灘校パソコン研究部部誌
2015 年文化祭号

NPCA（灘校パソコン研究部）

目次

目次 3

第 I 部 Hunting Birds 7
1 Introduction 9
2 Environment 10
3 Attack Outline 11
4 Establish Mapping 11
5 Proof of Concept 13
6 Prevention 13
7 Conclusion 13
8 Acknowledgments 14

References 15

第 II 部 かしこい機械の育て方
reyk(@reyk429) 17

第 I 章 はじめに 19

第 II 章 機械学習入門 21
1 奇怪な機械 21
2 ネットで得たデータ 22
3 口調という特徴 23
4 顔の detail が似ている 25
5 ニューロン導入論 ~ 脳 gain, no pain~ 28

第 III 章 今後の展望 41

第 IV 章 おわりに 43

参考文献 45

第 III 部　木にノードを加えて軽く炒めて食べたい
cloud 47

第 I 章　木とは 49

第 II 章　二分木 51

第 III 章　回レ！ 53

第 IV 章　赤黒木 55
1 つまり 2

第 IV 部　Web-othello についてのお話と小噺
KA-FU- & hinata 3

第 I 章　まえがき 5

第 II 章　Web-othello 作成の流れ 7
1 オセロ本体 7
2 AI 8
3 めでたくない 9

第 III 章　本題 11
1 Web、襲来 11
2 動かない、ページ 11
3 決戦、InternetExplorer 12
4 過去の自分が造りしもの 12
5 jQuery、来訪 16
6 瞬間、処理、重ねて 18
7 SQL インジェクション、そして 27
8 せめて、ログらしく 28

第 IV 章　まとめ 31
1 世界の中心で Lisp を叫んだもの 31

第 V 章　あとがき 33

参考文献 35

第 V 部　木に対する一般的なテク達
@okuraofvegetabl 37

第 I 章　はじめに 39

第 II 章　知識編 41
1 木の用語 41
2 木の問題について 44
3 データ構造 44
4 木に対するテクニック 55

第 III 章　実践編 63
1 Propagating tree (Codeforces Round #225 Div1 C) 64
2 On Changing Tree (Codeforces Round #232 Div1 C) 68
3 Running Away from the Barn
(USACO 2012 December Gold) 70
4 A and B and Lecture Rooms
(Codeforces Round #294 Div2 E) 73
5 Xenia and Tree (Codeforces Round #199 Div2 E) 76

第 IV 章　おわりに 79

参考文献 81

第 I 部

Hunting Birds

Hunting birds

Yuki Koike

NADA Junior and Senior High School
8-5-1, Uozaki-Kitamachi, Higashinada-ku, Kobe-shi, Hyogo 658-0082, JAPAN
poteticalbee@gmail.com

Abstract Stack Smashing Protection (SSP) is one of the oldest and fundamental protections against memory corruption exploits. Employing stack canaries to detect malicious corruption, SSP has proved to make exploiting memory corruption bugs to be more difficult. Stack canaries verify if a buffer has been overflown by checking the integrity of a value stored immediately after the buffer. Although some papers presented some trivial methods in Linux to defeat stack canaries, they require certain function pointers to be overwritten or only work in certain circumstances. Therefore, they are not so useful as a versatile exploit technique in the present because they require rare configuration and environments that are unlikely to be found in the real world. This paper proposes a new technique called "Master Canary Forging", which provides practical use in terms of adaptability and feasibility for killing stack canaries with no need for using other exploitation methods which modify other function pointers such as GOT Overwrites or SEH Overwrites.

1 Introduction

Stack Smashing Protection (SSP) is a security exploit mitigation that compilers use to detect during runtime whether a stack frame has been overwritten and to terminate the process if it has in order to mitigate the possibility of arbitrary execution of code by an attacker. The idea and first implementation of SSP was released in 1998 and has been modified and improved on ever since. A stack based buffer overflow is a vulnerability that allows an attacker to write passed the intended memory space allocated for a variable into the area on a call stack. For programs compiled with SSP, the attacker would have to overwrite pass the canary before reaching the frame pointer and return address on the call stack. It could be said that this is the most straightforward and easiest way to exploit memory corruption bugs since it is possible to directly tamper with call stacks, which are responsible for the program flow. This is the motivation for creating a method of bypassing these canary checks.

A stack canary can either be a random value hard to predict value that should not be able to be guessed or brute-forced, or a terminator value that has been carefully selected to prevent overwriting any data

passed it. If the random value would be able to be predicted or if the attacker could write pass the terminator value, then the attacker would be able to successfully overwrite the call stack and take control of the program. Therefore, it is necessary make sure that the random value is truly unpredictable and that the terminator value is unable to be written passed. Currently, stack canaries can be separated into three major groups: a) terminator, b) random, and c) random XOR. Most modern Linux distributions use random canaries. In the current glibc implementation, *_dl_setup_stack_chk_guard* in ld.so stores random values generated from random devices such as */dev/random* or */dev/urandom* in the fixed locations of the TLS segments or the BSS segments when a program is running. These values are called master canaries. Programs are compiled to be able to read these master canaries from the designated places in accordance with the design of glibc. Thus, different random values are generated every time the program is launched. These values are taken from the master canaries and placed after the allocated program variables and before the base pointer and return address when a function is called. When the function ends, the canary value is then checked to make sure that it matches with the master canary to find out that it has not been changed.

There are mainly two methods to bypass stack canaries with stack based buffer overflows: a) completely avoid the canary validation by abusing other function pointers and b) bypassing the canary validation by somehow overwriting it with the correct value. The techniques covered in [1], [2], and [3] give examples of a) and b). Master Canary Forging, described in this paper, does not deal with the first type and, as the name suggests, corrupts the master canaries. This is opposed to the techniques explained in [1] and [2], that take the opposite approach by finding out the values of the master canaries and overwriting the canary values to match them. This attack would be much more difficult if random XOR canaries were used, as they are in Windows, as a vulnerability that leaks stack address would also be necessary. The original idea for Master Canary Forging is mentioned in [4], however, there are only works on non-ASLR systems. I have improved on this idea to make Master Canary Forging work even if ASLR enabled. This is obvious when the target architecture does not support reading from the TLS segment because the BSS segment is always mapped at fixed addresses. Hence, this paper covers only the x86_64 architecture which uses the TLS segments for reading canaries.

2 Environment

This paper assumes the following environments:

- Linux Kernel 3.19
- Glibc 2.21
- GCC 4.9.2
- x86_64

Note that Master Canary Forging runs regardless whether or not DEP, ASLR, PIE, and RELRO are enabled.

3 Attack Outline

Master Canary Forging consists of the following phases:

1. Establish Mapping: Create a mapping by invoking *mmap* to make it and the TLS segment successive.
2. Master Canary Overwrite: Overflow a buffer in the mapping in order to forge a master canary.
3. Stack Canary Overwrite: Overflow a stack based buffer overflow.

Therefore, Master Canary Forging requires three conditions: the capability of calling *mmap* with specific arguments, the capability of overflowing a buffer in the mapping, and the capability of causing a stack based buffer overflow.

4 Establish Mapping

- The *addr* argument is NULL.
- *MAP_FIXED* and *MAP_32BIT* are not specified by the *flags* parameter.
- *–addr-compat-layout* is not indicated by the *setarch* command.
- The value of *RLIMIT_STACK* is not *RLIM_INFINITY*.
- */proc/sys/vm/legacy_va_layout* is set as 0.

In the *mmap* implementation of Linux Kernel 3.19 x86_64, if these conditions are satisfied, then mapping addresses are computed as follows in *unmapped_area_topdown*.

```
/* Check highest gap, which does not
   precede any rbtree node */
gap_start = mm->highest_vm_end;
if (gap_start <= high_limit)
    goto found_highest;


...


while (true) {
    /* Visit right subtree if it looks
       promising */
    gap_start = vma->vm_prev ? vma->
        vm_prev->vm_end : 0;
    if (gap_start <= high_limit && vma->
        vm_rb.rb_right) {
        struct vm_area_struct *right =
        rb_entry(vma->vm_rb.rb_right,
        struct vm_area_struct, vm_rb);
        if (right->rb_subtree_gap >=
            length) {
            vma = right;
            continue;
        }
    }

check_current:
    /* Check if current node has a suitable
       gap */
    gap_end = vma->vm_start;
    if (gap_end < low_limit)
        return -ENOMEM;
    if (gap_start <= high_limit &&
        gap_end - gap_start >= length)
        goto found;

    /* Visit left subtree if it looks promising
       */
        if (vma->vm_rb.rb_left) {
            struct vm_area_struct *left =
```

```
                rb_entry(vma->vm_rb.
                    rb_left, struct
                    vm_area_struct,
                    vm_rb);
            if (left->rb_subtree_gap >=
                length) {
                vma = left;
                continue;
            }
        }

    /* Go back up the rbtree to find next
        candidate node */
    while (true) {
        struct rb_node *prev = &vma->
            vm_rb;
        if (!rb_parent(prev))
            return -ENOMEM;
        vma = rb_entry(rb_parent(prev),
        struct vm_area_struct, vm_rb);
        if (prev == vma->vm_rb.rb_right)
            {
            gap_start = vma->vm_prev ?
                vma->vm_prev->
                vm_end : 0;
            goto check_current;
        }
    }
}

found:
    /* We found a suitable gap. Clip it with the
        original high_limit. */
    if (gap_end > info->high_limit)
        gap_end = info->high_limit;

found_highest:
    /* Compute highest gap address at the
        desired alignment */
    gap_end -= info->length;
    gap_end -= (gap_end - info->
        align_offset) & info->align_mask;

    return gap_end;
```

