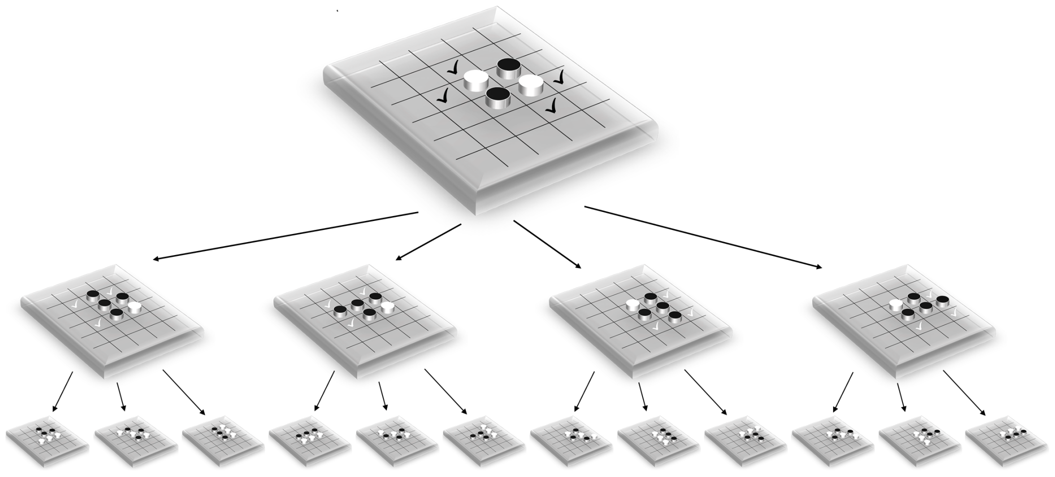

Fig. 2 The first three depths of the search tree for Othello 5×6

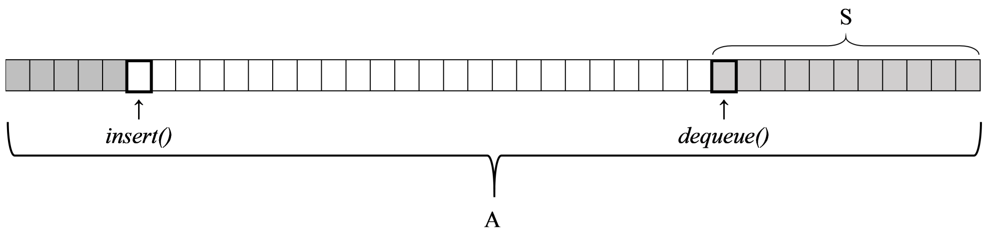

Fig. 3 An example to represent the data which is subject for *dequeue*() from a shifted-queue S. The dequeued item is expanded, and each one of its generated nodes is inserted in the leftmost empty cell of A as shown.

original form as shown in Figure 1(a) or rotated by 180 degrees and mirrored. Considering this symmetry, each state may have two variants which are identical if we consider both transformations shown in Figure 5 (starting from the up-right corner and circularly reach the center region counterclockwise, or starting from the down-right corner and circularly reach the center region clockwise). This consideration helps to reduce the memory usage while searching the entire search tree. To serve this purpose, we map each state to integers by applying two transformations, but only the smallest integer is stored in memory. The choice of transformation is attached to the integer for the retrieval of the actual state.

The reason for our circular choice of transformation has strong relation with our compression method, which is discussed later. Since the game starts from a center region of the board, the possibility of change in values of this region is more likely than the borders. For this intuitive reason, we assign higher positions to the farther cells from the center. The details of this mapping are discussed in the next section.

In the final stage of pattern finding, we reconsider these symmetries in addition to their similar variants starting from up-left and down-left corners. During the search, we do not consider these two additional transformations, because of yielding less ratio for our method of compression. Again, for the pattern lookup, it is also taken into account that anyone of the players can start the game in our consideration.

