B-02

メンタルシミュレーションモデルに基づいたプログラムの読みやすさ評価

Program Comprehensibility Evaluation Based on Queue-based Mental Simulation Model

吉岡 智哉† 福田 收真† 玉田 春昭†

Tomoya Yoshioka Kazumasa Fukuda Haruaki Tamada

1. はじめに

本稿は、プログラムの読みやすさを測定する一手法であるメンタルシミュレーションモデルに着目する。この手法は、人の短期記憶を FIFO キューでシミュレートするモデルである。この手法を用いて、一般的なプログラムを対象に、プログラムの読みやすさを測定することを目的とする。

メンタルシミュレーションの簡単な例を示すため、*a = b – 3;*というプログラム断片を考える。このプログラムを人が理解するときにはまず、(1) *b* の値を思い出す、(2) *b* の値から *3* を引く、(3) (2)で得られた値を、*a* の値として記憶する、という手順で行われる。この時、(1)で変数 *b* の値を覚えているか否かでこのプログラム断片を理解するコストが大きく変わる。変数 *b* の値を覚えていなければ、変数の値を思い出すという行為が必要となり、覚えているときより多くのコストが必要となる。

我々は、仮想メンタルシミュレーションモデル（VMSM）を測定するツール Fowl を作成した。このツールはプログラムの実行系列(AR-trace)から VMSM の 6 つのメトリクス(ASSIGNMENT、 RCL、 SUM_UPD、 VAR_UPD、 BT_CONST、 BT_VAR)を算出するツールである。Fowl をバブルソート、マージソート、クイックソートの 3 つのソートプログラムに対して適用し、メトリクスを測定した。その結果、それぞれのソートプログラムから計測した 6 つのメトリクス値を分析したところ、AR-trace の長さで正規化して比較するするとよいことがわかった。また、ASSIGNMENT メトリクスと SUM_UPD メトリクスが読みやすさに影響している可能性があることがわかった。

本稿は、次のように構成されている。第 2 章で、メンタルシミュレーションモデルの定義と、このモデルで用いるメトリクスについて説明している。

プログラム	変数	数値	処理
`a = 3;`	a	3	assignment
`b = 1;`	b	1	assignment
`c = a - 1;`	a	3	reference
	-	2	Operation
	c	2	Assignment
`b = c + b;`	c	2	Reference
	b	1	Reference
	+	3	Operation
	b	3	Assignment

(α)プログラム　　(β) AddTracer の出力

図 1 ：AddTracer の出力結果

第 3 章では、評価実験について述べる。

2. メンタルシミュレーション

2.1. メンタルシミュレーションモデル

本稿では、中村らによって提案されたメンタルシミュレーションモデルを利用する[2]。メンタルシミュレーションモデルでは人の短期記憶を FIFO キューで表す。本節では、中村らによって提案されたメンタルシミュレーションモデルを要約する。

q を、キューの最大長 *L* が与えられた FIFO キューとする。*q* の要素は、変数 *a* とその値 *v(a)*のペアとして考え、*e = (a,v(a))*として表す。また、要素が空の場合は *e = ε* として表す。

q に含まれる空でない要素の数を *l(q)*で表し、L_{-1}*l(q)*の数だけ要素を *q* に保存できる。また、空の要素の数だけ *q* に保存する事が出来る。

次に、メンタルシミュレーションの手順を以下のように示す。

手順1：プログラム*p*とそれに対する入力*I*が与えられたとき、AR-trace *T*が得られるとする。AR-Traceは実行系列*I*を与えることで、AR-trace($T = \{t_0,t_1,\ldots,t_n\}$, $t_i = (a,v(a),k(a))$で表される。*k(a)*はAR-traceの処理内容を表しており、`assignment`(代入)や`reference`(参照)、`operation`(計算）のいずれかである。

手順 2：*T* の各要素について、以下のことを考える。

a) もし検査対象 t_i の中の *k(a)*が `reference`（参照）の場合、$e_i(0 \le j \le l(q))$に *a* が含まれるかどうかを調べる。

YES：この場合、*v(a)*を覚えていることを表し、*q* から e_i を削除し、*q* の末尾に再び e_i を挿入する。この一連の操作を、`recall`*(q, a, val(a))*と表す。