Looking at this, in the above case, *mmap* adopts the topdown method which establishes a mapping at the highest address in free space capable of being mapped. On the contrary, it selects the bottomup method if those conditions not satisfied. In either case, a new mapped region is always adjacent to some region which has already been mapped unless the highest region is unmapped. That means that it is possible to overwrite all of the master canary values if the mapped area for a buffer overflow is located in front of the TLS segment, and all areas between them are consecutive and writable. Although these requirements may at first seem difficult to be met, they are actually fulfilled in most applications. This is due to the front of the TLS segment being empty at all times unless the application makes its own call of *mmap* because the TLS segment is the second to last to be mapped by *_dl_allocate_tls_storage*, and meanwhile there is no hole created by munmap between any regions, and the heap region, which is the last, is allocated by another system call, *sbrk*. In this scenario, the attacker must then consider how to directly call *mmap*. *mmap* is essentially a pretty low level system call, so few applications but some exceptions using *mmap* in their code. However, there is a widely used glibc function which calls *mmap*: *malloc*. *malloc* is well known to deal with allocated requests that exceed *MMAP_THRESHOLD* bytes by creating a new pool with *mmap* [5]. This trait enables an attacker to make a heap area continuing to the TLS segment if it is possible to send

specific sized allocation requests. Summarizing the above, the conditions mentioned in chapter 3 can be rephrased in the following:

- The capability to control allocation sizes of *malloc*
- A heap based buffer overflow
- A stack based buffer overflow

5 Proof of Concept

Code 1 poc.c

```
/*
 * gcc poc.c -fstack-protector-all -Wl,-z,
       now,-z,-relro
 */

#include <stdio.h>
#include <stdlib.h>

void stack_overflow(void) {
    char stack_buf[16];

    fread(stack_buf, 1, 48, stdin);
    return;
}

int main(void) {
    size_t alloc_size = 0;
    size_t read_size = 0;
    char *heap_buf;

    if (scanf("%zu", &alloc_size) != 1) return
          -1;
    if (scanf("%zu", &read_size) != 1) return
          -1;

    heap_buf = (char*)malloc(alloc_size);
    if (!heap_buf) return -1;

    fread(heap_buf, sizeof(char), read_size,
          stdin);
    stack_overflow();
    free(heap_buf);
    return 0;
}
```

Code 2 exploit.py

```
print "{}".format(0x21000)
print "{}".format(0x23720)
print "a" * (0x23720+0x30)
```

The binary is available at: https://github.com/potetisensei/MasterCanaryForging-PoC/.

6 Prevention

I recommend using random XOR canaries in order to prevent this exploitation method and to increase the sources of entropy enough to ensure that they are truly unpredictable.

7 Conclusion

This paper describes a new exploitation technique to defeat stack canaries called Master Canary Forging, which uses *malloc* to utilize the properties of *mmap* to establish successive memory mappings in order to overwrite the master canary values. It is architecture dependent and currently has been proven to work on the x86_64 architecture with the latest stable Linux Kernel, glibc, and gcc. There are some other cases where Master Canary Forging will succeed on other architectures but is out of scope of this paper.

8 Acknowledgments

I thank Isaac Mathis for his kind and swift feedback and proofreading, and Yuma Kurogome who double-checked my PoC.

References

[1] Paul Rascagneres. Stack Smashing Protector http://www.hackitoergosum.org/2010/HES2010-prascagneres-Stack-Smashing-Protector-in-FreeBSD.pdf

[2] Ben Hawkes. Exploiting OpenBSD http://inertiawar.com/openbsd/hawkes_openbsd.pdf

[3] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/defeat-stackguard.pdf

[4] Hagen Fritsch. Stack Smashing as of Today https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

[5] malloc(3) - Linux man page http://linux.die.net/man/3/malloc

第 II 部

かしこい機械の育て方

reyk(@reyk429)

第 1 章

はじめに

こんにちは。68 回生の reyk と申します。宗教上の理由により部誌に寄稿するのは今年で初めてなのでほとんどの方は初めましてですね。最近になってハンドルネームを変えたので初めましてではない方も初めましてかもしれません。

さて、この記事のテーマは「機械学習」です。みなさんも、最近流行りの「ビッグデータ」というフレーズ、一度は耳にしたことがあると思います。Google の検索エンジンをはじめ、Amazon の商品推薦システムとか、NHK の twitter トレンドとか、ああいうやつです。しかし、耳にしたことがあっても、その中身・仕組みが何なのかよく知らない、という人も少なくはないと思います。この記事では、「機械学習」というものに少し足を踏み入れて、ビッグデータというものの裏でどのような技術が使われているのか、ということを紹介していきたいと思います。

第 II 章

機械学習入門

1　奇怪な機械

「機械学習」というワードが突然出てきましたが、学習する機械、と聞くと少し奇妙な感じがするかもしれません。機械学習という字面から何となく意味は分かりそうですが、その意味するところは何なのでしょうか。例によってとりあえず Wikipedia 先生 [11] に聞いてみましょう。

> 機械学習（きかいがくしゅう、英: machine learning）とは、人工知能における研究課題の一つで、人間が自然に行っている学習能力と同様の機能をコンピュータで実現しようとする技術・手法のことである。

... 字面以上の情報はあまり得られなかったみたいなので、具体例を挙げて説明します。

人間は、日頃から無意識のうちに多くのことを学習しています。例えば、普段何気なく行っている、歩く、つかむ、といった動作も、もちろん生まれてすぐできたわけではありません。繰り返し動作を試み、失敗、成功を積み重ねることで、次第に「コツ」のようなものを身につけ、失敗することなくできるようになるのです。これも一種の学習です。「文字を読む能力」や「音・声を聞き分ける能力」などの認識能力、「ボールの飛ぶ方向を予測する能力」や「相手の感情・反応を推測する能力」といった予測能力についても、認識・予測の「コツ」、なんとなくの「パターン」を学習によって見つけることで可能となっているのです。機械学習とは、このような人間の学習プロセスを真似ることで、機械に認識能力や予測能力を身につけさせよう、という試みなのです。

さて、人間の学習プロセスを真似する、ということですが、具体的にはどのように行うのでしょう。子供に勉強を教える、という例と比較して考えると、

1. 問題を与えて解かせる
2. (子供|機械) から返ってきた答えを正解と比較する

3. 答え合わせをもとに、(考え方|処理方法) を修正させる

という感じになります。
機械学習というのは基本的にこのようなプロセスで行われます[1]。
ただ、人間の高度な学習能力は未だ完全に解明されているわけではなく、それを機械で実現するのは難しいです。そのため、そこは機械の処理能力を生かして、才能を努力で補うように学習能力の低さをデータの量で補います[2]。ここがビッグデータと呼ばれる所以です。

まとめると、機械学習では、機械に与えるべき大量の問題集であるデータを集め、機械の問題に対する処理方法のモデルを作ることが課題となります。

ところで、このまま抽象的な感じで行くと終始ふわふわした説明だけになってしまいそうなので、以下では問題を「文字認識」ということに限定し、実際に文字認識のプログラムを作る際の流れに従って話を進めていきたいと思います。途中のソースコードは python で記述しています。

2　ネットで得たデータ

機械に問題を解かせることを考える前に、まずは解かせるための問題、つまりここでは文字の画像と、正解である文字の種類のラベルが必要です。自前で用意しよう、と思うかもしれませんが、いっぱい集めるのは一苦労です (特にラベル付け)。特に機械学習においては、前述の理由からデータは多ければ多いほど良く、またある程度の精度を実現するためには最低でも 10000 くらいのデータは必要です。
しかし、時代は 21 世紀、インターネット上には、文字画像や人の顔の画像、天気のデータから株式市場のデータに至るまで、様々なデータが保存されています。機械学習の際にはこうしたデータを利用すると便利です。今回は ETL 文字データベース [4] のひらがな約 6000 データ、カタカナ約 10000 データを使わせて頂きました。このデータセットにはその画像がどの文字であるか、というラベルもあらかじめ含まれています。

ところで、データを全て学習用に使ってしまっては、学習をさせた後にちゃんと学習が成功したのか、ということを評価することができなくなってしまいます。よって通常はデータを学習用のデータとテスト用のデータの二つに分け、学習後にテストを行うことで学習結果を評価します。この学習用のデータを**訓練用データ**、テスト用データを**評価用デー**

[1] 正確にはこのように正解と比較して処理方法を修正するような学習方法は「教師あり学習」と呼ばれ、それに対して正解の与えられないクラスタリングなどの「教師なし学習」という学習方法も存在します。
[2] 現在の機械であれば 1 秒間に 10^8 回程度の計算が可能なので、処理量にもよりますが 1 秒間に 100 ～ 100000 程度のデータは処理できます。

タと呼びます。今回は全データの 5% を評価用データとして使うこととします。

3　口調という特徴

特徴抽出―概略編

さて、次は問題の解き方、文字の認識方法について考えます。機械学習の定義に従い、人間の学習方法について考え、それを機械に適用することを考えます。

文字だと少し分かりづらいので、初めに顔認識について考えてみましょう。私たち人間は人の顔を見て、どのようにその人が誰であるかを認識しているのでしょうか。髪の色・長さ、眉毛の太さ、目の大きさ、鼻の高さ、顔の形... これらの判断材料を総合的に評価し、あらかじめ頭の中にたくさん保存してある顔のイメージのうち最も似ている顔を、その人の顔だと認識し、その顔に関連付けられている人をその人だと認識するのです。ここではこれらの判断材料のことをまとめて**特徴**と呼び、一つ一つの判断材料を特徴の成分と呼ぶことにします。

文字の話に戻りましょう。文字にはどのような特徴成分があるでしょうか。線の太さ、線がつながっているかなどは、書く人によって変わるため、あまり関係なさそうに思えます。そうすると、文字の特徴は文字の形ぐらいしかなさそうです。ただ、文字の形といってもそれは一言で表せるものではありません。それぞれの成分は機械に理解できる形式、つまり数値に変換しなければなりません。よって最終的に特徴はいくつかの数値の組として表されることになります。

単純化して数値に落とし込むために、とりあえず画像を小さい格子単位に分割します。さらにその中の線分を単純化して、その線分が大体どの向きであるか、というのを表すために、次の図のような単純な方向成分に分解することを考えます。