4. Othello Number System

Each spot on the board is either vacant or filled by a white or black disk. Therefore, a ternary number system is trivial to represent each state [9]. We do not know a bijection for this problem, but the number of arrangements is reduced from $3^{M\times N}$ to $2^4 \times 3^{M\times N-4}$, or even to $2^5 \times 3^{M\times N-5}$ in some cases, by utilizing specific properties of Othello. Therefore, rather than a 3-base system of numbering, we introduce a mixed-base number system of bits and trits. There are $M \times N$ spots, and four of the center region have only two variants in either white or black, and the other $M \times N - 4$ spots have three variants in white, black, or vacant. In total, we have $2^4 \times 3^{M\times N-4}$ combinations. Firstly, the places of the digits are fixed. The choice of transformation is attached to the number for state retrieval. When the state is retrieved from the number, we need to know the transformations applied for mapping. It is considered as an additional single digit with a base equal to the number of different transformations. See Figure 6.

Based on this mapping system, the digit corresponding to the corner spot gets the highest place of 30, the center spots get the places of 4, 3, 2, and 1, and the choice of transformation gets the lowest place of 0 for mapping. Consequently, each number occupies $\lceil \lg(2 \times 2^4 \times 3^{26}) \rceil = 46$ bits to store in memory and better fits in primitive data types for programming purposes. Besides, this *Othello number system* makes the range of mapped numbers $[0, T \times 2^4 \times 3^{M\times N-4} - 1]$ tighter than $[0, T \times 3^{M\times N} - 1]$ of a simple ternary number system, where T is the number of transformations applied for symmetry. Furthermore, it is more efficient also for our next purpose of compression. Note that when $M = N$ and $M \bmod 2 = 0$, the upper bound is improved to $2^5 \times 3^{M\times N-5}$, because there exists only one state in depth one by considering all symmetries of the board.

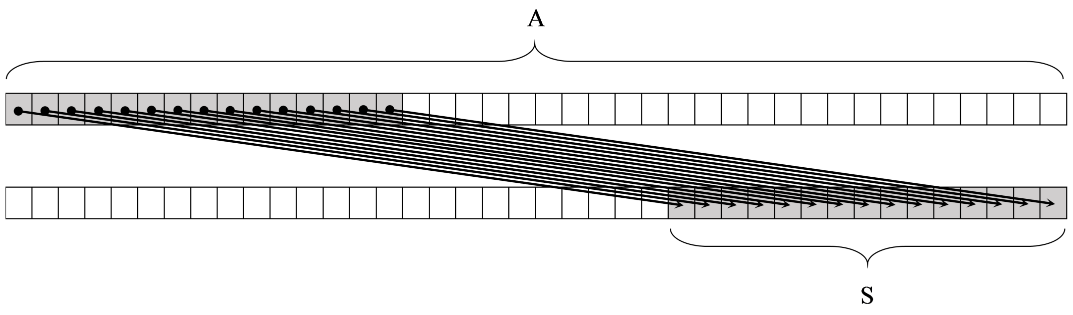

Fig. 4 An example to show the operation *shift*() of a shifted-queue S inside an array A.

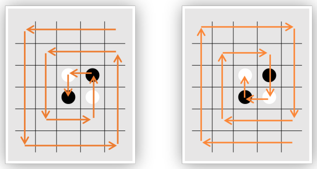

Fig. 5 Two different ways of transformation for each state

5. Gapkeep Compression

It is observed that the quotient obtained by dividing the upper bound of the range in Othello number system (i.e. $d \times 2^4 \times 3^{M \times N-4}$) by the maximum number of frontiers in the search space, needs much less number of bits than $\lceil \lg(d \times 2^4 \times 3^{M \times N-4}) \rceil$, on average. Therefore, we maintain the frontiers in both A and S sorted increasingly and store the (non-negative) difference values (i.e. gaps) between each pair of neighbors, instead of storing the actual values. This is done for all states in an array other than the leftmost state. As for the leftmost state, we stick its actual value together with the total number of states to each array for the construction of *dequeue*() operation from the shifted queue S. Although the average number of bits needed for storing the differences is much less than the number of bits needed to store the actual values, there is no guarantee that all difference values fit in a fewer number of bits. For this issue, we establish a positive and variable-length integer data type *povaleint*, which is of n_1 bits basically.

