ハッシュライフ拡張ライブラリの実装

岸本 誠[1,a)]

概要：ライフゲームの盤面を四分木で表現し，またハッシュ値にもとづいて部分木を共有することで効率的にライフゲームを扱う手法であるハッシュライフを，ruby の拡張ライブラリとして実装している．ハッシュライフの大きな特徴である高速化などは未実装であり，発表および本稿執筆時点での現状にもとづき報告する．また特に，筆者による独自の工夫として，8×8 の矩形の段階でビット演算処理による手法に切り替える，という実装を行っているためその部分については詳しく説明する．加速手法については未実装の段階で，手元の PC（CPU は Phenom II X4 905e, 2.5GHz）では，1024×1024 の約 2 万世代を約 6 分で計算できている．

キーワード：ライフゲーム，ハッシュライフ，ruby 拡張ライブラリ

1. はじめに

ハッシュライフは，ライフゲームの盤面中の $2^n \times 2^n$ の矩形の領域を，四分木構造により表現するライフゲーム処理系の実装手法である．ハッシュ値と連想配列（ハッシュテーブル）を利用して，内容が同一の部分木を共有する．すなわち，実際の内部構造としては木ではなく有向非巡回グラフ (DAG) になっていて，(1) 必要な記憶容量を，同じパターンが多数現れる典型的な場合には著しく削減する　(2) 同じパターンの将来世代のパターンもやはりいずれも同じであるので，それについても共有し，さらにはキャッシュする．以上によって，ライフゲームの盤面を効率的に扱うことができる手法である．

以上のような利点の一方で，ノードを表現するメモリオブジェクトの管理などが必要であり実装には手間が多い．より単純な点では，盤面を一世代先に進めるだけでも，一般的な手法と比べて手数が掛かるといった点も欠点と言える．

筆者はこれを実装するにあたり，既存のスクリプティング言語処理系の拡張ライブラリ，具体的には ruby の拡張ライブラリとすることで，そのオブジェクト管理を利用し実装の手間を省くことにした．また，盤面を一世代先に進める処理について，ハッシュライフとしての実装ではなく，各行の内容が入ったワードのビット演算により次の世代を得る手法とのハイブリッドで実装した．後者は，既に同じことが行われている実装がどこかには存在するだろうが，筆者自身の工夫によるものであるため，詳細に説明する．

本稿は以上の内容を説明した後，今後実装する予定の機能についても述べる．

2. ライフゲームとハッシュライフ

ライフゲームは 2 次元のセルラー・オートマトンであり，各セルは「生命」が「いる」「いない」の 2 状態のどちらかの状態をとる．通常，有限個

[1] (所属なし)
[a)] ksmakoto@dd.iij4u.or.jp

の「いる」状態のセルがあり，残りの無限個のセルは「いない」[*1] 状態である [*2]．次の世代における各セルの状態は，8 近傍（ムーア近傍）と，そのセル自身の状態から決まる [*3]．ライフゲームの詳細については文献なども豊富であるのでそちらに譲る [*4]．

続いてハッシュライフについて説明する．ハッシュライフはライフゲームのコンピュータ・プログラムを実装する際の手法で，$2^n \times 2^n$ の矩形の領域を，高さ n の再帰的な四分木で表現する．さらにその四分木の実装において，それが表現する矩形の内容にもとづいたハッシュ値を利用して，同じ内容であればノードを共有し，具体的なデータ構造としては有向非巡回グラフ (DAG) とする．またノードの再利用のためにハッシュ値をキーとする連想配列（ハッシュテーブル）を使う．このためハッシュライフと呼ばれている．

発表ではハッシュライフ自体についての説明も行ったが，こちらでは筆者が実装にあたって参照した文献を挙げるのみとする．ライフゲームほどには豊富ではないが，現状で実装を自作し始めることが容易な程度の情報は入手可能であろう．