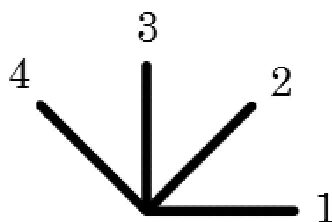


図 II.1

このそれぞれの方向成分の長さ (強さ) を数値化すれば、文字の形の情報を数値の組で表

すことができそうになってきました。

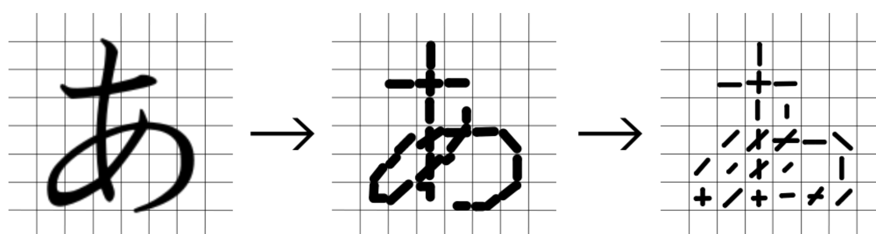


図 II.2　数値化の流れ

このようにして、元の文字の形の情報を (格子の数)×4 個の数値として表すことができました。このように元のデータから特徴を表す数値を計算する過程を**特徴抽出**と言います。

特徴抽出―実装編

文字認識におけるこのような特徴抽出の方式を**加重方向指数ヒストグラム方式**と呼びます。概略編では詳細を省略しましたが、実際の実装では参考文献 [6] に従い、以下のような手順で特徴抽出を行いました。

1. 画像の整形 (2 値化[3]、ノイズ除去[4])
2. 画像の大きさの正規化[5]
3. 輪郭の平滑化
4. 輪郭線抽出
5. 分割化と各格子内の各方向成分の抽出
6. 加重方向指数ヒストグラムの計算

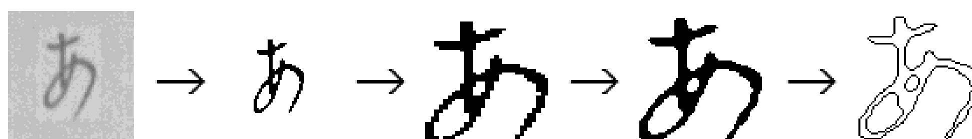


図 II.3　画像の整形から輪郭線抽出までの流れ

形の情報は各格子内の方向成分として表現しているため、画像の大きさを正規化して文字の位置を合わせる必要があります。

[3] 画像の色を白と黒だけにする
[4] 文字以外の不要な情報の除去
[5] 文字が画像にぴったり収まるように調整し、大きさを揃える

また輪郭線以外の部分は方向を検出できないため、輪郭線のみについて方向成分を計算しています。これによって同時に太さの情報を無視することができます。

方向成分の抽出では、斜めも含めて隣接する黒画素についてその方向を方向成分として抽出しています。

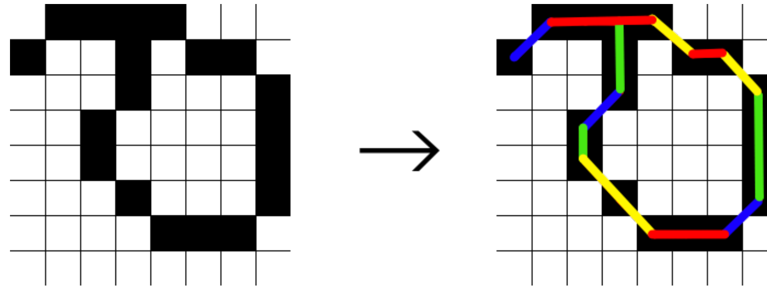


図 II.4　方向成分抽出。この図では図 II.1 の番号付けを使うと (成分 1, 成分 2, 成分 3, 成分 4) = $(6, 3, 6, 4)$ となる

各格子 (ここでは 7×7) の方向成分を抽出した後は、適切な重みづけ[6]をして周辺の格子の方向成分を足し合わせることで、データごとに多少位置がずれても大幅な数値のずれが生じないようにします。

以上の操作をして代表のマスを 4×4 個選ぶことにより、最終的に $4 \times 4 \times 4$ 個の数値として文字の形を表現することができます。

4　顔の detail が似ている

類似度を定義する

前節で人間の認識方法について述べたことをもう少し詳しく説明すると、人間は頭の中でインスタンス[7]をいくつかのカテゴリーに分類していて、新しいインスタンスに出会った際には、似ているインスタンスが属しているようなカテゴリに分類している、と言うことができそうです。文字認識であれば各文字データがインスタンス、文字の種類がカテゴリに当たります。この分類方法を機械に適用することを考えます。(このように分類の仕方を学習する機械のことを特に**分類器**と呼びます。)

そこで、インスタンス同士が似ている度合い、類似度について考えてみます。もし、特徴を表す数値が 1 つなら、数値が近いほど似ていると言えるのは明らかです。この自然な拡張として、数値が 2 つ、3 つと増えていっても、特徴同士の近さ、距離によって類似度が

[6] 参考文献 [6] ではガウスフィルターを用いています。
[7] ある一人の顔や一つの文字など一つの具体的な認識対象をここではこう呼ぶことにします。

定義できそうに思えます。

ここで、2 つの N 個の数値の組 $(A_1, A_2, ..., A_N), (B_1, B_2, ..., B_N)$ の距離を次のように定義します。

$$Distance(A, B) = \sqrt{\sum_{k=1}^{N}(A_k - B_k)^2}$$

これも 1 次元、2 次元での距離の定義の自然な拡張といえます。以下ではインスタンスの類似度としてその特徴を A,B としたときの $Distance(A, B)$ を用いることとします[8]。

さて、インスタンス同士の類似度が定義できたところで、分類すべきカテゴリはどのように判断すればよいでしょうか。「似ているインスタンスが属しているようなカテゴリ」の解釈によって、色々な方法が考えられますが、ここでは

1. 新しいインスタンスとの類似度の小さい順から複数のインスタンスのうち、最も多くのインスタンスが属しているようなカテゴリに分類する

という考え方に従って文字の分類を行ってみたいと思います。

K 近傍法による分類

新しいインスタンスに対して、今までに入力された全てのインスタンスとの類似度を計算し、似ているものから K 個を取り出します。その中で最も多くのインスタンスが属しているようなカテゴリに、新しいインスタンスを分類します。なお、下のソースコードでは簡潔さのために一部実装の詳細を省略した部分があります。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from random import shuffle

#距離関数の定義
def Distance(a,b):
    s = 0
    for i in range(len(a)):
        s += pow(a[i] - b[i], 2)
    return s
```

[8] 類似度が小さいほど似ている、ということに注意してください。

```
#カテゴリは [0,56) の数字で表されており、
#それぞれ JIS X 0201 におけるカタカナの文字コードに対応している ([166,222))