Since there is no need to store identical states, we do not keep a difference of zero, so we store d, the difference subtracted by one, to narrow the range by one value for a (very tiny) memory saving. If d is less than $2^{n_1} - 1$, then d fits to be stored only in the n_1 bits, and no more memory is needed for it. Otherwise, we break d into the couple of $2^{n_1} - 1$ and $d - (2^{n_1} - 1)$ parts. The first part is stored in the n_1 bits and we try to store the second part in the next n_2 bits of the memory, by dedicating $n_1 + n_2$ bits for d in total. If it still does not fit to store d, then we repeat the same procedure using the next n_3, or $n_3 + n_4$, ..., or $n_3 + ... + n_m$ bits of memory (see Figure 7). Therefore, the maximum number of bits to store a data element of type povaleint would be $n_1 + ... + n_m$. Here we open the optimization problem of assigning the most memory-efficient values for n_1, n_2, ..., and n_m, and establish a hardware-friendly setting of $n_i = 8 \times 2^{i-1}$ for $m = 4$, achieving a compression ratio of about 7.0 (above 85% of space saving).

We call this technique *gapkeep*, which is useful in the frontier search when the frontiers are represented in integers, and the upper bound of integers has a small-enough quotient to the maximum number of frontiers in the search space. Note that n_m must fit the value of $d - \sum(2^{n_i} - 1)$ for $i = 1, ..., m-1$. For an even better performance, we use an uncompressed buffer to temporarily store the generated states until it becomes full. The filled-up buffer is then compressed into A altogether. It is worth mentioning about this method of compression that, less uniform distribution of frontiers to the range of integers leads to more ratio of compression since the difference values are stored in memory. Note that the mapping of Othello number system is not uniform at all, so that we next introduce a method to achieve a reasonable distribution of states while keeping the same ratio of compression.

6. Basket Distribution and Parallelization

The availability of a big memory shared among many processing cores, motivated us to parallelize the frontier search by distributing the states of the povaleint type in shifted queue towards speeding up the process. Othello number system is too far from a uniform distributing function. For such issue, we propose a near-uniform distribution method, named *basket distribution*, that is suitable for applying to the mapping systems which are naturally non-uniform. This leads to a fair parallelism among processing cores. In our method of distribution, we separately make t queues maintained by t threads. At each stage of the search process, after expanding all states and before shifting the queues, we distribute all frontiers as follows. Local frontiers of each queue are split into k parts according to their values, such that the ith part consists of the local states with values in the range $[(i-1) \times U/k, i \times U/k)$ where U is the upper bound of the state values (e.g. $2 \times 2^4 \times 3^{26}$ for our study case of Othello 5×6 number system). See the example of Figure 8.

After that, the number of states in the ith part of all queues is accumulated to a_j to determine t breakpoints for distribution. For example, the first breakpoint j_1 is set to collect and merge $a_1 + a_2 + ... + a_{j_1}$ states from the first j_1 parts of all queues for assigning them to the first shifted queue in the next stage of the search process. The second breakpoint is to collect $a_{j_1+1} + a_{j_1+2} + ... + a_{j_2}$ states, and so on. Since the nature of Othello number system provides a quite non-uniform distribution of the states, we set k large enough and use a *basket* for making the size of collections nearly equal to reach a near-uniform distribution of states. Each time, we replete the basket by collecting all states of the j parts, such that rather choosing $j-1$ does not fill up the basket. Note that the basket size is the number of all frontiers

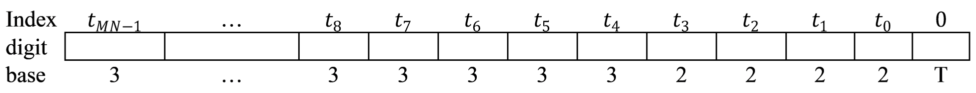
Index t_{MN-1} … t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1 t_0 0
digit
base 3 … 3 3 3 3 3 2 2 2 2 T

Fig. 6 Representation of the Othello number. The lowest position of 0 is dedicated for the digit corresponding to the transformation. t_i is the digit's position which corresponds to the ith spot of the board in the transformation, decreasingly. T is the number of transformations, applied for symmetry.

