アルゴリズム 79－4
（2001．7．27）

詰将棋を解くプログラムにおける効率的なハッシュの利用法について

長井 歩　　今井 浩

東京大学大学院理学系研究科情報科学専攻

〒113-0033 東京都文京区本郷 7-3-1

要旨

将棋やチェスなどの思考ゲームの探索アルゴリズムに関してはこれまでよく論じられてきた．しかし，深さ優先探索法につきものである，ハッシュの実際の実装法については殆んど論じられることはなかった．詰将棋を解くアルゴリズムの研究はこの10年の間に大きく進歩したが，ハッシュの実装については，これまでと同様に，あまり論じられることはなかった．我々は詰将棋を解くアルゴリズムの研究を行ってきて，十分な成果を挙げてきたが，これまであまり論じられることのなかった，ハッシュの実装について，この論文で論じたい．我々の実装しているハッシュでは，占有率が80％になると，SmallTreeGCを用いて，全ハッシュの30％を捨て，80％未満ならSmallTreeReplacementで置き換えている．この2つのパラメータの根拠となるデータを提示する．

A Method to use Hash Table Efficiently When Solving Tsume-Shogi

Ayumu Nagai　　Hiroshi Imai

Department of Information Science, Faculty of Science, University of Tokyo

7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan

Abstract

The algorithms to search adversary-agent games like Shogi or Chess have been widely discussed. Although hash tables (transposition tables) are always implemented with depth-first algorithms, they are rarely discussed. This situation is also the same with Tsume-Shogi. We have already studied an algorithm to solve Tsume-Shogi problems. In this paper we would like to take up our implementation of hash table. Our hash table removes 30 % of all entries by SmallTreeGC when load factor exceeds 80 %. While load factor is under 80 %, the hash table replaces an entry with a new one by SmallTreeReplacement. We would like to show some figures that are grounding these numerical values.

1　はじめに

一般的に，最良優先探索法では，探索した節点に関するすべての情報をリストなどの形で保持するのに対し，深さ優先探索法では，ハッシュの中に登録するという形で保持する．最良優先探索法ではリストが不可欠であるのに対し，深さ優先探索法においては，ハッシュは必ずしも必要ではないが，あった方が以前の探索結果を利用でき，効率的な探索が可能となる．このように，深さ優先探索法の方が柔軟性に富むのである．したがって，思考ゲームの探索においては，深さ優先探索法とハッシュという組合わせを用いることが多い．

深さ優先探索法とハッシュという2つのテーマのうち，深さ優先探索法については，古くから論じら

れてきた [9]. しかし，もともと探索の改良法 (enhancements) として登場した [10] こともあり，ハッシュについてはあまり論じられることはなかった．

現在チェスや将棋で実際に実装されることの多いハッシュは，closed hash の一番単純なモデルである．すなわち，各局面に対し Zobrist hashing [11][5][15] と呼ばれる方法によって乱数の値 (64bit 程度) を割り当て，この値を hash key として用いる．また，ハッシュへの登録・参照には高速性が要求されるので，衝突が起こっても rehash はしないことが多い．

これまでにハッシュについて論じた主要な論文としては，ハッシュを使うことの有用性について論じたもの [7][8]，それから，ハッシュ関数によって得られた hash value の衝突した場合のために，hash value の同じところに 2 つのエントリを登録できるようにした，two-level transposition table [3] がある．しかし，two-level transposition table は，1 回だけ rehash を行う closed hash の一種であり，rehash がチェスでも有用だったというにすぎない．結局，チェスや将棋においては，ハッシュは単純な実装にして，高速性に期待するというのが実際のところである (図 1 参照)．