if __name__ == '__main__':
#   feature_list = [(特徴のリスト, ラベル) in 全てのデータ]
    shuffle(feature_list) #ランダムに訓練用データと評価用データに分ける
    N = len(feature_list) #データセット数
    K = 20
    correct = 0 #正答数
    #番号 [0, N - N/20) のデータを訓練用に、
    #番号 [N - N/20, N) のデータを評価用に用いる
    for index in range(N - N/20, N):
        #全ての距離（類似度）を計算
        distance = [(Distance(feature_list[index][0], feature), label)
                        for (feature, label) in feature_list[0, N - N/20]]
        distance.sort() #類似度の小さい順にソート
        appeared = [0] * 56 #各カテゴリの出現回数
        for i in range(K):
            appeared[dis[i][1]] += 1
        #最も多く現れたカテゴリを採用
        result = appeared.index(max(appeared))
        #正答数のカウント
        if result == feature_list[index][1]:
            correct += 1
    print correct, '/', N/20
```

実行結果は以下のようになりました。

データセット	正答サンプル数	全サンプル数	正答率
ひらがな (ETL4)	268.6	305	88.1%
カタカナ (ETL5)	513.6	530	96.9%

どちらのデータセットでもかなり高い正答率が実現されていることが分かります。

以上のような分類法を **K 近傍法**と呼びます。

K 近傍法の問題点

前節では距離によって類似度を定義し、それに基づいた K 近傍法という分類法を紹介しました。今回の場合はこの方法でうまくいきましたが、この K 近傍法にはいくつか問題点があります。

まず、K 近傍法は訓練データ全てとの距離の計算を行うため、計算量がかなり大きいです。適宜枝刈りをして計算量を減らすことは可能ですが、訓練データを全て保存しておく必要があるのは変わりません。これは訓練データが多いほど学習能力の上がる機械学習においては中々厳しい制約です。

次に、成分の重みづけが違う場合に、あまり良い結果が得られない、ということがあります。例えば、人の特徴として、身長と体重を考えた時に、身長を m、体重を kg で表したとします。すると、(身長, 体重) = $(1.7, 62.3), (1.7, 63.3), (2.7, 62.3)$ という 3 人の特徴を考えた時、1 人目と 2 人目、1 人目と 3 人目の類似度はどちらも 1.0 となり、同じだけ似ているということになってしまいます。これがおかしいのは明らかでしょう。

さらにもう一度人の特徴を例にとって考えてみます。(身長 (cm), 体重 (kg)) = $(170, 70), (170, 100), (170, 130)$ という 3 人の特徴について、距離で考えると 1 人目と 2 人目、2 人目と 3 人目の類似度は同じになりますが、人間なら肥満度という点から考えて 2 人目と 3 人目の方が似ていると考えるのが普通でしょう。このように、距離による類似度は成分同士の比率による類似度などをうまく表現できないという問題点があります。

では、どのようにすれば改善できるのでしょうか。一つには、データを正規化する[9]、コサイン類似度を用いることなどが考えられます。特徴空間を平面によって 2 つのグループに分割する SVM という手法も有名ですが、これらは今回は説明しません。次の節では、幅広い応用の効く**ニューラルネットワーク**というものを紹介して、この記事を締めくくることとします。

5　ニューロン導入論 ～ 脳 gain, no pain～

ニューラルネットワーク導入

ここで一度、機械学習の定義に戻って考えてみましょう。機械学習とは、人間の学習プロセスを機械に真似させることで、問題を解こうとする試みでした。これを突き詰めて、

[9] 数値を 0 ～ 1 に揃えること

学習プロセスにおいて人間の脳みその中で起こっていることを機械で表現してみよう、というのが、これから説明するニューラルネットワークの考え方です。

まず、人間の脳みその働きは、以下のようなニューロンという神経細胞によって成り立っている、ということが解明されています。

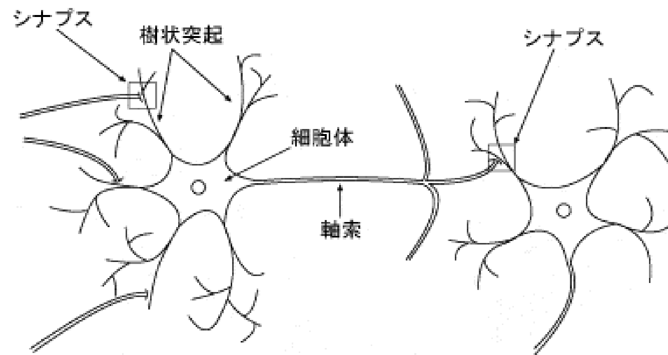


図 II.5　ニューロン

これをモデル化すると次のようになります。

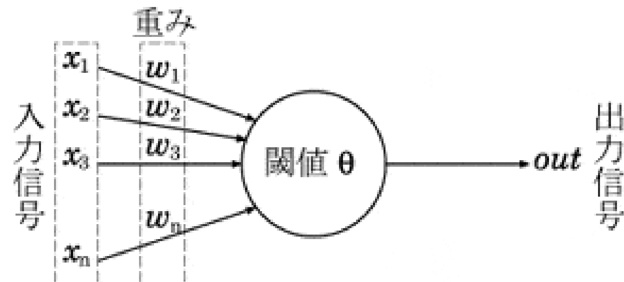


図 II.6　ニューロン模式図

この図は、一つのニューロンが辺の重みによって重みづけされた他のニューロンの出力を入力として受け取り、その合計がある閾値 θ を超えると、何らかの信号が出力される、という過程を表しています。このような繋がりが集まって一つの巨大なネットワークを形成しているのです。

これは機械においては、**辺 (edge)** によって繋がる**頂点 (node)** として表現することができます。信号の伝達については、辺によってつながった他の頂点の出力を $X_1, X_2, ..., X_n$、それぞれの入力辺の重みを $W_1, W_2, ..., W_n$ とし、頂点の内部関数を f として、

$$out = f(\sum_{k=1}^{n} X_k W_k)$$

として出力 out を定義します。ここで内部関数 f は、次のシグモイド関数と呼ばれるものとします。この理由は後述します。

$$f(x) = \frac{1}{1 + e^{-x}} \tag{II.1}$$

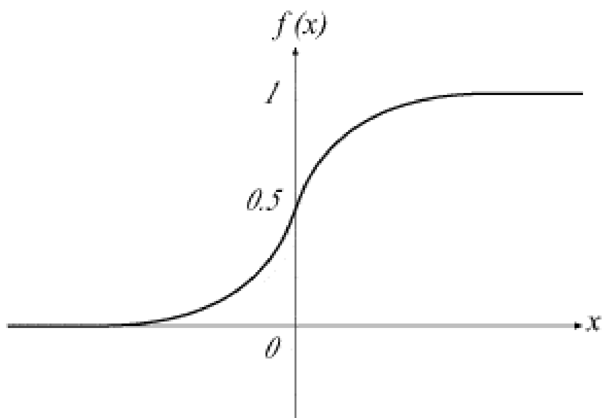


図 II.7　シグモイド関数

さらに、ニューロンの繋がりを単純化し、頂点たちが層をなしていると考え、それぞれの頂点を**入力層**、いくつかの**隠れ層 (中間層)**、**出力層**に分けます。また信号の伝わる方向は一方向のみで、内部ループは存在しません。このようなネットワークモデルを**多層パーセプトロン**と呼び、今回は特に一つの隠れ層と入力層、出力層からなる 3 層パーセプトロンを扱います。

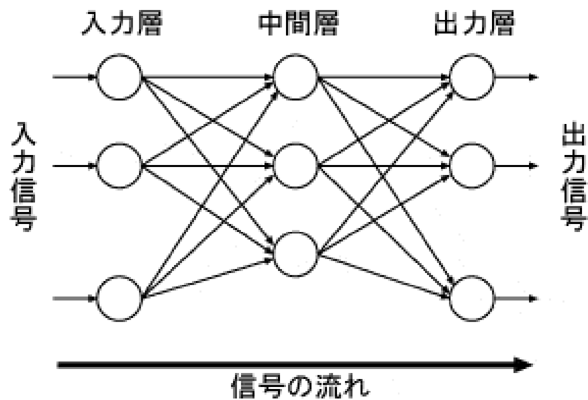


図 II.8　3 層パーセプトロン

入力層の頂点に数値を入力すると、上記の式に従ってそれぞれの頂点の入力値・出力値が計算され、出力層の出力値が全体としての出力となります。機械学習では、インスタンスの特徴の各成分を入力頂点に、各カテゴリを出力頂点に割り当て、特徴を入力すると各カテゴリのスコア[10]が算出されるようにします。この数値の最も高いカテゴリが、このインスタンスの属するカテゴリと判定される、ということになります。

最初の辺の重みづけはランダムに割り当てます。そのため、出力される数値に特に意味はありません。訓練データを入力する度に、出力された数値を正解 (そのインスタンスがどのカテゴリに属するか) と比較し、出力される数値が正解に近づいていくように辺の重みづけを変更します。これを繰り返すことで、最終的に正解に近い結果を出力するようなネットワークが構成されることになります。以上のような学習方法のことを**誤差逆伝搬法 (バックプロパゲーション)** と呼びます。

(図 II.5〜II.8 の引用元:[7])

誤差逆伝搬法

「出力される数値を正解に近づける」というのを言い換えると、「出力と正解との誤差を小さくする」ことが目的である、と言えます。これで、解くべき問題をより明確な問題に置き換えることができました。ニューラルネットワークにおけるこの問題について考える前に、基本的な最小化問題に対するアプローチを説明します。

[10] そのカテゴリに属している可能性、度合いを表す数値

最急降下法

次のような簡単な式を最小化することを考えます。

$$f(x) = x^2 - 4x + 5$$

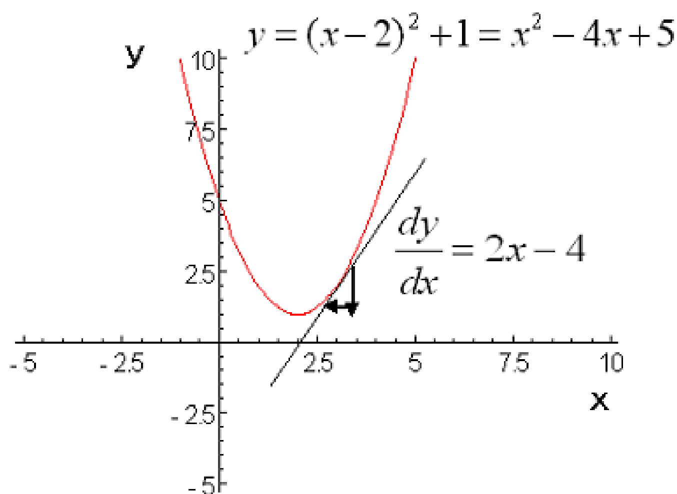


図 II.9　図の引用元:[9]

今、x の値を適当に決め、その点での接線の傾きが求まっているとします。そして、ここから x の値を修正することで、$f(x)$ の値を小さくしていくことを考えます。すると、接線の傾きが正なら x を少し小さく、負なら少し大きくすることで、$f(x)$ の値が小さくなっていきそうだということが分かります。そして、最小値付近では、接線の傾きは 0 に近づいていくことも分かります。

以上の考えをもとに、学習率 η (ある程度小さい数) というものを導入して、

$$x \leftarrow x - \eta f'(x)$$

という式によって x を更新していきます。$f'(x)$ が大きい時には x は大幅に変化します。逆に $f(x)$ がほぼ最小化されている場合には x はほぼ変化しないため、ほぼ一定に保たれます。

この方法によって $f(x)$ は次第に小さくなっていきますが、必ずしも最小化されるわけではなく、実際には極小値にしかなりません。しかし、今は値の最小化ということよりも値を小さくしていくことに関心があるため、このことはあまり問題にはなりません。ただ、機械学習において、初期値によっては悪い極小値に陥り、どれだけ訓練データを増やして

も良い結果が出ないということもあり得る、ということは頭に留めておく必要があります。

同様にして、n 変数の関数 $f(x_1, x_2, ..., x_n)$ についても、そのそれぞれの変数による偏微分[11]を考え、

$$x_1 \leftarrow x_1 - \eta \frac{\partial f(x_1, x_2, ..., x_n)}{\partial x_1}$$
$$x_2 \leftarrow x_2 - \eta \frac{\partial f(x_1, x_2, ..., x_n)}{\partial x_2}$$
$$...$$

という更新を繰り返すことで、f を最小値に近づけることができます。このように目的関数 f の値を小さくしていく方法を**最急降下法**と呼びます。

卑近な例で言うと、これは山の中で遭難してしまった際に、下の方に降りていけばいつか町に出るだろう、という考え方と同じです。先ほどの注意と同様、この方法は盆地が存在すると詰みます。

誤差関数の最小化

話を元に戻しましょう。これから話を進める上で、いくつかの記号を導入します。ここで関数 $f(x)$ は式 II.1 で定義したシグモイド関数です。

- 入力層、隠れ層、出力層の頂点数をそれぞれ N_{in}, N_{hid}, N_{out} とする。
- 入力 $(I_1, I_2, ..., I_{N_{in}})$ に対する隠れ頂点の出力を $(H_1, H_2, ..., H_{N_{hid}})$、出力頂点の出力を $(O_1, O_2, ..., O_{N_{out}})$ とする。
- 入力頂点 i から隠れ頂点 j への辺の重みを W_{ij} とし、隠れ頂点 j から出力頂点 k への辺の重みを W'_{jk} とする。
- $S_j = \sum_{i=1}^{N_{in}} I_i W_{ij}$ とすると、$H_j = f(S_j)$ である。
- $T_k = \sum_{i=1}^{N_{hid}} H_i W'_{jk}$ とすると、$O_k = f(T_k)$ である。
- 入力に対する正解を $(A_1, A_2, ..., A_{N_{out}})$ とする。これはこのインスタンスが属するカテゴリを cat としたとき $A_{cat} = 1$ であり、それ以外について $A_k = 0$ となる。

色々書いていますが、今までに述べたことを記号を導入して整理しただけです。

[11] 偏微分とは、関数をその変数だけに依存する関数だと考え、その他の変数を定数と考えて微分することです。これも、1 変数の微分と同様に他の変数が固定された状態での $f(x_i)$ という関数の接線の傾きと捉えることができます。

さて、今最小化[12]したいのは誤差関数です。これを E とおき、

$$E = \frac{1}{2}\sum_{k=1}^{N_{out}}(A_k - O_k)^2 \tag{II.2}$$

とします。$\frac{1}{2}$ は微分した際の係数を消すため、2 乗しているのは正負をまとめて扱うためです。このような誤差の定義の仕方は機械学習においてしばしば用いられます。

先ほど説明した最急降下法を用いることを考えましょう。E を最小化したいので、E を変化させたい変数で偏微分すれば、その変数の変化させるべき値が求まります。ここで A_k は定数なので、これは $O_1, O_2, ..., O_{N_{out}}$ の関数になっています。ところが、O_k は直接変化させることはできず、変化させるのは W_{ij}、W'_{jk} なので、E をこれらで微分した結果を求める必要があります。

まずは W'_{jk} の修正すべき差分 $\Delta W'_{jk}$ を求めましょう。微分の連鎖律[13]を用いると、