NO：この場合、*v(a)*を忘れていることを表し、*v(a)*を思い出すために *T* を t_{i-1} から t_0 の順に参照し、*v(a)*を得る。*v(a)*を得れば、*q* の末尾に *e* = (*a*, *v*(*a*))を挿入する。この一連の操作を、`backtrack`(*q, a, T, i*)と表す。

b) もし、検査対象 t_i 中の *k(a)*が `assignment` であれば、それは変数 a とその値 *v(a)*を新たなチャンクとして記憶することを表している。そのため、q が *l(q)<L* であれば、*q* の最後に *e=(a, v(a))*を挿入する。もし、*l(q)=L* であれば, *q* の最初の要素を削除した後に、*e* を *q* の最後に挿入する。この操作を`assignment`*(q, a, v(a))*と表す。

†京都産業大学, Faculty of Computer Science and Engineering, Kyoto Sangyo University

表１：３つのソートプログラムの VMSM メトリクスと AR-trace の平均値

	ASSIGNMENT	RCL	SUM_UPD	VAR_UPD	BT_CONST	BT_VAR	AR-trace 長
バブルソート	17953.6	25332.4	12705.6	1166949.8	12704.6	37427.4	103525
マージソート	8011	4593.7	5441	12629.1	7704	8830.6	31357
クイックソート	3746.7	1861.1	2497.3	27254.5	2019.7	4873.6	14570

2.2. メンタルシミュレーションモデルのメトリクス

メンタルシミュレーションモデルのメトリクスは、中村ら[2]や、石黒ら[1]によって提案されている。具体的な 6 つのメトリクスの定義を、以下のように示す。

メトリクスについて述べる前に、変数更新ベクトル(VUF-vector,*variable update frequency vector)*を定式化する。*p* を与えられたプログラムとし、$A(p) = \{a_1,a_2,...,a_m\}$を*p*に含まれる変数の集合とする。この時、$u(a_k)(a_k \in A, 0 \leq k \leq m)$を、*T*の中の$a_k$への代入回数とする。その時、与えられたプログラム*p*とそれに対する入力*I*を与えた*p*の*VUF-vector*は$U(p, I) = \{u(a_1), u(a_2), ..., u(a_m)\}$と表す。

ASSIGN：

`assignment` と `operation` が実行された回数である。このメトリクスは変数の値を記憶するコストを表す。

RCL：

`recall` が実行された回数である。このメトリクスは変数の値を読み返すコストを表す。

BT_CONST：

定数を`backtrack`した回数である。このメトリクスは定数の値を読み返すコスト表す。

BT_VAR：

定数を除いた変数を`backtrack`した回数である。このメトリクスは忘れた値を読み返すときにかかるコストを表す。

SUM_UPD：

$\sum_{k=0}^{m} u(a_k)$

VAR_UPD：

このメトリクスは*U(p,i)*の分散を表す。

SUM_UPDとVAR_UPDは変数の更新頻度と再計算コストを表しており、メンタルシミュレーションと密接な関係があるメトリクスである。

ここに、ある入力I_eが与えられたときに、次の式を満足させるプログラムp_1とp_2がある。

・$A(p_1) = \{m_1, n_1\}, U(p_1, I_e) = \{7, 1\}$

・$A(p_2) = \{m_2, n_2\}, U(p_2, I_e) = \{4, 4\}$

変数*m,n*はそれぞれ更新された回数を表しており、p_1から$(m_1,n_1) = \{7, 1\}$、p_2 から$(m_2, n_2) = \{4, 4\}$という結果が得られた。

P_1では$n_1 < m_1$であり、更新頻度の少ないm_1のほうが最新の値を覚えておくことが簡単であると考えられる。

p_1もp_2も、プログラム中の変数の更新回数の合計がどちらも8であるので、互いにSUM_UPDは8である。しかし、p_2はp_1よりメンタルシミュレーションする事が難しい。なぜなら、それぞれの変数の更新頻度を見ると、p_2はm_2とn_2が同じ頻度で更新されているので、m_2とn_2の両方を意識しなければならないからである。一方、p_1はm_1だけが7回更新されており、m_1のみに意識を集中できる。つまり、変数の更新頻度の分散が低い場合、メンタルシミュレーションは、多くの変数の変化を把握する必要があるため、読みにくくなっていると言える。