ハッシュライフの提案者は，最初に発見された無限に成長するパターンであるグライダーガンで知られる Ralph William Gosper Jr. (Bill Gosper) である [1]．提案中ではこの手法が必要になるようなパターンの例として，もう一つの成長型の類型である puffer train で，やはり彼の発見によるパターン [*5] が，5533 世代まで成長させて初期部分の「煙」が安定するのを待つ必要がある，というものを挙げている．2012 年の夏のプログラミング・シンポジウムでの発表「セルオートマトンのプログラムハック」[6] では他のいくつかのハックとともにエッセンスが説明されている．具体的な（Java による）コード片を伴った解説である Dr Dobb's Journal の記事 [3] を参考にすれば，各自の実装のコードは書き始められるであろう．プログラミング言語に依存していない解説としては，はてな id:Nos による記事 [4] と，同人誌「チーズケーキ研究 17.08」収録の記事 [5] がある．

[*1] 「生」と「死」とすることも多いが，死体が無限に広がっている，というのでは印象がよくないと筆者は感じるので，単に「いない」とする．

[*2] 無限個の「いる」セルがあるようなことを許すと，一直線に無限に「いる」が並んでいるという，無限に増えるパターンが簡単に作れてしまう．

[*3] このことを指して，ハッシュライフの原文献 [1] では 9 近傍の規則（nine-neighborhood rule）と言っている．

[*4] 『ライフゲイムの宇宙』[2] など

[*5] Puffer 2 などと呼ばれている `http://www.conwaylife.com/wiki/Puffer_2` (2018 年 11 月 27 日閲覧)

3. データ構造

Ruby や Python などのスクリプティング言語処理系では，C 言語ないし，それらの言語処理系のインタプリタを実装している言語で，ライブラリを実装できるようなものが多い．用途としては，しばしばそういった言語に対して言われる「遅い」という不満に対する回答であるとか，既存のライブラリとのインタフェース層を実装し，それらの言語から既存のライブラリを利用する，といったものがある．これを利用すると，言語処理系が持っているオブジェクトシステムやガベージコレクタなどを利用できるため，特に C 言語ではそれらを自分で実装する手間が大幅に省ける．

筆者は，スクリプティング言語処理系 `ruby` の拡張ライブラリとして実装することを選択した．さらに将来的には，Ruby による GUI フロントエンドや，ブラウザアプリ用のサーバと組み合わせるバックエンドとして使うといった場合にオーバヘッドが少なく使える利点がある．

`ruby` 拡張ライブラリでは，オブジェクトのデータ構造として任意の構造体を使うこともできる．次のような構造体を定義する．

```
typedef struct hl_node_t {
    VALUE nw, ne;
    VALUE sw, se;
    VALUE hash_key;
    VALUE g_cache;
    int lv;
} hl_node;
```

`VALUE` は任意の Ruby オブジェクトを値にすることができる型である．`nw`, `ne`, `sw`, `se` は四分木の

各枝，hash_key は 4 個の子盤面から計算したハッシュ値を保存しておくためで，g_cache は次の世代を求めたものをキャッシュしておくためのメンバである．lv（レベル）はそのノードの，葉からの高さである．

nw, ne, sw, se には，もっとも下のノードではセルの「いる」「いない」の状態を表現する情報が入り [*6]，それ以外の上のノードではそれぞれの四分の一の盤面を表現している子ノードが入る．

また，次のような関数も用意する．

```
static void hl_node_mark(void *ptr)
{
    hl_node *nod = ptr;
    rb_gc_mark(nod->nw); rb_gc_mark(nod->ne);
    rb_gc_mark(nod->sw); rb_gc_mark(nod->se);
    rb_gc_mark(nod->g_cache);
}
```

ruby ではマーク＆スイープ GC を行うので，拡張ライブラリのオブジェクト中で VALUE 型を直接扱っている場合は，マーク操作でそれを追跡しなければならない．hash_key は fixint の値であるので省略している（Ruby レベルからは変更できないので問題は起きない）．