$$\Delta W'_{jk} = -\eta\frac{\partial E}{\partial W'_{jk}} = -\eta\frac{\partial E}{\partial O_k}\frac{\partial O_k}{\partial T_k}\frac{\partial T_k}{\partial W'_{jk}} \tag{II.3}$$

となります。それぞれの値を計算していきます。まず式 II.2 より

$$\frac{\partial E}{\partial O_k} = -(A_k - O_k)$$

です。O_k 以外の変数がすべて消えるのは、偏微分なので O_k 以外の変数を全て定数とみなすためです。

次に、簡単な計算により、式 II.1 のシグモイド関数について

$$f'(x) = f(x)(1 - f(x))$$

が成り立つことが分かります。よって、

$$\frac{\partial O_k}{\partial T_k} = \frac{\partial f(T_k)}{\partial T_k} = f(T_k)(1 - f(T_k)) = O_k(1 - O_k)$$

となります。頂点の内部関数にシグモイド関数を用いたのは、このように微分して元の関数の式で表せるという性質を用いると計算が簡単になるからです。

最後に、

$$\frac{\partial T_k}{\partial W'_{jk}} = \frac{\partial(\sum_{j=1}^{N_{hid}} H_j W'_{jk})}{\partial W'_{jk}} = H_j$$

[12] ここでの「最小化」は厳密な意味での最小化というより、目的関数を小さくしていくという意味合いで使っています。

[13] $z = f(y), y = g(x)$ と表されている時に、$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$ が成り立つという法則

以上から、式 (3) より

$$\Delta W'_{jk} = -\eta(-(A_k - O_k))O_k(1-O_k)H_j = \eta(A_k - O_k)O_k(1-O_k)H_j = \eta H_j \delta_k$$

$(\delta_k = O_k(1-O_k)(A_k - O_k)$ とおいた)

と表せ、きれいな式になりました。

次は ΔW_{ij} を求めます。先ほどは W'_{jk} が出力 O_k にしか関係していなかったため、$\frac{\partial E}{\partial O_k}$ のみを考えればよかったのですが、今度はそうはいきません。そこで、合成関数に関する次の公式を用いることになります。

関数 $f(y_1, y_2, ..., y_n)$ について、$y_j = g_j(x_1, x_2, ..., x_m)$ と表されているとき、

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

が成り立つ。

証明は省略して認めることとします。E が $O_1, O_2, ..., O_{N_{out}}$ の関数であり、$O_k = f(T_k)$ と表されているので E は $T_1, T_2, ..., T_{N_{out}}$ の関数です。これと $T_k = \sum_{j=1}^{N_{hid}} H_j W'_{jk}$ より T_k が $H_1, H_2, ..., H_{N_{out}}$ の関数として表されていることに注意してこれを用いると

$$\begin{aligned}\Delta W_{ij} = -\eta\frac{\partial E}{\partial W_{ij}} &= -\eta\frac{\partial E}{\partial H_j}\frac{\partial H_j}{\partial W_ij} \\ &= -\eta(\sum_{k=1}^{N_{out}}\frac{\partial E}{\partial T_k}\frac{\partial T_k}{\partial H_j})\frac{\partial H_j}{\partial S_j}\frac{\partial S_j}{\partial W_{ij}} \\ &= -\eta(\sum_{k=1}^{N_{out}}\frac{\partial E}{\partial O_k}\frac{\partial O_k}{\partial T_k}\frac{\partial T_k}{\partial H_j})\frac{\partial H_j}{\partial S_j}\frac{\partial S_j}{\partial W_{ij}}\end{aligned}$$

となります。前二つの式はすでに計算済みで、その積は $-\delta_k$ であり、

$$\frac{\partial T_k}{\partial H_j} = \frac{\partial \sum_{j=1}^{N_{hid}} H_j W'_{jk}}{\partial H_j} = W'_{jk}$$

$$\frac{\partial H_j}{\partial S_j} = \frac{\partial f(S_j)}{\partial S_j} = H_j(1-H_j)$$

$$\frac{\partial S_j}{\partial W_{ij}} = \frac{\partial \sum_{i=1}^{N_{in}} I_i W_{ij}}{\partial W_{ij}} = I_i$$

です。以上から、

$$\Delta W_{ij} = \eta(\sum_{k=1}^{N_{out}} \delta_k W'_{jk}) H_j(1-H_j) I_i = \eta I_i H_j (1-H_j) \sum_{k=1}^{N_{out}} \delta_k W'_{jk}$$

これでようやく更新式が求められました。隠れ層が 2 つ以上になった際にも、2 つ目の更新式と同じ形の式で更新することができます。

5a　ニューラルネットワークまとめ

上で求めた更新式をもう一度書きます。

$$\delta_k = O_k(1-O_k)(A_k - O_k)$$

$$W'_{jk} \leftarrow W'_{jk} + \eta H_j \delta_k$$

$$W_{ij} \leftarrow W_{ij} + \eta I_i H_j (1-H_j) \sum_{k=1}^{N_{out}} W'_{jk} \delta_k$$

δ_k には、各カテゴリでのスコアの誤差と、シグモイド関数を微分した形が含まれています。シグモイド関数は入力の値が極端になるほど傾きの小さくなる関数です。スコアが極端 (ほとんど確定している) と修正分が小さくなり、そうでないとき (未確定) は修正分が大きくなる。そしてそれに誤差分がかかると考えれば、なんとなくこの式の直感的な理解もできるのではないでしょうか。

また、誤差を最小化したいのなら更新を止まるまで実行していかなくていいのか、と思う人がいるかもしれません。これは一見正しいようですが、このように一つのデータセットに対して過度に誤差を少なくしようとすることは、他のデータセットに対しての誤差を大きくすることにつながりかねません。一般に特定のデータに対してのみ良い精度の分類を行う分類器より、全てのデータに対しそこそこの結果を出す分類器の方が優れていると言えます。ニューラルネットワークの場合、全てのデータセットに対してそこそこの結果を出すためには各データセットに対し一回更新を実行するだけで十分です。このようなまんべんなくまともな結果を出すような分類器の性能のことを**汎化性能**と呼び、一つのデータセットに対して過度に学習して汎化性能が下がってしまうことを**過学習**と呼びます。

それでは、最後にニューラルネットワークを用いて実際に学習をさせてみます。下のソースコードにおいて、$feed_forward$ という関数が入力から各頂点の出力を計算する関数、$back_propagate$ という関数が誤差から辺の重みづけを変更する関数です。実際の訓練・評価の部分は K 近傍法のソースコードとほぼ同じなので省略します[14]。

[14] 訓練には $back_propagate$ を、評価には $feed_forward$ を用います。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from math import exp
from random import *

#シグモイド関数
def sigmoid(x):
    return 1 / (1 + exp(-x))

#シグモイド関数を微分した関数
def dif_sigmoid(y):
    return y * (1 - y)

def rand():
    # [-0.1,0.1) のランダムな数を返す
    return random() * 0.2 - 0.1

def randtable(n, m):
    #rand 関数を用いてランダムな (n,m) テーブルを作成

class Neuralnet:
    #初期化
    def __init__(self, n_input, n_hidden, n_output, rate):
        #入力層→隠れ層への辺の初期化
        self.in_edge = randtable(n_input, n_hidden)
        #隠れ層→出力層への辺の初期化
        self.out_edge = randtable(n_hidden, n_output)
        self.in_data = [0] * n_input # I_i
        self.hid_data = [0] * n_hidden # H_j
        self.out_data = [0] * n_output # O_k
        self.rate = rate #学習率η
        self.n_in = n_input #入力層の頂点数
        self.n_hid = n_hidden #隠れ層の頂点数
        self.n_out = n_output #出力層の頂点数

    #前方向計算
```

```
    def feed_forward(self, data):
        #I_i への入力
        for i in range(self.n_in):
            self.in_data[i] = data[i]
        #H_j の計算
        for j in range(self.n_hid):
            s = 0
            for i in range(self.n_in):
                s += self.in_data[i]
                        * self.in_edge[i][j]
            self.hid_data[j] = sigmoid(s)
        #O_k の計算
        for k in range(self.n_out):
            s = 0
            for j in range(self.n_hid):
                s += self.hid_data[j]
                        * self.out_edge[j][k]
            self.out_data[k] = sigmoid(s)
        return self.out_data

    #逆方向誤差伝搬
    def back_propagate(self, data, answer):
        #各頂点の出力計算
        ret = self.feed_forward(data)
        #data : I_i, answer : A_k, ret : O_k
        out_dif = [] #δ_k
        for k in range(self.n_out):
            out_dif.append((answer[k] - ret[k])
                               * dif_sigmoid(ret[k]))
        hid_dif = [] #ΔW_ij の H_j * (1 - H_j) * Σ の部分
        for j in range(self.n_hid):
            s = 0
            for k in range(self.n_out):
                s += self.out_edge[j][k]
                                * out_dif[k]
            hid_dif.append(dif_sigmoid(self.hid_data[j]) * s)
        #各辺の重さを修正
```

```
for i in range(self.n_in):
    for j in range(self.n_hid):
        self.in_edge[i][j]
            += self.rate * hid_dif[j] * data[i]
for j in range(self.n_hid):
    for k in range(self.n_out):
        self.out_edge[j][k]
            += self.rate * out_dif[k] * self.hid_data[j]
```