3. 評価実験

3.1. ツールと環境

プログラムから、AR-trace を計測するために AddTracer[3]を利用した。また、AddTracer により得られた AR-trace からメトリクスを算出するためのプログラム Fowl を作成した。

実験には、iMac(Mac OS X 10.7.4(Lion)),Intel Core i5 2.7GHz 4GB RAM を用いた。また、実験対象プログラムとして Java を採用した。Java の実行環境は Java SE 1.6.0_33 である。

3.2. 実験手順

我々は、プログラムの読みやすさを測定するため、3 つのソートプログラムを用意した。それぞれバブルソート、マージソート、クイックソートである。

それぞれのプログラムを図 2 から図 4 に示す。それぞれアルゴリズムを単純に実装したものである。ソート対象のデータは 100 個の整数値である。0 から 10^5 の範囲の乱数を用意した。その後、それぞれのアルゴリズムを用いてソートし、3 つのプログラムに対してメンタルシミュレーションモデルのメトリクスを 10 回計測した。

実験を行う際、短期記憶を表す FIFO キューの長さを決める必要がある。人の短期記憶は、Cowan らが容量の制限の無いプログラムを読む時、4 つの情報を保持できると報告している[3]。ただし、我々はメンタルシミュレーションの観点から考え、保持できる短期記憶の量を減らし、キューの長さを 3 とした。

3.3. 実験結果

表 1 に、3 つのソートプログラムから算出したメトリクスの平均値を示している。メトリクスの値が小さいほど、そのプログラムは読みやすいと言える。

表 1 をみると、バブルソートのすべてのメトリクスの平均値がほかのソートプログラムに比べ全て高くなっている。つまり、バブルソートは人にとって読みにくいプログラムという結果となった。

しかし、バブルソートは簡単なソートアルゴリズムであり、この結果は直感とは異なる。VAR_UPD 以外のメトリクスは AR-trace の行数に依存する。さらに、バブルソートの計算量は $O(n^2)$であり、マージソートとクイックソートの計算量は共に $O(nlog(n))$である。バブルソートのほうが計算量は大きく、メトリクスの値も大きい。そこで、3 つのソートプログラムのメトリクスを AR-trace の長さで正規化した。

表２：AR-trace の行数で正規化された VMSM メトリクス

	ASSIGNMENT	RCL	SUM_UPD	VAR_UPD	BT_CONST	BT_VAR
バブルソート	0.173407	0.244764	0.122684	11.271878	0.122675	0.361469
マージソート	0.255477	0.14606	0.173518	0.405205	0.247409	0.282776
クイックソート	0.257165	0.127635	0.171444	1.86791	0.138636	0.334575

表 2 に、ソートアルゴリズム毎のメトリクスの平均値を正規化した値を示す。

3.4. 考察

表 2 から、ASSIGNMENT と SUM_UPD メトリクスはバブルソートが一番低く、マージソート、クイックソートがほぼ同じ値となっている。これは、バブルソートの計算と代入回数が一番少ないことを表している。一方、RCL メトリクスはバブルソートが一番大きく、マージソート、クイックソートが同程度となっている。これらのことから、バブルソートは相対的に代入回数が少ないものの、いろいろな値が変更されていると考えられる。つまり、配列がまんべんなく更新されると考えられる。一方のバブルソート、マージソートは配列が局所的に頻繁に更新されると考えられる。このことは VAR_UPD メトリクスの結果からもわかる。VAR_UPD がバブルソートだけ非常に高い値になっている。VAR_UPD は変数ごとの更新回数の分散である。つまり、多くの変数が頻繁に更新されていると高い値になる。そのため、バブルソートでは、配列の各要素が頻繁に更新されていることがわかる。

もし仮にバブルソートがマージソート、クイックソートに比べて、絶対的に読みやすいとするならば、AR-trace 長で正規化後の ASSIGNMENT と SUM_UPD メトリクスの低さが、他の VAR_UPD メトリクスなどよりも読みやすさに大きく寄与しているといえる。ただし、今後の様々なプログラムでの追加調査が必要である。また、VMSM は読みやすさ／読みにくさを測定するものではなく、プログラムの特徴を得るためのツールとして利用できる可能性もある。