その他，オブジェクトのサイズを得る関数を含め，以下のような構造体により，ruby ランタイムにこのデータ構造についての情報を渡す．

```
struct rb_data_type_struct hl_node_type = {
    .wrap_struct_name = "HL_Node",
    .function = {
        .dmark = &hl_node_mark,
        .dfree = 0,
        .dsize = &hl_node_dsize,
        .reserved = {(void *)0, (void *)0}
    },
    .parent = (rb_data_type_t *)0,
    .data = (void *)0,
    .flags = (VALUE)0
};
```

[*6] ここでは原理そのままの実装について述べている．後述する実際の，筆者による現在の実装では，より上の段階で盤面を表現するビットマップ的なデータになっている．

4. アルゴリズム

ハッシュ値の求め方と，そのハッシュ値にもとづいたノードの比較は次のようになる．

(疑似コード)

```
ハッシュ値 = h(nw.obj_id, ne.obj_id,
              sw.obj_id, se.obj_id)
```

nw, ne, sw, se は各ノード，obj_id は，メモリアドレス等を利用した，重複しないことが保証されているオブジェクト ID などと呼ばれる値，h はハッシュ関数とする．

ここで，ハッシュ値は衝突の可能性があるから，ノード a とノード b とを比較する手続きは次のようになる．

(疑似コード)

```
IF ノード a.obj_id ＝ ノード b.obj_id THEN
    ノード a と ノード b は同一
ELSE_IF ノード a.hash ≠ ノード b.hash THEN
    ノード a と ノード b は異なる
ELSE
    それぞれの nw, ne, sw, se について
    obj_id が等しいか比較
      ==> 全て等しければノード a と
          ノード b の内容は同一
FI
```

(これについてはハッシュ値の衝突の扱いを間違えていたため，実際の実装のコードはまだ無い)

以上によって，同一の内容を表現しているオブジェクトならば，ボトムアップに唯一であるようにしながら，四分木（四分 DAG）を構成すればよい．

実装の手順としては，以上のようなデータ構造によって，空白（全て「いない」）の盤面を生成する手続きをまず実装し，続いて，盤面を視覚的に表示する手続きや指定した位置のセルをトグルする手続きを実装すれば，ハッシュライフのアルゴリズムを実装しデバッグする用意が整う．それらの詳細は省略する．

続いて，次の世代を求める最も肝要な関数について述べる．

まず先頭部分は次のようになっている．

```
static VALUE
node_gen(VALUE obj)
{
    hl_node *nod;
    TypedData_Get_Struct(obj,
            ...(略)..., nod);
    {
        VALUE const cache = nod->g_cache;
        if ( ! NIL_P(cache)) {
            return cache;
        }
    }
    if (nod->lv == 2) {
        return nod->g_cache =
                node_gen_8x8(nod);
    }
```

Ruby オブジェクトから中身の構造体を取り出し，既にキャッシュに次の世代を求めた結果があればそれを使う．続いて，8×8 まで木が小さくなっていたら，後述するビット演算処理による基底処理のほうに切り替える．

次がハッシュライフの中心的な処理である．

```
    VALUE nw, ne, sw, se;
    nw = nod->nw; ne = nod->ne;
    sw = nod->sw; se = nod->se;
    VALUE n00, n01, n02;
    VALUE n10, n11, n12;
    VALUE n20, n21, n22;
    n00 = center(nw);
    n01 = centered_h(nw, ne);
    n02 = center(ne);
    n10 = centered_v(nw, sw);
    n11 = center(center(obj));
    n12 = centered_v(ne, se);
    n20 = center(sw);
    n21 = centered_h(sw, se);
    n22 = center(se);
    int const lv = nod->lv;
    return nod->g_cache = get_node(lv-1,
        node_gen(get_node(
          lv-1, n00, n01, n10, n11)),
        node_gen(get_node(
          lv-1, n01, n02, n11, n12)),
        node_gen(get_node(
          lv-1, n10, n11, n20, n21)),
        node_gen(get_node(
          lv-1, n11, n12, n21, n22)));
```

- 2 段階小さい矩形 9 個を作る
- その矩形 4 個から作った中間盤面について再帰する，を 4 回おこなう
- その 4 個をまとめることで，自分自身の中央部分についての次の世代が得られる