学習の結果は以下のようになりました。

データセット	正答サンプル数	全サンプル数	正答率
ひらがな (ETL4)	220.0	305	72.1%
カタカナ (ETL5)	508.3	530	96.0%

カタカナのデータセットでは高い正答率を実現していますが、ひらがなのデータセットの結果はいまいちです。これはデータセットの数が少ないことが関係していると思われます。一般にニューラルネットワークは大体の問題に応用できる代わりに、比較的収束が遅い (良い結果を得るために必要なデータセットの量が多い) ことが知られています。

第 III 章

今後の展望

第 II 章では、画像から加重方向指数ヒストグラムという方式を用いて特徴を抽出し、その情報と K 近傍法・ニューラルネットワークを用いて文字認識を行いました。では、特徴を抽出せず、画像データをそのまま[1]ニューラルネットワークに渡すとどうなるのでしょうか。画像を 10*10 ピクセルに圧縮し、200 個の隠れ頂点を持つニューラルネットワークに渡してみます。(カタカナデータセットのみ)

データセット	正答サンプル数	全サンプル数	正答率
カタカナ (ETL5)	384.3	530	72.5%

高い精度とは言えませんが、特徴抽出の手間なしでこの精度が実現できる、というのは注目に値します。実は、今回紹介したニューラルネットワークの階層を増やし、さらに改良を重ねることで、元のデータから自動で特徴抽出のようなことが行われ、かなり高い精度での識別が実現される、ということが分かってきています。

以前は今回紹介した通り、元のデータからある意味手動で特徴を抽出し、それを学習モデルに渡すことで機械学習を行ってきました。しかし、この方法では、特徴抽出の技術によって結果が大きく左右され、しかも今回見たように特徴抽出には分野ごとに異なる洗練されたアプローチが必要とされるため、機械学習には職人技とも呼ばれる特徴抽出技術が必要でした。

そのため、その特徴の抽出すらも機械に丸投げすることができる、というのは非常に画期的であり、Google や Facebook などの大企業をはじめ、世界中の企業・研究者がこの方法に注目しています。この深い階層のニューラルネットワークによる**ディープラーニング (Deep Learning)** という分野は、今後大きく発展していくことが期待されています。今回はこのような軽い紹介に留まってしまいましたが、非常に魅力的な分野であることは間違いないので、興味のある方は是非勉強してみることをお勧めします。

[1] 実際は各ピクセルで黒い画素を 1、白い画素を 0 とする特徴を入力として与えています

第 IV 章

おわりに

お疲れ様でした。今回は機械学習の大まかな流れ、K 近傍法、ニューラルネットワークなどについて紹介しましたが、クラスタリングなどの教師なし学習、自然言語処理、分類器であるベイズ分類器・決定木・SVM(サポートベクトルマシン)、そして第 III 章で紹介した Deep Learning など、今回扱えなかった話題はたくさんあるので、より詳しく知りたい方は参考文献を参照してみてください。Deep Learning に関しては中々まだ勉強しやすい環境は整っていないようですが、多くの研究グループが Deep Learning を行うためのソフトウェアをオープンソースにしているそう [10] なので、そちらを試してみるのも良いと思います。

また、この記事についてのご質問、ご指摘などありましたら、以下の連絡先までご連絡頂けると幸いです。私自身まだ機械学習の勉強を始めて間もないので、どんな内容でも参考にさせて頂きます。

最後になりましたが、私の稚拙な文章をここまで読んで下さり、ありがとうございました。もしこの記事によって機械学習に興味を持って頂け、この記事が皆さんの理解の一助となれたのならば、これ以上嬉しいことはありません。

連絡先
Gmail: ryunosuke.iwai@gmail.com
Twitter: @reyk429

参考文献

[1] Toby Segaran. (2008).『集合知プログラミング』(當山 仁健・鴨澤 眞夫 訳) オライリー・ジャパン

[2] 高村 大也. (2010). 『言語処理のための機械学習入門』(奥村 学 監修) コロナ社

[3] Bishop, C.M. (2012). 『パターン認識と機械学習』(元田 浩・栗田 多喜夫・樋口 知之・松本 裕治・村田 昇 監訳) 丸善出版

[4] WordPress. (2014). 『ETL 文字データベース』 http://etlcdb.db.aist.go.jp/?lang=ja

[5] Media Drive Corporation. http://mediadrive.jp/technology/techocr10.html

[6] 木村 拓也. (2011). 『複数人の類似筆記者の文字特徴を利用したひらがな文字認識』

[7] 村上, 泉田研究室. 『ニューラルネットワーク』 http://ipr20.cs.ehime-u.ac.jp/column/neural/index.html

[8] Akira Iwata, Toshiyuki Matubara. (1996).『ニューラルネットワーク入門』http://www-ailab.elcom.nitech.ac.jp/lecture/neuro/menu.html

[9] 金久保　正明. http://www.sist.ac.jp/∼kanakubo/index.html

[10] Hatena blog 『Deep Learning でラブライブ！キャラを識別する』 http://christina.hatenablog.com/entry/2015/01/23/212541

[11] Wikipedia 『機械学習』 http://ja.wikipedia.org/wiki/機械学習

第 III 部

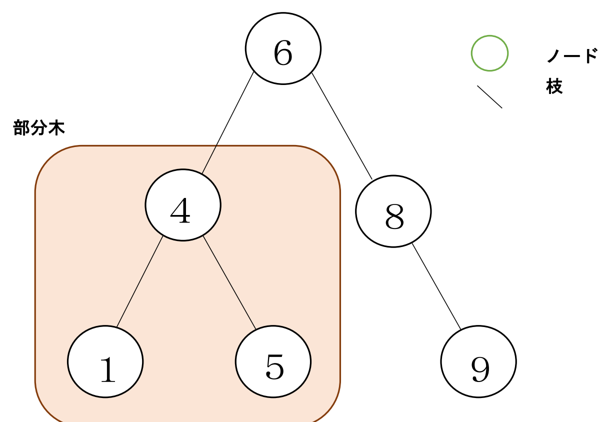
木にノードを加えて軽く炒めて食べたい

cloud

第 1 章

木とは

おはようございます。７１回生の cloud です。木についての駄文を書いていくので、どうぞお付き合い下さい。木とは配列などど同じ ” データ構造 ” の一つです。データ構造というのはデータ（値）を効率的に格納する構造です。例として、本を整理するときにどのような順番で並び替えるかを考えます。本のタイトルや作者の五十音順で並べたり、図書館と同じ分類法で並べたりなどいくつか考えられますが、誰もページ数で並べようとは思いません [要出典]。普通、人はページ数で本を選ばないからです。早くも話がそれましたが、この本の並べ方が構造、本がデータに当たります。そのうち、木とは下の図のような下向きに枝が分かれていく構造を指します。要するに樹形図です。

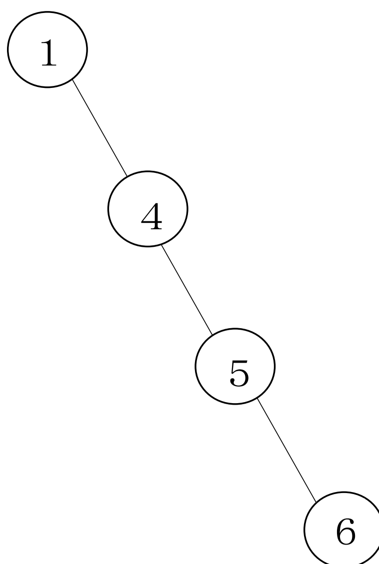


図の一つ一つの〇をノード（節）と言い、木の全てのノードは、0 個以上の子ノードを持ちます。その元のノードは親ノードと言います（上の図で１の親ノードは４）。親ノードを持たないノード（図では８）を根ノードと言い、１つの木につき１つあります。逆に子ノードのないノードをを葉ノード（図では１，５，９）と言い、ひとつの木にひとつ以上存在します。部分木は木の一部分で木の構造になっている部分を指す。根ノード以外を根とする部分木を真部分木と呼ばれる。高さは、あるノードからその子孫ノード（葉ノードのうちどれかになる）への枝の数の最大値（図の６の高さは３）で、深さは、逆にあるノードから根ノードまでの枝の数（図の８の深さは８）のことです。

