進捗管理・報告(2020/10/6)

1. 現在取り組んでいること

可逆コンピューティングの研究

2. 進捗状況

・可逆コンピューティングについてと、この分野で行われてきたことの調査

3. 前回からの進捗

・先週提案した幅優先探索を実装した

(探索が成功しなかった場合の処理は考え中でまだ実装できていない)

(また、フラグを返す場合などは2パスにする必要がある)

＊実装したプログラムについて、理論的には問題がないと考えるが、現在のROOPLPPインタープリタでは実行できない(新美くんが対応してくれるみたいです！)。

→ 問題はexploringLimitDepth(深さ優先探索での深さ制限を示す値)を消去する処理の部分でlocal、delocalでdelocalを挟んでいることを処理できないため

具体的には、「local int resultNodeDepth = nil」が、

「delocal int exploringLimitDepth = resultNodeDepth」に対応すると解釈されている。

```
local int resultNodeDepth = nil
call result::computeDepth(resultNodeDepth)
delocal int exploringLimitDepth = resultNodeDepth
uncall result::computeDepth(resultNodeDepth)
delocal int resultNodeDepth = nil
```

・実装したプログラムの計算量

ー時間計算量

サブルーチンである深さ優先探索の計算量

・探索する要素数nに対してO(n)

→stepは同じ辺を最大で2回移動する

辺の数は、要素数nに対して最大n-1

よってstepが実行される回数は2(n-1)でありO(n)

幅優先探索の時間計算量

・サブルーチンである深さ優先探索を探索するノードが見つかるまで呼び出す

→2分木において、要素数nの木の最大の深さは⌊lg n⌋

探索において計算量が最悪であるのは、探索対象が図1の場合

従って深さ優先探索が⌊lg n⌋回実行されるためO(n・⌊lg n⌋) ←タイトではない？

ー空間計算量

深さの情報を保存せず逐次計算する方式に変更したため定数O(1)

ーゴミ出力量

実行途中に必要な変数は全て消去できるため0

ータイトな時間計算量

・二分木の場合

深さ優先探索が繰り返される回数　　深さ優先探索が探索する要素の個数.

$$\left(\sum_{i=1}^{\lfloor \lg n \rfloor + 1} (2^{i-1} + 1)\right) - 1$$

$$= \sum_{i=1}^{\lfloor \lg n \rfloor + 1} 2^{i-1} + \sum_{i=1}^{\lfloor \lg n \rfloor + 1} 1$$

$$= \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} + (\lfloor \lg n \rfloor + 1)$$