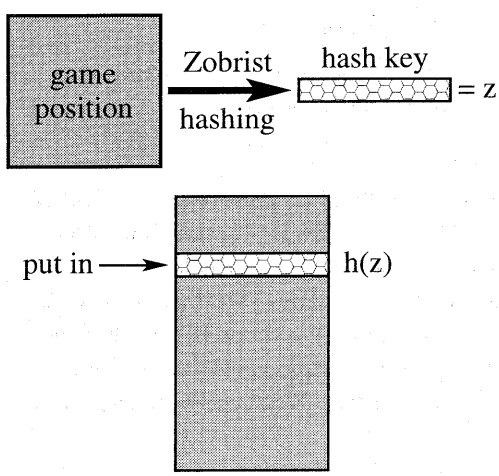

図 1: 最も一般的なハッシュの実装

詰将棋においては，状況は若干複雑になる．詰将棋では合駒が可能なので，過去に出現したのと同じ盤面で持駒だけが違うという状況が頻繁に発生する．この時，持駒だけが異なる 2 局面間に優越関係 [13] が発生して，探索せずにいきなり「詰む」ということが分かる場合がある．そこで，現在出現している局面に対してハッシュの参照を行う際に，同時に現局面と優越関係にある局面をも参照できるようにすることが多い．具体的には，盤面と持駒で別々に hash key を計算するようにし，たとえ持駒が異なっていても，盤面の同じ局面には同じ hash value が割り当てられるようにする．このようにすると，ハッシュへの登録・参照を行う際に，複数の index を参照するようになる．すると必然的に，hash value が衝突した場合に rehash を行うような実装になる．しかし rehash の方法は一意ではない．確実に言えることは，hash hit するまで rehash し続けるというのは現実的でないことである．ハッシュにはとにかく高速性が要求されるので占有率が高い状態で rehash し続けると，ハッシュの登録・参照のオーバーヘッドが大きくなってしまう．rehash の回数には上限を設けるのは当然として，rehash するごとに別のハッシュ関数を用いると，衝突の可能性が減るというメリットはあるが，ハッシュ関数を引く手間がかかる上に，メモリ上をあちこち飛び飛びに参照することになるので，キャッシュミスが起こりやすくなり，ハッシュに要求される高速性が損なわれるデメリットがある．

この辺の実際の詳しい実装については，脊尾の実装しか知られていない．脊尾の実装では，ハッシュ関数によって得た hash value が衝突を起こした場合，すぐ隣りのエントリを最大 100 個ほど参照する，linear probing を採用している [13]. (つまり rehash の回数は 100 回前後; 図 2 参照)

また，ハッシュが十分にあれば助かるのであるが，現実には解くのに非常に時間のかかる問題もある．そのような問題を解く際には，ハッシュを使い尽くしてしまうことがある．この時何らかの方法で古いエントリを捨てて新しいデータを書き込めるようにしないと，探索を続行できない．そのためのアイデアが，いくつか知られている．野下は，

- 前回の GC から今までに参照したエントリを残す
- 詰手数の長いエントリを残す

のような基準を用いている [15]. また脊尾は，

- 不詰のエントリや，詰・不詰のまだ不明なエントリよりは，詰みのエントリを残す

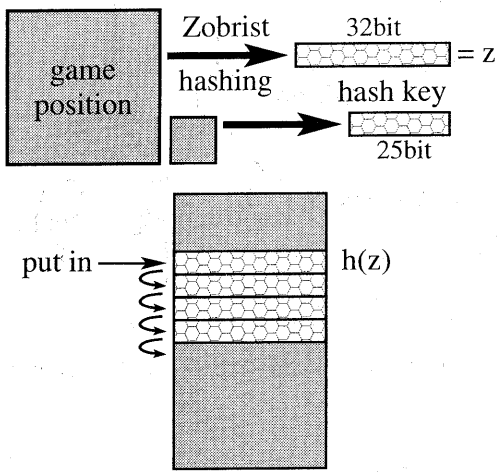

図 2: 脊尾の実装

- 詰みのエントリ同士では，詰手数の長い方を残す．
- 不詰のエントリ同士では，より証明数の大きい方を残す

という基準を用いている [13].