第 II 章

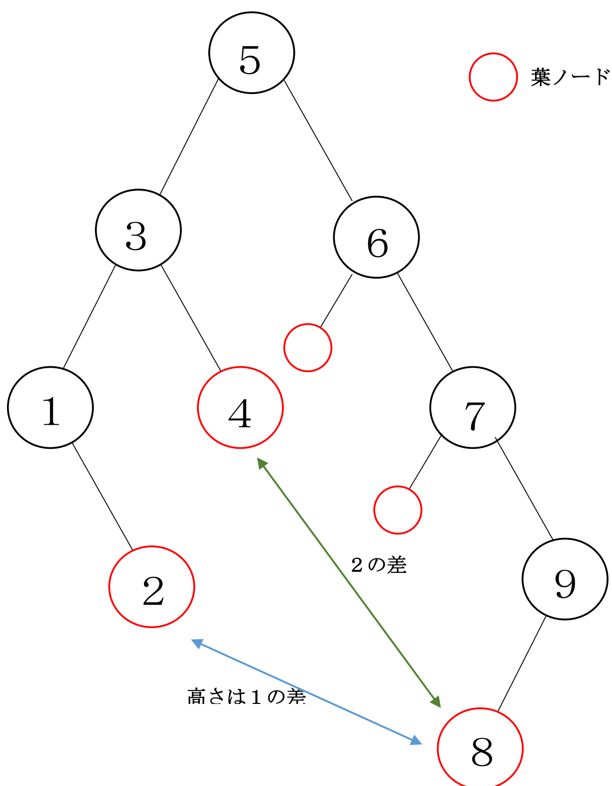
二分木

ノードの持つ子ノードの数が２以下の木です。ふつう、左の子ノード < 親ノード < 右の子ノードで、上の図も当てはまります。　この木の特徴として、探索をするときにその対象が親ノードとの大小から左右のどちら側にあるかを容易に調べることが挙げられます。しかし、下の図のような極端な例では、一つ一つを探索しているのとかかる最悪の時間が変わらず、木を使う意味がなくなります。



そこで、根ノードから葉ノードまでの枝の数を出来るだけ少なく、つまり各葉ノードまでの枝の数をほぼ一緒にすることを考えます。平衡２分木　二分木に対して、木の回転（後述）等をを行い木の高さを低くした木です。　これには何種類かあり、次のようなものが挙げられます。

AVL 木　AVL 木は、次の 2 つの条件を満たした上でノードを増やすときに回転を行うことでこの条件を常に満たすようにします。2 つの子ノードを持つノードにおいて、右の部分木と左の部分木の高さが 1 以内である 1 つの子ノードしか持たないノードの子ノードは葉ノードであること　必ず左右の部分木の高さの差が 1 以内でも、すべての葉の高さの差が 1 以内であるとは限りません（下の図）。



この木にノードを加えるとき、この木を ” 回転 ” させることで上の条件をみたすようにします。回転をノードを増やすたびに行うのでそれだけの手間はかかるものの探索をする時間は短縮されます。

第 III 章

回レ！

　例えば、下の図１の青の所に１つノードを追加するとすると、下の図２のように回転させます。

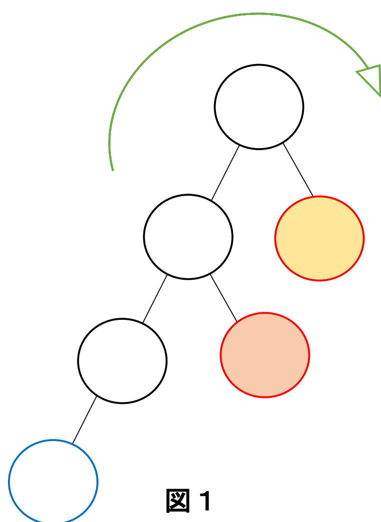


図１

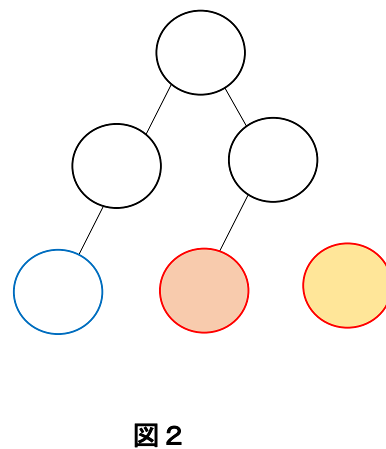
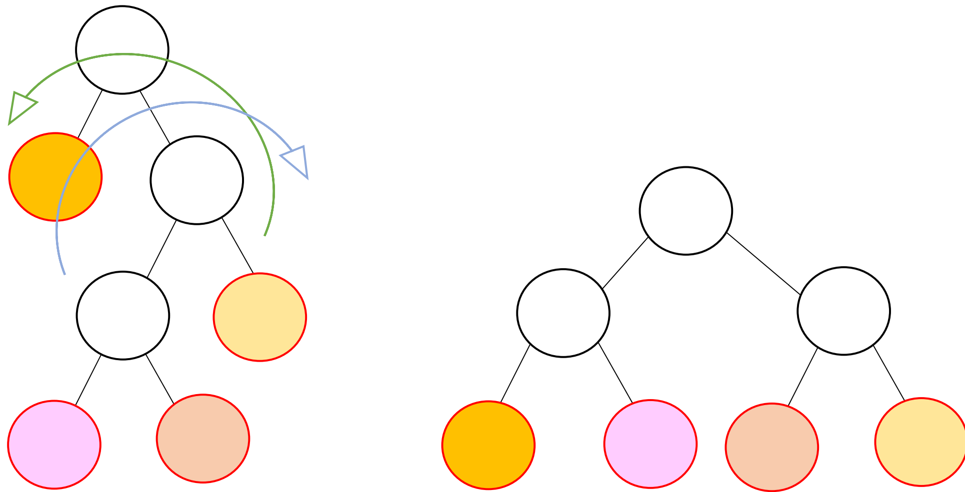


図２

　　木を回転させるときは、精々回転の中心のノードと周辺２〜３個のノードをいじるだけなので、簡単ですが下のように２回以上回転出来る場合もあります。



とはいえ、回転させる回数は木の高さ以下なので、値の追加並みに速く処理できます。ですが、常に高さを一定に保つ性質上、探索は容易なのに対し木を保つのに時間がかかる場合があります。この木はこれらのことから、ノードの追加、削除の殆ど無いデータに向いています。

第 IV 章

赤黒木

そこで、条件を緩くした赤黒木を使います。　赤黒木は全てのノードが赤か黒に塗られていて、以下の 3 つの条件を全て満たします。

1. 根ノードは黒
2. 赤のノードの子ノードは黒
3. 葉ノードにたどるときに通る黒の個数が、 どのノード、どの葉に対しても等しい

これらの条件により一番低い高さの葉ノードは一番高い葉ノードの高さの 1/2 となることが保証されます。　この木も、AVL 木と同様に回転によって平衡を保ちます。（一番

左の赤を追加）

　このような手順で、（各葉ノードに部分木が付いていても可）赤黒木の平衡を保つことができました。　これ以外の追加の仕方もありますが、省略します。

1　つまり

　O(N) を回避しましょう。それは線形リストです。
　長々とこんな文章をお読みいただき有難うございました。

第 IV 部

Web-othello についてのお話と小噺

KA-FU- & hinata

第1章

まえがき

72 回生 (当時中 1) の KA-FU-です。今回の文化祭ではゲーム展示の一つ、Web-othello を作りました。Web-othello の作成者は二人で、私はオセロ本体と AI [1] を担当しました。もう一人は、同級生の期待の Web マン [2] 、hinata です。Web マンなので、もちろん Web に関する様々なことを手伝ってもらいました。この部誌の内容は題名の通り、Web-othello 作成中のエピソードに関連したお話や、小噺のようなものです。また、この部誌はかなりの部分が小噺、つまりネタになっています。なので、まじめに書かれた部分だけ読みたい方は次のリンクから飛んで読んでください。

- 読みやすいコードについて
- jQuery について
- 非同期プログラミングについて
- コールバックについて
- Promise について
- Generator パターンについて
- SQL インジェクションについて
- あとがき

また、著者の部分に「KA-FU-」と「hinata」とありますが、一部のセクションのみを hinata が書き、それ以外は私、KA-FU-が書かせて頂くという形式になっております。ご了承ください。

[1] 人工知能
[2] Web 開発をする人

第 II 章

Web-othello 作成の流れ

1　オセロ本体

私は作成当初 C という言語 (通称 C 言語) を使ってプログラミングをしていました。NPCA では、最初に勉強するプログラミング言語として C 言語を多くの人にすすめています。 Wikipedia によると、

> 『UNIX [1] および C コンパイラ [2] の移植性を高めるために開発してきた経緯から、オペレーティングシステム [3] カーネル [4] およびコンパイラ向けの低レベル記述ができる』

だそうです。

とある一般人「オペレーティングシステムとかよくわかんないですけどそれ、簡単に Web で使えるんですか」

そうですよね、そんな気がしますよね。実際難しくて諦めました...。

ということで、JavaScript というウェブブラウザ [5] 上で動く唯一の言語 (通称 JS) を使うことにしました。

とあるロリコン「えっ JS...？女子小学生のことですか」

違います、日頃 JS を使ってる hinata が怒りだすのでやめてください。 Wikipedia によると

---

[1] 後述するオペレーティングシステムの種類の一つ
[2] プログラミング言語をコンピューターがわかるものに変換してくれるもの
C コンパイラは C 言語を変換する
[3] Windows や OS X 等のこと
[4] オペレーティングシステムの中でも中心的な機能の部分
[5] Chrome や Safari、Firefox、InternetExplorer 等のこと

> 『実行環境が主にウェブブラウザに実装され、動的なウェブサイト[6]構築や、リッチインターネットアプリケーションなど高度なユーザインタフェース[7]の開発に用いられる。』

まぁ、Web サイトで使うんですよ、はい。ここらへんは Web マンである hinata に説明してもらったほうがいいのですが、熱弁しだすので遠慮しておきましょう。

JS は C 言語に書き方が似ているので、意外とオセロ本体は簡単に完成しました。そう、オセロ本体は...。

2　AI

オセロ本体と一緒に AI も作ったのですが、考えるのに非常に時間がかかる上に、別にそこまで強くないというお粗末なものでしたので、改良することになりました。

新しい AI は MySQL というツールを使ってデータベースを操作する贅沢なものにしようと考えました。データベースとは、"データ"という単語が入っていることから想像できるように、使いやすく整理してあるたくさんのデータです。Wikipedia によると

> 『特定のテーマに沿ったデータを集めて管理し、容易に検索・抽出などの再利用をできるようにしたもの。』

だそうです。まぁ、データの集まりですよ、はい。このデータベースはサーバーにあって、ブラウザ上で動く JS を使って操作をするのは無理らしいので、仕方なく違う言語を使うことにしました。その言語は、PHP: Hypertext Preprocessor というもの (通称 PHP) です。