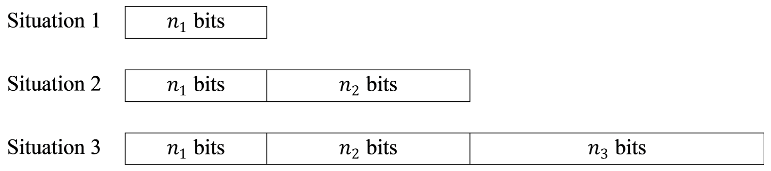

Fig. 7 In this example, each number of povaleint type may be stored in n_1, $n_1 + n_2$, or $n_1 + n_2 + n_3$ bits of memory.

over t. This approach of choosing breakpoints provides a satisfying distribution based on the quite non-uniform mapping of Othello number system. In this way, our technique of parallelization needs twice the memory size of the sequential approach. That is because we use temporary queues of the povaleint type for final distribution to avoid overlapping of data. All modules in this method work almost fair in parallel, other than the small and very rapid module of accumulation that needs a tiny portion of time for completion at each round. The technique is novel and general so that it can be utilized in every frontier search with a non-uniform distribution of mapping.

7. Binarization and Pattern Lookup

Here, we aim to import image files into `boolean` matrices. That is because we look for patterns of a font (see Figure 9) with a set of 96 characters of 6×6 pixels, which is found on the web in image format, arranged in 3 rows and 32 columns [17]. We first convert the image into a portable bitmap file. Then the file is read into 6×6 `boolean` matrices, each representing a character of the font. The characters have their last column unfilled and proper to look for on a 5×6 board. We look for all patterns while considering the symmetric patterns as well as color-negating of each character.

8. Result

We have used the interface of Open Multi Processing in C++ programming language, executed on a supercomputer. In this environment, 96 processors of Intel Xeon E5-4655 v3, each comprising 6 cores was fully utilized. Each core was assigned to a thread, so that totally 576 cores are used in parallel for two hours to complete the search of the tree. Note that we did set the size 2^{29} for arrays of `byte` data type to handle distributed shifted queues by the threads, accompanied by `long` buffers of size 2^{20}. The number of parts is set to $k = 2^4 \times 3^{12}$ in our method of distribution.

The number of states as a function of depth, together with the number of states at which the turn is skipped, and the opponent continues are shown in Figure 10. Depth 20 of the search is the widest depth. For this depth, the minimum number of bits to represent each state in gapkeep compression is calculated. The percentage of necessary bits among the entire nodes is shown in Figure 11. Furthermore, the result of basket distribution for depth 20 is shown in Figure 12.

The states of final depth are stored on disk. In the stage of font lookup, 66 exact patterns of characters and 30 similar patterns to the other characters are found as illustrated in Figure 13. In this figure, the necessary transformations for each character are specified. Note that only one pattern is reported even if more patterns are found for a specific character. In addition to the exact patterns, we find similar patterns to the characters which have no precise pattern on this board.

9. Remarks

In this paper, we introduce algorithmic methods of compression and parallelization, as well as data structural techniques of representation and mapping. These methods and techniques are implemented for conducting frontier search towards reaching the final depth of a massive state-space graph. We eventually generate the set of goal patterns and find several patterns of a character-set among them. Our method of compression suggests an optimization problem, which is left open to solve. A fun application for the gapkeep compression would be exploring a time-memory tradeoff. The proposed method of distribution is not perfectly uniform, so that one may improve it for the purpose of more accurate parallelization. It looks fun to invent a number system for a game. Note that the Othello number system is not a bijection, hence a more efficient function might be possible for the board of this game. In addition, for square boards with even side, the upper bound of Othello number system can be improved from $2^4 \times 3^{M \times N-4}$ to $2^5 \times 3^{M \times N-5}$, because there would be only one state in depth 1 by considering all symmetries. Although the data structure of shifted-queue is optimized for memory, the time needed to shift might be an area of improvement to work. During the search, we take only two transformations into account for the symmetry because the consideration of more transformations leads less ratio of compression. Hence, it would be fun to present a tradeoff between them. By the enlargement of the board (e.g., 6×6), the space would become much bigger so that we suggest establishing a phase-based search. The initial phase is done in one pass and generates the states until a reachable depth by all threads. Then, the frontiers of each individual thread are carried on by all threads in a pass to reach a higher reachable depth. After

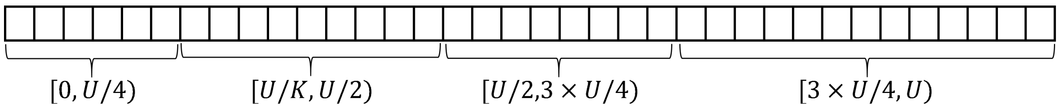

Fig. 8 An example of partitioning the local queue for four splits.