2 我々のハッシュの実装

我々の実装では，持駒を除いた各盤面に，Zobrist hashing により 64bit の乱数値を割り当て，また持駒については脊尾の方法 [14] により 25bit 割り当てている．hash key としては，両者を合せた 89bit を用いているが，ハッシュ関数には 64bit の乱数値のみを与えて hash value を得ている．したがって，持駒が異なっていても，盤面の同じ局面に対しては同じ hash value が割り当てられる．我々のプログラムでは，hash value の衝突が起こっているときには，rehash を最大 9 回している．つまり，各局面に関するデータの登録場所としては 10 個の候補がある．ハッシュ関数については，毎回別のハッシュ関数を呼ぶのでは前章に述べたように，オーバヘッドが大きくなって好ましくないので，double hashing [4] の一種を用いている．具体的には，Zobrist の 64bit 乱数値を z，普通の意味でのハッシュ関数を $h(z)$ とすると，1 ～ 8 までの数値を返す第 2 のハッシュ関数 $h_d(z)$ を用いて，

$$h_i(z) = h(z) + i \cdot h_d(z) \qquad (i = 0, \ldots, 9)$$

のようにして計算している (図 3 参照)．第 2 のハッシュ関数 $h_d(z)$ の返り値は，キャッシュミスが発生するのをなるべく抑えるために小さめの値を取るようにし，また常に 1 では団子状にエントリが固まってしまいやすくなるので，それを避けるために，このように小さめの値でばらけるようにしている．

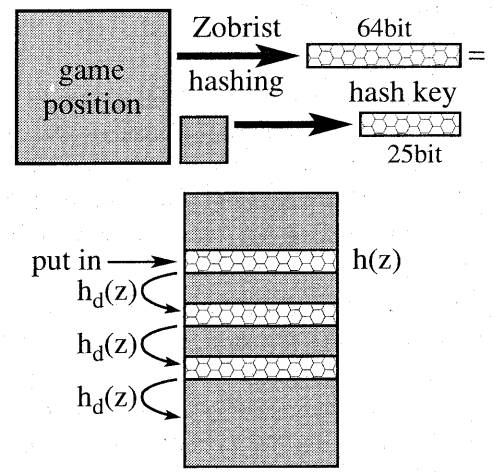

図 3: 我々の実装

ここで，登録場所の候補 10 個 (rehash の回数が 9 回) という数字の根拠としては，一般に，候補の個数が多くなればハッシュ参照のオーバーヘッドが増して，高速性という観点から良くない．しかし，候補の個数が少ないと，書き込めなくなることがある．候補 5 個だとそういう事態が発生したが，10 個ならそういう事態は現実的には発生していないので，10 個にしている．この辺りは，実装の仕方などにも多いに依存する．

将来的には，この登録場所の候補 (あるいは最大 rehash 回数) は動的に単調増加させるようにしたい．

ハッシュを使い尽くした後，古いエントリを捨てるためのアルゴリズムとして，長井はいくつかの手法を提案している [6]．その中で我々の実装に用いているのは，SmallTreeReplacement と SmallTreeGC である．最適なエントリの捨て方のアルゴリズムは，ハッシュの実装に依存する．文献 [6] で

はハッシュには open hash (chain hash) を用いていた．今回は closed hash を用いているので，エントリの捨て方のアルゴリズムは次のようにしている．

1. ハッシュの占有率が 80 % に達したら，全ハッシュの少なくとも 30 % を SmallTreeGC で捨てる．SmallTreeGC というのは，全エントリを見て，ノードの部分木の小さい順にエントリを消していくというものである．ノードの部分木とは，そのノードの下に存在するノードの個数のことである．

2. 占有率が 80 % に，SmallTreeReplacement で置き換える．SmallTreeReplacement とは，置き換え候補となっているいくつかのエントリを見て，部分木の最も小さいエントリを消して新しいエントリに置き換えるというものである．

さて，次章では SmallTreeGC を呼ぶタイミングである，ハッシュの占有率 80 %というパラメータ (以後，パラメータＡと呼ぶ) と，その時 30 % のエントリを捨てているが，この 30 % というパラメータ (以後，パラメータＢと呼ぶ) の根拠を示し，妥当性を検証する．