4. まとめ

本稿では、仮想メンタルシミュレーションモデルを用いて、プログラムの読みやすさの評価を 3 つのソートプログラムに対して行った。その結果、AR-trace の長さで正規化することで、プログラムの実行規模によらず比較できることがわかった。今後の課題として、より多くのプログラムを対象に実験を行い、VMSM の各メトリクスの値の大小と、読みやすさの関係を測定することが挙げられる。また、他のソフトウェアメトリクスも測定し、VMSM と比較することも考えらえる。

参考文献

[1] M. Nakamura,A. Monden,T, Itoh, K. ichi Matsumoto,Y. Kanzaki,and,H. Satoh, “Queue-based cost evaluation for mental simulation process,in program comprehension,” in *Proc. 9th International Software Metrics,Symposium (METRICS 2003)*,September 2003,pp. 351–360.

[2] 石黒 誉久,井垣 宏,中村 匡英,門田 暁人,松本 健一,``変数更新の回数と分散に基づくプログラムのメンタルシミュレーションコスト評価'', 電子情報通信学会技術報告, ソフトウェアサイエンス研究会, Vol.SS2004-32, pp.37—42, November 2004.

[3] H. Tamada, “AddTracer: Injecting tracers into java class files,for dynamic analysis,” December 2004. [Online]. Available:,http://se.naist.jp/addtracer

[4] N. Cowan, “The magical number 4 in short-term memory: A reconsideration of mental storage capacity,” pp. 87–185, October 2001.


```
import java.lang.reflect.Array;
public class BubbleSort implements Sort{
    @Override
    public void sort(int[] array){
        for(int i = 0; i < array.length - 1; i++){
            for(int j = i + 1; j < array.length;
j++){
                if(array[i] > array[j]){
                    int tmp = array[i];
                    array[i] = array[j];
                    array[j] = tmp;
                }
            }
        }
    }
}
```

図２：バブルソートのプログラム

```
public class MergeSort implements Sort{
    @Override
    public void sort(int[] array){
        if(array.length != 1){
            int length = array.length / 2;
            int[] left = new int[length];
            int[] right = new int[array.length
- length];

            for(int i = 0; i < length; i++){
                left[i] = array[i];
            }
            for(int i = length; i < array.length;
i++){
                right[i - length] = array[i];
            }
            sort(left);
            sort(right);

            merge(array, left、 right);
        }
    }

    private void merge(int[] original, int[]
left、 int[] right){
        int leftIndex = 0;
        int rightIndex = 0;
        int index = 0;

        while(leftIndex   <   left.length   &&
rightIndex < right.length){

if(left[leftIndex]<(right[rightIndex]) < 0){
                original[index]               =
left[leftIndex];
                leftIndex++;
            }
            else{
                original[index]               =
right[rightIndex];
                rightIndex++;
            }
            index++;
        }

        int[] rest;
        int restIndex;
        if(leftIndex >= left.length){
            rest = right;
            restIndex = rightIndex;
        }
        else{
            rest = left;
            restIndex = leftIndex;
        }

        for(int i = restIndex; i < rest.length;
i++、 index++){
            original[index] = rest[i];
        }
    }
}
```

図３：マージソートのプログラム

```
public class QuickSort implements Sort{
    @Override
    public void sort(int[] array){
        quickSort(array, 0, array.length - 1);
    }

    private void quickSort(int[] array、 int
firstIndex、 int lastIndex){
        // 適当に値ピボットを選ぶ.
        int   pivot   =   array[(firstIndex   +
lastIndex) / 2];

        int i = firstIndex;
        int j = lastIndex;
        while(true){
            // 左から pivot より大きな値を見つける.
            while(pivot > array[i]){
                i++;
            }
            // 右から pivot より小さい値を見つける.
            while(pivot < array[j]){
                j--;
            }
            if(i >= j){
                break;
            }
            swap(array,i, j);
            i++;
            j--;
        }
        if(firstIndex < i - 1){
            quickSort(array, firstIndex, i - 1);
        }
        if(j + 1 < lastIndex){
            quickSort(array, j + 1, lastIndex);
        }
    }

    private void swap(int[] array, int I, int
j){
        int value = array[i];
        array[i] = array[j];
        array[j] = value;
    }
}
```

図４：クイックソートのプログラム