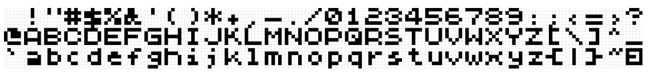

Fig. 9 The character set found [17] and used as a reference font for pattern-lookup. Characters are arranged in 3 rows and 32 columns of patterns. Each row and column includes six pixels. From left to right, row by row up to down, the patterns represent the characters ‘ ’, ‘!’, ‘”’, ‘#’, ‘$’, ‘%’, ‘&’, ‘’’, ‘(’, ‘)’, ‘*’, ‘+’, ‘,’, ‘-’, ‘.’, ‘/’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘:’, ‘;’, ‘¡’, ‘=’, ‘¿’, ‘?’, ‘@’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, ‘Z’, ‘[’, ‘\’, ‘]’, ‘ˆ’, ‘_’, ‘‘’, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’, ‘l’, ‘m’, ‘n’, ‘o’, ‘p’, ‘q’, ‘r’, ‘s’, ‘t’, ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, ‘z’, ‘’, ‘—’, ‘’, ‘˜’, and ‘©’, respectively. Note that we ignore the last column of all patterns to adapt with our 5 × 6 board for looking up, since it is not filled up.

Depth	States	Turn Skips	Rival Plays
0	1	0	0
1	4	0	0
2	12	0	0
3	51	0	0
4	204	0	0
5	993	1	1
6	4,629	0	0
7	22,880	4	4
8	114,395	7	7
9	557,539	106	106
10	2,733,070	359	329
11	12,396,346	2,791	2,754
12	54,972,198	10,331	9,873
13	213,892,489	61,982	61,121
14	783,658,713	211,950	206,406
15	2,399,717,094	1,001,948	991,410
16	6,649,274,013	2,841,158	2,801,993
17	14,842,465,438	10,027,682	9,962,281
18	28,711,051,982	22,927,812	22,787,926
19	43,297,039,709	58,079,834	57,897,571
20	53,516,410,536	104,874,332	104,646,686
21	49,327,239,781	188,837,063	188,618,642
22	34,458,121,325	261,947,202	261,744,466
23	16,656,788,055	333,070,297	332,859,774
24	5,373,298,808	339,160,512	338,932,289
25	1,004,538,211	291,725,986	291,194,066
26	83,175,694	190,658,093	186,909,984

Fig. 10 The numbers of states are shown in the second column as a function of depth. The number of states, at which the turn is skipped to the opponent, is shown in the third column. The fourth column shows the number of states, at which the opponent continues the game. Therefore, before reaching the final depth, at 5,811,761 states of the board, no player can continue the game.

the accomplishment of all passes, the second phase is done, and the next phase starts in a similar way. At the end of each pass, the nodes must be stored on disk according to our suggestion. In this way, the set of patterns in the final depth would be finally reached.

Acknowledgements

The first author expresses thanks to Japan Advanced Institute of Science and Technology for providing a visiting position and giving him access to its powerful supercomputers during the course of this project.

References

[1] Buro, M., Ontanon, S. and Preuss, M.: Guest Editorial Real-Time Strategy Games, *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 8, No. 4, pp. 317–318 (online), DOI: 10.1109/TCIAIG.2016.2601116 (2016).

[2] Demaine, E. D. and Demaine, M. L.: Fun with fonts: Algorithmic typography, *Theoretical Computer Science*, Vol. 586, pp. 111–119 (online), DOI: 10.1016/J.TCS.2015.01.054 (2015 (It was also given as an invited talk at the 7th International Conference on Fun with Algorithms (FUN 2014); Lipari Island; Italy; July 13; 2014; pages 1627.)).

[3] Erik D. Demaine: Mathematical and Puzzle Fonts/Typefaces.

[4] Federation, B. O.: Forty Billion Nodes Under The Tree (1993).

[5] Frankland, C. and Pillay, N.: Evolving game playing strategies for othello, *Evolutionary Computation (CEC), 2015 IEEE Congress on*, IEEE, pp. 1498–1504 (2015).