という処理である．参考文献にある，ハッシュライフ自体の解説と併読すれば，straightforward に実装しているだけなので理解は容易であろう．

続いて，筆者独自の工夫である，ビット演算処理とのハイブリッド化について述べる．

現在の実装では，前述のように 8×8 を表現している所から先は，四分木では表現していない．具体的には，そのノードが持つ 4 個の 4×4 の矩形はいずれも葉であるとして，ビットマップ化してその整数値として表現している．なお，この選択は ruby 拡張ライブラリとして実装していることによる制約と関連がある．ruby の実装では，VALUE 型のワードの LSB をタグに使い，fixint に収まる整数値を表現するオブジェクトとそれ以外のオブジェクトの識別に使っている．そのため，ワードが 64 ビットの場合，64 ビットの全てが必要な整数値は fixint に収まらない可能性があるから，以上のような葉の扱いを，もう一段大きい枝にはできない．

このようなハイブリッド化による影響は，center という名前の関数にも及んでいるが，ここではそれは省略し，node_gen_8x8 についてのみ述べる．

ライフゲームの次の世代を求める処理はビット演算でもできる，ということは既に知られている [7]．抽象的にはビット長が無制限のビットストリング演算が必要である．しかしハッシュライフ

では，ノードが表現している範囲の中央部分のみが求まれば良いので，このビット演算による手法と相性が良いと筆者は考えている．以下，実装を見てゆく．

```
typedef uint64_t U64;
int const nw = FIX2INT(nod->nw);
int const ne = FIX2INT(nod->ne);
int const sw = FIX2INT(nod->sw);
int const se = FIX2INT(nod->se);
U64 x =
  ((U64)((unsigned)nw & 0xffffU) << 48) |
  ((U64)((unsigned)ne & 0xffffU) << 32) |
  ((U64)((unsigned)sw & 0xffffU) << 16) |
  (U64)((unsigned)se & 0xffffU) ;

// Hacker's Delight 7-2
x = ((x & 0x0000ff000000ff00L) << 8) |
    ((x >> 8) & 0x0000ff000000ff00L) |
    (x & 0xff0000ffff0000ffL) ;

x = ((x & 0x00f000f000f000f0L) << 4) |
    ((x >> 4) & 0x00f000f000f000f0L) |
    (x & 0xf00ff00ff00ff00fL) ;
```

前半はそれぞれの四分の一の矩形を取り出して順番に詰めているだけである．後半のコメントで示しているのはシャッフルのコードであり[*7]，視覚的に説明すると，次のようにワード内のビットを入れ替えている．

処理前のビットに，MSB から順に
次のように名前を付けたとする

```
00 01 02 03 04 05 06 07
08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17
18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27
28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37
38 39 3a 3b 3c 3d 3e 3f
```

[*7] 詳細には，全ビットをシャッフルする操作の一部をおこなっている．

処理後は次のように並んでいる

```
00 01 02 03 10 11 12 13
04 05 06 07 14 15 16 17
08 09 0a 0b 18 19 1a 1b
0c 0d 0e 0f 1c 1d 1e 1f
20 21 22 23 30 31 32 33
24 25 26 27 34 35 36 37
28 29 2a 2b 38 39 3a 3b
2c 2d 2e 2f 3c 3d 3e 3f
```

残りは次のようになっている．

```
uint64_t const w = x >> 8;  // x-
uint64_t const y = x << 8;  // x+
/* http://parametron.blogspot.com/2010/05/life-
game_10.html */
uint64_t const a0 = w & y;
uint64_t const b0 = w ^ y;
uint64_t const c0 = x ^ b0;
uint64_t const d0 = c0 >> 1;
uint64_t const c1 = c0 << 1;
uint64_t const e0 = c1 ^ d0;
uint64_t const f1 = (b0 & e0) | (c1 & d0);
uint64_t const c4 = (x & b0) | a0;
uint64_t const b1 = c4 << 1;
uint64_t const c5 = c4 >> 1;
x = (((b1 & c5) | (a0 | f1)) ^
     ((a0 & f1) | (b1 | c5))) &
    ((b0 ^ e0) | x);
return INT2FIX(
  ((x >> 30) & 0xf000)|
  ((x >> 26) & 0x0f00)|
  ((x >> 22) & 0x00f0)|
  ((x >> 18) & 0x000f) ) ;
```