$$= 2^{\lfloor \lg n \rfloor + 1} + \lfloor \lg n \rfloor$$

$$= n/2 + \lfloor \lg n \rfloor$$

$$= O(n)$$

$\lfloor \lg n \rfloor - 1 = \lg n - \lg 2.$

$= \lg n/2.$

<!-- 実装した幅優先探索 -->

```
class Node
   Node left
   Node right
   Node parent
   int value
   method constructor(int val, Node left, Node right, Node parent)
       call setValue(val)
       call setLeftChild(left)
       call setRightChild(right)
       call setParent(parent)
   method setValue(int val)
       value ^= val
   method setLeftChild(Node node)
       copy Node node left
   method setRightChild(Node node)
       copy Node node right
   method setParent(Node node)
       copy Node node parent
   method computeDepth(int depth)
       if parent=nil then
           skip
       else
           depth += 1
           call parent::computeDepth(depth)
       fi parent=nil
class Tree
   Node root
   method setRoot(Node node)
       copy Node node root
   // 探索中のノードとその直前に探索したノードから、次に探索するノードを決定する
   // また、探索はノードの深さによって制限される
   method step(Node cur, Node pre, int exploringLimitDepth)
       local Node next = nil
```

```
local int curDepth = nil
call cur::computeDepth(curDepth)
if cur.parent=pre then
    // 通常の判定に加えて、次の判定を追加する
    // 現在探索中のノードの深さが探索深さ制限を超えていたら
    // (>= にしているのは、exploringLimitDepthの値が0でも対応できるようにするため)
    if (cur.left=nil && cur.right=nil) || curDepth>=exploringLimitDepth then
        // 次に探索するノードは直前に探索したノード(親)
        pre <=> next
    else
        if cur.left=nil then
            // 次に探索するノードは右の子
            copy Node cur.right next
        else
            // 次に探索するノードは左の子
            copy Node cur.left next
        fi cur.left=nil
        uncopy Node cur.parent pre
    fi (cur.left=nil && cur.right=nil) || curDepth>=exploringLimitDepth
else if cur.right!=nil then
    if cur.right=pre then
        // 次に探索するノードは親
        copy Node cur.parent next
        uncopy Node cur.right pre
    else
        // 次に探索するノードは右の子
        copy Node cur.right next
        uncopy Node cur.left pre
    fi cur.parent=next
else
    // 次に探索するノードは親
    copy Node cur.parent next
    uncopy Node cur.left pre
fi cur.right!=nil
```

```
        fi (next.parent=cur && (cur.left=next || cur.left=nil)) || (cur.right=nil &&
cur.left=nil) || curDepth>=exploringLimitDepth
        uncall cur::computeDepth(curDepth)
        delocal int curDepth = nil
        pre <=> cur
        cur <=> next
        delocal Node next = nil
    // O(1)の可逆深さ優先探索を行う
    // resultの初期値はroot.leftである必要がある
    // また、探索はノードの深さによって制限される
    // 例：exploringLimitDepthの値が2の場合
    //     → 深さ2以上のノードは探索されない
    method depthFirstSearch(int key, Node result, int exploringLimitDepth)
        local Node pre = nil
        copy Node root pre
        from pre=root loop
            call step(result, pre, exploringLimitDepth)
        until result=root || result.value=key
        if result!=root then
            uncopy Node result.parent pre
        else
            uncopy Node root.left pre
        fi result!=root
        delocal Node pre = nil
    method breadthFirstSearch(int key, Node result)
        local int exploringLimitDepth = nil
        from exploringLimitDepth=0 loop
            exploringLimitDepth += 1
            local Node node = nil
            copy Node root.left node
            call depthFirstSearch(key, node, exploringLimitDepth)
            if node!=root then
                copy Node node result
                uncopy Node result node
            else
```

```
                uncopy Node root node
            fi result.value=key
            delocal Node node = nil
        until result.value=key
        local int resultNodeDepth = nil
        call result::computeDepth(resultNodeDepth)
        delocal int exploringLimitDepth = resultNodeDepth
        uncall result::computeDepth(resultNodeDepth)
        delocal int resultNodeDepth = nil
class Program
    Tree tree
    Node dummy
    Node node1
    Node node2
    Node node3
    Node node4
    int dummyValue
    int key
    Node result
    method main()
        new Node dummy
        new Node node1
        new Node node2
        new Node node3
        new Node node4
        dummyValue -= 1
        call dummy::constructor(dummyValue, node1, nil, nil)
        call node1::constructor(11, node2, nil, dummy)
        call node2::constructor(22, node3, node4, node1)
        call node3::constructor(33, nil, nil, node2)
        call node4::constructor(44, nil, nil, node2)
        new Tree tree
        call tree::setRoot(dummy)
        key += 44
        call tree::breadthFirstSearch(key, result)
```

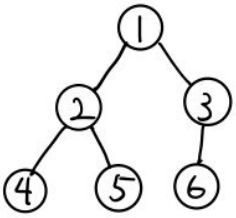
・木構造における可逆探索アルゴリズムの一般的な考察

一列に並べた走査順から連続する2つを抜き出すと、走査における位置が一意に定まるようなアルゴリズムは空間計算量を定数にできる？

→深さ優先探索は本質的に走査順が一意であり空間使用量で最も効率的

幅優先探索は「深さ優先探索の走査順＋探索可能な要素の深さ」で一意にしている

→どちらも走査過程において連続する2つを取り出すと、走査においてどの場所なのかが一意に定まる



深さ優先. 1→2→4→2→5→2→1→3→6.

幅優先. 1+深さ1 → 2+1 → 1+1→3+1→1+2

→2+2→4+2→2+2→5+2→2+2→1+2

→3+2→6+2

4. 今後の課題

・Analyzing Trade-offs in Reversible Linear and Binary Search Algorithms で提案されている線形探索の問題点を発見し、改善案を提案する。