[6] Iwata, S. and Kasai, T.: The Othello game on an n n board is PSPACE-complete, *Theoretical Computer Science*, Vol. 123, No. 2, pp. 329–340 (online), DOI: 10.1016/0304-3975(94)90131-7 (1994).

[7] Knuth, D.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, third edition (1997).

[8] Korf, R. E., Zhang, W., Thayer, I. and Hohwald, H.: Frontier search, *Journal of the ACM (JACM)*, Vol. 52, No. 5, pp. 715–748 (2005).

[9] Lucas, S. M.: Learning to play Othello with n-tuple systems, *Australian Journal of Intelligent Information Processing*, Vol. 4, pp. 1–20 (2008).

[10] Members, U. S. O. A. C.: Upcoming OTHELLO Tournaments (2018).

[11] Michael Buro: LOGISTELLO's Homepage (2011).

[12] Nijssen, J.: Playing Othello Using Monte Carlo, *Strategies*, pp. 1–9 (2007).

[13] Oikawa, T., Yamazaki, K., Taniguchi, T. and Uehara, R.: A Peg Solitaire Font, *Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture* (David Swart Carlo H. Séquin and Fenyvesi, K., eds.), Phoenix, Arizona, Tessellations Publishing, pp. 183–188 (2017).

[14] Ryuhei Uehara: Peg Solitaire Font 5x7.

[15] Takeshita, Y., Sakamoto, M., Ito, T. and Ikeda, S.: Perfect Play in Miniature Othello, *International Conference on Genetic and Evolutionary Computing (GEC 2015)*, Springer, Cham, pp. 281–290 (online), DOI: 10.1007/978-3-319-23207-2_28 (2016).

[16] Takeshita, Y., Sakamoto, M., Ito, T., Ito, T. and Ikeda, S.: Reduction of the search space to find perfect play of 6 6 board Othello (2017).

[17] Wiki, U.: File:Charset 6x6 192x18.png - Uzebox Wiki (2012).

1 bit	2 bits	3 bits	4 bits	5 bits	6 bits	7 bits	8 bits	9-16 bits	17-32 bits
6.01%	42.62%	21.81%	10.92%	3.61%	2.01%	4.47%	2.15%	6.33%	0.06%

Fig. 11 The minimum number of bits needed for each state to be stored as difference values and the percentage of such states in the widest depth of the search tree, i.e., depth 20, for Othello 5×6. For example, 42.6% of the states need only 2 bits for storage, and more than 93.6% of states can be stored using only one byte for each. Such data would be an essential input for our open optimization problem of setting memory-efficient parameters. Note that less than 0.000004% of the states need more than 32 bits for storage. This shows the power of gapkeep compression.

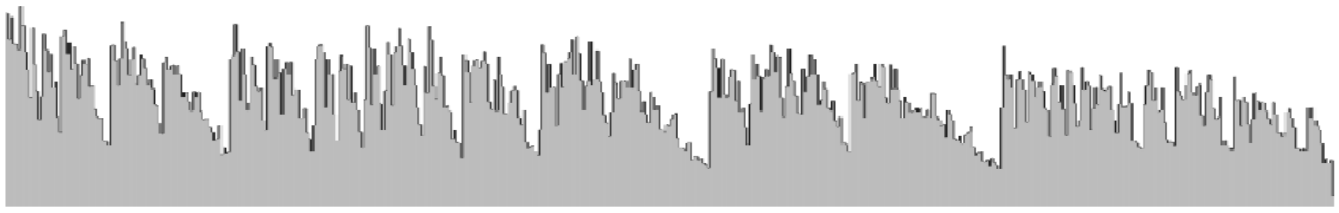

Fig. 12 The basket distribution of states among 576 threads in the widest depth of search space. The heaviest thread (i.e., 7th thread) manages 0.32% of the expanded states in this depth. Note that the original mapping of Othello number system is quite non-uniform.