一般に，SmallTreeGC を呼ぶタイミングを遅く (パラメータＡを大きく) したり，SmallTreeGC によって捨てるエントリの割合を少なく (パラメータＢを小さく) したりすると，ハッシュの平均的な占有率が上がって，hash value の衝突が起こりやすくなるので，ハッシュの登録・参照のオーバーヘッドが少し大きくなる．しかし，重要なエントリが GC されずにハッシュ中に残り，必要なときに参照できる可能性が高まるので，著しく探索量が減ることもある．更にいうと，実際の探索木は木ではなくグラフなので，逆に著しく探索量が増えることもある．例えば，図 4*B-C* のように探索して，*C* が詰まなかったとする．その後，*B* の探索を終えたあと，*A-C* のように探索して，*C* が詰んだとする．すると，*B* に対応するハッシュデータが残っていると，*B-C* の詰手順に気づきにくくなる．運良く *B* のハッシュデータが捨てられていると，*B* を探索したときに即座に *B-C* の詰手順に気づくことが出来る．このように，実際には複雑な要因が絡み合っている．

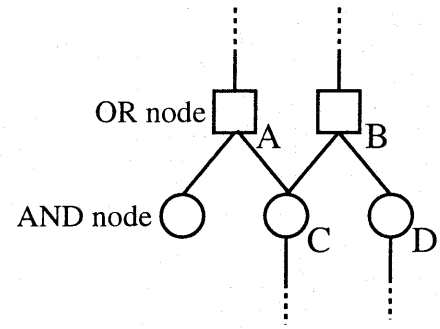

図 4: *B* のデータは捨てられていた方が良い場合もある

3 実験結果

SmallTreeGC や SmallTreeReplacement のようなアルゴリズムの導入による最大の成果は，非常に少ないメモリしか用いなくても，かなり難易度の高い問題が解けることである．例えば，「ミクロコスモス」という，現在までの史上最長の詰将棋がある．脊尾の開発したソフト「脊尾詰」が史上初めて「ミクロコスモス」を解いたとき Pentium 166MHz，RAM 256MB というマシン環境で，ハッシュに 224MB 使用して 20 時間 52 分 2 秒かかっている [1][2]．ここで注目したいのはハッシュに使用したメモリ 224MB という事実である．その後もっと優れたマシン環境で解けた例が報告されている [12] が，いずれも 224MB 以上のメモリを用いている．しかし我々のプログラムは，UltraSparcII 400MHz，RAM 4GB というマシン環境で，ハッシュに 28MB 使用して 3593.47 秒 で解いている．このように，圧倒的に少ないメモリ環境でも難問詰将棋を解くことができる．これから掲載する実験結果は，「将棋図巧」「将棋無双」「続詰むや詰まざるや」の結果であるが，これらの作品集中の問題をすべて，ハッシュ 2.8MB で解いていること自体，注目に値する．

「将棋図巧」「将棋無双」は難解作の多いことで有名な，江戸時代の詰将棋作品集である．これらの作品集の問題のうち，不詰の問題を除いた「将棋図巧」99 題，「将棋無双」94 題を実験対象とする．また，江戸時代から昭和までの名作詰将棋を集めた「続詰むや詰まざるや」についても，不詰の問題を除いた 195 題を対象とする．

さて，表1は，パラメータAを60 %，70 %，80 %，90 %にした場合(80 %のときを基準とする)の実行時間の変化の割合の平均を表す．表2 は，その時の実行時間の合計を表す．ただし，問題の中にはすぐに解けるものも多い．そのような問題は評価から外した．その理由は，ここでは実行時間で比較するので，そのような問題はばらつきが大きくなりやすい上に，SmallTreeGC，SmallTreeReplacement の効果をあまり確かめることができないからである．問題の絞り方としては，パラメータAを80 %，パラメータBを30 %に設定した場合に，SmallTreeGC を行うほど十分に難易度の高い問題に絞った．そのような問題の問題数は，表1と表2に記入してある．尚，「将棋図巧」8番は他の問題と比べて問題サイズが大きく，この1題の持つ影響力が強すぎるので，今回の実験からは除外した．

hash size\占有率	60 %	70 %	80 %	90 %
2.8MB (109 題)	0.1090	0.1267	0	−0.0733
5.6MB (67 題)	0.3515	0.1949	0	−0.0288
8.4MB (48 題)	0.0513	0.1000	0	−0.0318
11.2MB (42 題)	0.1582	0.0039	0	0.0716