ワードの処理であるので，64 ビットマシンで効率よく処理できることを意図している．実装前との比較で，1 割程度の速度向上があった．

5. テストと計測

数百〜数千世代にわたって不規則な成長を続ける「長寿型」を使い，最終状態が他の手法による実装によって得られた「ゴールデン」と一致する

か，を比較し，テストとしている．また，所要時間をベンチマークにも使っている．

またキャッシュ率のベンチマークも実装中である．ノードへの新しい参照について，既にあるノードを利用できた場合をヒット，そうでなかった場合をキャッシュミスとして調べたところ，だいたいどの長寿型でもヒット率は 8 割前後であった．

ノードが表現するサイズ別に計測すべきであり，それは今後の課題である．

6. 今後の課題

ハッシュライフの大きな利点のひとつである加速が未実装であり，それを実装することで巨大で非常な世代数を要するパターンを実行する，という最終目標に近付くことが課題である．

また，ノードが持っているキャッシュについては弱参照（GC によって対象が回収されることがあり，その場合には無効になる参照）であるべきであり，その実装が必要である．

ビット演算処理の 256 ビット化も，256 ビット演算が可能な環境を入手し次第，実装したい．

7. 質疑応答

シンポジウムでの発表では，以下のような質疑応答があった．

- Q. ビットの詰め合わせはシフトではなく共用体を使うと効率的になるのでは？ （丹治昭夫 (デンパン)）
- A. (近年の) コンパイラはそれぐらいは賢い，という意識で書いているが，やってみたら良いコードが生成された，ということは十分あると思う．
- (コメント) セルの情報としてのビットを並べる順番に，Morton order とかヒルベルト曲線とかの，4 分木とうまく合う順序を使うことで，詰替えなどをうまいことできないか．（松崎公紀 (高知工科大)）
- (A.) (できるかなぁ……)

8. まとめ

筆者が実装中のハッシュライフについて紹介した．ruby 拡張ライブラリとして実装することで省力化を図っている．ただし，GC などへの配慮がその代わりに必要なので，そういった点について主に紹介した．

さらに筆者独自の工夫として，ビット演算処理によるライフゲームの手法とのハイブリッド実装としているので，それの詳細を紹介した．64 ビットワードの計算機が一般的になり，SIMD 拡張によって 256 ビットの処理も効率よくできるようになってきている近年では，有望な手法であると考えている．

謝辞　参考文献の著者の皆様，シンポジウム会場にて議論・質疑・コメントをいただいた皆様に感謝いたします．

参考文献

[1] Bill Gosper: **Exploiting Regularities in Large Cellular Spaces**. Volume 10 of Physica D. Nonlinear Phenomena, 1984.

[2] 有澤誠 (訳): ライフゲイムの宇宙 新装版. 日本評論社, 2003.

[3] Tomas G. Rokicki: **An Algorithm for Compressing Space and Time**. Dr Dobb's Journal, 2006. http://www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478 (2018 年 11 月 27 日閲覧).

[4] はてな id:Nos: **hashlife その 2 - Nos の日記** http://d.hatena.ne.jp/Nos/20140928/1411884782 (2018 年 11 月 27 日閲覧), 2014.

[5] kriw: ハッシュライフの話. 「チーズケーキ研究 17.08」28-43, チズケ研究会, 2017.

[6] 和田英一: セルオートマトンのプログラムハック. 夏のプログラミング・シンポジウム 2012, 2012.

[7] 和田英一: **Life Game** https://parametron.blogspot.com/2010/05/life-game.html https://parametron.blogspot.com/2010/05/life-game_09.html (2018 年 11 月 28 日閲覧). ブログ「パラメトロン計算機」, 2010.