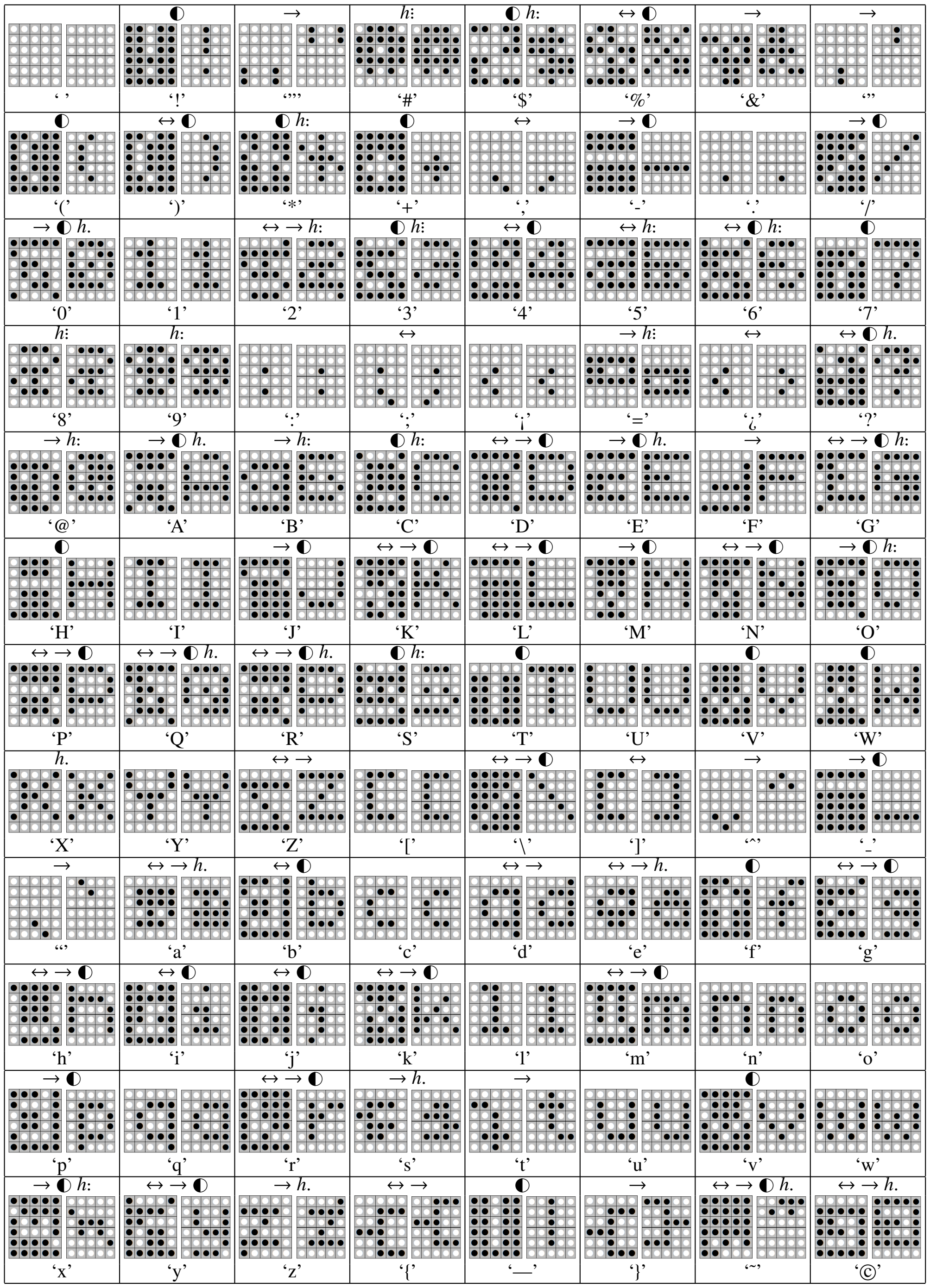

Fig. 13 Othello Font based on a character set [17]. Above each pattern, the necessary transformation(s) are shown by symbols to reach the intended character, aligned right beside. ≪→≫ stands for a rotation of 180 degrees, ≪↔≫ stands for a mirror transformation, and ≪◐≫ stands for the color exchange. In addition to 96 exact patterns, 13 patterns with the hamming distance of 1, 13 patterns with the distance of 2, and 4 patterns with the distance of 3 are found for the rest of characters. Hamming distances of 1, 2, and 3, are shown using the symbols ≪*h*.≫, ≪*h*:≫, and ≪*h*⁝≫, respectively.