表 1: SmallTreeGC を呼び出す時の占有率(パラメータA)を変えたときの実行時間の変化の割合 (80 %のときを基準とする; パラメータBは30 %に固定)

hash size\占有率	60 %	70 %	80 %	90 %
2.8MB (109 題)	10566.2	7038.3	6487.1	4712.1
5.6MB (67 題)	7126.6	6776.3	6302.1	4723.3
8.4MB (48 題)	4388.0	4721.3	4494.7	6200.4
11.2MB (42 題)	4999.1	3862.1	4635.2	4476.5

表 2: SmallTreeGC を呼び出す時の占有率(パラメータA)を変えたときの実行時間の合計 [秒]

表1における「変化の割合」とは，単に比を取っているのではない．例えば，問題数を n として，問題 $i(0 \leq i < n)$ を解くのに，a_i の時間がかかったとする．基準となるパラメータの場合には b_i の時間がかかったとする．単純に比で計算するなら，「比の平均」は次のようになる．

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{a_i}{b_i} \tag{1}$$

この場合，平均が1より小さければ，基準のパラメータより良いことが期待できる．しかし，これでは，$a = \{100, 200\}$, $b = \{200, 100\}$ のとき，すなわち問題が全2問からなり，ある問題は，条件Xで100 秒，条件Yで 200 秒で解け，またもう一方の問題は，条件Xで 200 秒，条件Yで 100 秒で解けた場合，条件Yを基準とすると，平均は $\frac{1}{2}(\frac{100}{200} + \frac{200}{100}) = 1.25$ となる．しかしこの場合，明らかに2つの条件の間に優劣をつけられない．それよりは，次のようにして「変化の割合の平均」を定義する．

$$\frac{1}{n}\sum_{i=0}^{n-1} x_i \tag{2}$$

$$\text{ただし，} x_i = \begin{cases} a_i/b_i - 1 & (a_i \geq b_i) \\ 1 - b_i/a_i & (a_i < b_i) \end{cases}$$

この場合，平均が0未満なら，基準のパラメータより良いことが期待できる．すると今の例では，$\frac{1}{2}\{(1 - \frac{200}{100}) + (\frac{200}{100} - 1)\} = 0$ となり，確かに直感に沿っている．表1における「変化の割合の平均」とは，式(1)ではなく，式(2)で計算している．表1は，ハッシュサイズが 8.4MB 以下の場合には，パラメータAを90 %にした方がわずかに良いことを示している．表2も，ハッシュサイズが 5.6MB 以下の場合には，パラメータAを90 %にした方が良いことを示している．表2は数少ない超難問の実行時間の影響を強く反映しているので，表1の補助的資料と位置づけた方が良い．

表3は，パラメータBを10 %，20 %，30 %，40 %にした場合(30 %の時を基準とする)の実行時間の変化の割合の平均を表す．表4は，その時の実行時間の合計を表す．ここでも簡単な問題は除外している．

hash size\回収率	10 %	20 %	30 %	40 %
2.8MB (109 題)	−0.0224	0.0021	0	−0.0013
5.6MB (67 題)	0.4396	0.1086	0	0.0306
8.4MB (48 題)	0.4287	0.0820	0	0.1852
11.2MB (42 題)	0.5035	0.0764	0	0.0830

表 3: SmallTreeGC で捨て去るエントリの割合(パラメータB)を変えたときの実行時間の変化の割合 (30 %の時を基準とする; パラメータAは80 %に固定)

表3は，パラメータBは30 %の時が一番良いこと

hash size\回収率	10 %	20 %	30 %	40 %
2.8MB (109 題)	7768.3	6495.9	6487.1	5364.3
5.6MB (67 題)	9360.6	5173.4	6302.1	5600.6
8.4MB (48 題)	13713.4	4541.5	4494.7	5452.5
11.2MB (42 題)	6968.2	4198.8	4635.2	5015.1

表 4: SmallTreeGC で捨て去るエントリの割合 (パラメータB) を変えたときの実行時間の合計 [秒]

を示している．表4は，パラメータBが30 %で必ずしもいつも最善になるとは限らないことを示している．しかしこの辺は微妙な問題であり，表4は数少ない超難問の実行時間の影響を強く反映しているので，表3の補助的資料と位置づけた方が良い．

4 まとめ

これまであまり論じられてこなかった，思考ゲーム探索におけるハッシュの実装法について論じた．我々の詰将棋を解くプログラムでもハッシュが実装されており，これについて詳しく解説した．我々のハッシュでは SmallTreeGC と SmallTreeReplacement を実装しており，ここに2つのパラメータが必要になる．

パラメータA，つまり SmallTreeGC を呼ぶタイミングは，現在の我々の実装ではハッシュの占有率が 80 % を越えた時としているが，実験の結果，90 % にしても良さそうである．

パラメータB，つまり SmallTreeGC で全ハッシュエントリのどのくらいを捨て去るか，については，現在の我々の実装では 30 % を捨てているが，実験の結果，これは 30 %程度で良いことが分かった．

これで，2つのパラメータの値の根拠を示すことができた．

謝辞

詰将棋の問題や実験データをはじめ，様々な情報を提供して頂いた岡崎正博氏，門脇芳雄氏，山田剛氏，加藤徹氏，更にプログラムの実装にあたって，ご自分の実装について色々ご説明下さった脊尾昌宏氏に深く感謝致します．

参考文献

[1] コンピュータ将棋協会メーリングリスト, 1997.

[2] コンピュータ将棋協会誌, 8 1997.

[3] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement Schemes for Transposition Tables. *ICCA Journal*, 17(4):183–193, 1994.

[4] D.E. Knuth. *The Art of Computer Programming Sorting and Searching Second Edition*, volume 3. Addison-Wesley, 1998.

[5] D. Levy and M. Newborn (小谷善行監訳). コンピュータチェス, pages 214–219. サイエンス社, 1994 (原書 1990).

[6] A. Nagai. A new Depth-First-Search Algorithm for AND/OR Trees. Master's thesis, Department of Information Science, University of Tokyo, Japan, 1999.

[7] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.

[8] J. Schaeffer and A. Plaat. New Advances in Alpha-Beta Searching. In *Proceedings of the 24th ACM Computer Science Conference*, pages 124–130, 1996.

[9] C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 7(41):256–275, 1950. See also: Levy,D.N.L.(ed.) Computer Games I, Springer-Verlag, pp.81–88,1988.

[10] D. Slate and L. Atkin. *Chess Skill in Man and Machine*, chapter 4 Chess 4.5 – the Northwestern University chess program, pages 82–118. Springer-Verlag, 1977.

[11] A.L. Zobrist. A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison WI, USA, 1970. Reprinted in *ICCA Journal*, Vol. 13, No. 2, pp. 69–73, 1990.

[12] 加藤徹. 超長編作品のプログラムによる解図と検討, 2 1998. コンピュータ将棋協会の岡崎正博氏より頂く.

[13] 脊尾昌宏. 共謀数を用いた詰将棋の解法, volume 2, chapter 1 コンピューター将棋の進歩, pages 1–21. 共立出版, 1998.

[14] 脊尾昌宏. 詰将棋を解くアルゴリズムにおける優越関係の効率的な利用について. In *Game Programming Workshop in Japan '99*, pages 129–136, 1999.

[15] 野下浩平. 詰将棋を解くプログラムＴ２, volume 1, chapter 3 コンピューター将棋の進歩, pages 50–70. 共立出版, 1996.