とある主婦「えっ DHC？よく健康食品を買わせてもらってるわ。」

聞き間違いも甚だしいです、化学研究部員でもある hinata がドコサヘキサなんとか[8]とか言い出すのでやめてください。 Wikipedia によると、

> 『PHP はサーバーサイド・スクリプト言語として利用されており、Web サーバ上で動作し、Web サーバ上で PHP スクリプトの文書が要求されるたびに、その PHP スクリプトが実行され、結果をウェブブラウザに対して送信する。』

まぁ、サーバーで動くんですよ、はい。で、できた AI がデータベースを存分に使うくせにランダムに打つのとあまり変わらないというもので、更に悪くなってしまいました。うひーつらい。ということで、また新しい AI を考える羽目になったのでした、めでたしめでたし。

---

[6] ページをもう一度読み込まなくても、そのまま表示が変化するウェブサイトなど
[7] そのシステムをユーザーが操作するのに必要なもの
[8] ドコサヘキサエン酸 (DHA)
DHC の健康食品の中ににこれを主成分とするものがある

3　めでたくない

実はこの頃に Lisp という言語を使うようになっていました。 Wikipedia によると

> 『LISP は現在でも広く使われている年代物の言語の一つである。』

だそうです。Lisp を使うようになってからというもの、Lisp 以外の言語を読むことが難しくなってきました。なので、PHP を使って新しい AI を作るのをやめ、PHP 内で Lisp を実行することで Lisp を使った AI を作成しました。~~しかし、やっぱりお粗末なものにしかならず、諦めました。~~ こ、今回は、まだマシなものが出来上がったので、い、一件落着です。

第 III 章

本題

1　Web、襲来

本題に入ってすぐに余談なのですが、Web-othello 作成のいきさつをば。僕は一学期の終わりから二学期の初めの間 (夏休みは何もしてませんでしたが) に、C 言語で CUI [1] のオセロを作りました。文化祭のゲーム展示ではこれを GUI [2] 化して使いまわす気まんまんでした。しかし、二学期に入ってから一ヶ月ほどたったときに hinata が入ってきて、なんと

hinata「プログラミングに興味を持ち、家で JS を勉強していたので少しなら書ける。」

というではありませんか。これは利用するしかないと Web 化を決意した次第であります。まさかあんな事になるとはこの時誰も思いませんでした...

2　動かない、ページ

僕の環境 (Chrome) でオセロ本体が正常に動いたので、他のウェブブラウザでもテストする必要があります。しかし、私は Mac ユーザーなので InternetExplorer(以後、IE) でのテストはちょっとめんどくさいのです。なので、その時 Windows ユーザーだった hinata に IE でのテストを頼みました。(hinata が Mac ユーザーになった今も頼んでいますが。) すると、

hinata「『遊ぶ』ボタンが反応しないです。あと、IE8 以前でレイアウトがおかしいです。それ以外は大丈夫です。」
KA-FU-「ファッ」

[1] キーボードで入力し、文字で出力するユーザインタフェース
[2] キーボードだけでなくマウスなどでも入力でき、また出力も文字だけでなくウィンドウや画像など様々なものを出力するユーザインタフェース

ひーつらいですねぇ。どうにかしなければ。

とある一般人「え、見出しまでつけておいて問題点それだけですか。」

いやー、それだけだったんですよねこれが。まぁ、問題点ですから少ないほうがいいんですよ、はい。で、困ったものです、私には IE のことなんてさっぱりですし一体どうしたものか。(ググればどうにかなるでしょというツッコミは受け付けておりません)

3　決戦、InternetExplorer

IE での問題点が発覚した後しばらくして、

KA-FU-「hinata、IE の件どうなった？」
hinata「『遊ぶ』ボタンの方は type 属性が button の input 要素ではなく、type 属性が button の button 要素を使うことで解決しました。」
KA-FU-「... はい？」
hinata「レイアウトの方は IE8 以前を対応しないことにして解決しました。」
KA-FU-「お、おう。ま、まったく、Web マンは最高だぜ!!(動揺)」

＿人人人人人人＿
＞ 突然の解決 ＜
 Y^Y^Y^Y^Y

決戦なんてなかった。まぁ、hinata 曰く、根本的な解決にはなってないそうなので実際決戦はなかったらしい。

4　過去の自分が造りしもの

オセロ本体が完成してからしばらくたって、私は hinata に言いました。

ソースコードを読む前に言っておくッ！おれは今ソースコードをほんのちょっぴりだが読んだ
い、いや. . . 読んだというよりはまったく理解を超えていたのだが. . . . . .
あ、ありのまま今起こった事を話すぜ！
「おれはソースコードを読んでいたと思ったら全くもってソースコードを理解できなかった」

な、何を言っているのかわからねーと思うが
おれも何が起こったのかわからなかった...
頭がどうにかなりそうだった... 頭が悪いだとか寝ぼけていただとか
そんなチャチなもんじゃあ断じてねえ
もっと恐ろしいものの片鱗を味わったぜ...

そう、オセロ本体を少し変えようと思って自分が書いたソースコードを読んでみたら、それが何をしてるのか全然わからないのです。これは困った。読めないソースコードを直すなんてほぼ無理です。

とある一般人「かなりやばそうじゃないですか。いったいどうやって解決したんですか。」

~~結局諦めました~~ わ、私の実力でどうにかしましたよ。

ということで、ここでは『リーダブルコード』 [1] を参考に、読みやすいコードについて少しばかり書かせていただきます。

読みやすいコードとは

まず、ここでの読みやすいコードは一体何を指すのかを決めておく必要があります。ここでは、読みやすいコードとは、可読性が高いコードという意味だけでなく、そのソースコードについて何も知らない人が読んでも比較的短い時間で理解できるコードを指します。

とあるプログラマ「なんで他人に分かる必要があるんだよ、このコード見るの俺だけだぜ？」

と思う人もいるかもしれません。しかし、「そのソースコードについて何も知らない人」というのは他人だけでなく、そのソースコードの中身を忘れてしまった未来の自分も含んでいます。そう考えると、後のほうでそのソースコードが原因のバグが見つかったりしたときにも、やはり読みやすいコードである方が良いのです。

名前

変数や関数の名前についてです。ではまず、私が読めなかった (過去の自分の) ソースコードコードで使われている変数や関数の名前を少し見てみましょう。

実際に使われている名前	その変数や関数の中身
pname	プレイヤーの名前
game	決着がついたかどうか

place	AI が駒を置く場所
black	黒の駒を描画して保存する
white	白の駒を描画して保存する

我ながらひどい...。これらの名前の悪い点には次のようなものがあります。

- 名前を見ても中身が何かわからない
- 変に省略している
- 区切られていないので読みにくい

では、“pname” について、悪い点上から変えていきましょう。

1. まず、名前を見たら中身が何かわかるようにします。中身は「プレイヤーの名前」なので、“playername” などが良いと思います。(正しい英語の文法にする必要はありません。)
2. 次に、「変に省略している」ですが、これは 1 で直っているのでよしとします。
3. 最後に、区切ります。この変数が使われているのは JS のソースコードの中なので、“playerName” と区切る場所の直後の文字を大文字にするのがいいかと思います

このように、“pname” という意味不明な名前から、“playerName” というわかり易い名前になりました。(わかりやす差には個人差があります。) また、3 では言語によってよく使われる区切り方が違います。JS では区切る場所の直後を大文字に、Lisp なら区切る場所に “-”、Python [3] なら区切る場所に “_”、という感じだと思います。

では、次に「ホームディレクトリ配下のファイルとサブディレクトリのリスト」が入った変数に名前をつけてみます。そのまま名前に情報を詰め込むと、“list-of-files-and-subdirectories-within-home-directory” のようになるかと思います。(先ほど言ったように、正しい英語の文法にする必要はありません。) これでいいという人もいるかもしれませんが、私は長すぎると思います。このように、ただ情報を詰め込むだけでもダメで、その変数や関数の説明を正確に、そして短く書く必要があります。

では、次のようなコードがあったとします。

```
(defparameter right 1)
(defparameter left 2)
(defparameter tmp right)

(setf right left)
```

[3] 広く使用されているプログラミング言語で、さまざまなものをプログラムできる、非常に便利な言語。**しかし Lisp には劣る。**

```
(setf left tmp)
```

“right” と “left” の中身を入れ替えています。“right” と “left” に入ってる値が何なのかがわかりませんが、今見るべきはそこではありません。“tmp” です。“tmp” というのは “temporary” の略で、“temporary” というのは「一時的」という意味です。では、この “tmp” という名前の変数にはどういう意味があるのでしょう。実は、正解は「何の意味もない」です。“temporary” の意味のとおり、ただ一時的に値を保存しておくための変数で、それ自体に何の意味もありません。こういう時に、“tmp” のような名前をつかいます。変数の一生が短くても、本当に何の意味もない場合しか “tmp” は使ってはいけません。

他にもいろいろ気をつけれる点はあるのですが、~~めんどくさいですし、~~ここまでにしておきます。名前を考えるのも簡単じゃないのです。

コメント

ソースコードに書くコメントについてです。コメントは読み手にコードを読んだだけではわからない書き手の考えを伝えるためのものです。だからもちろん、コメントに書いてはいけないのは、コードを読めばすぐわかることです。例えば

```
;; hoge は要素数 10 の配列
(defparameter hoge (make-array 10))
```

こんなコードがあったとしましょう。CommonLisp が少しでもわかる人だったらきっと全員コメントに対して

とある CommonLisp が少しでもわかる人「んなこたわかっとるわ！！！」

と叫びたくなるのではないでしょうか。(叫びたくなり度には個人差があります。) このコメントはコードを見ればすぐにわかるので全くの無意味です。こういうコメントはしてはいけません。また、名前で補える情報はコメントにせずに名前で補うべきです。では、どのようなコメントをすればいいのかを、具体例を 2 つ挙げて説明します。

- ある機能を実装しようとしていたけど、その日のうちに出来上がらず、途中で一度寝ることにしたとします。もしその人が忘れっぽい人なら次の日の朝には何を実装しようとしていたのかわすれているかもしれません。こういうときは、次にやろうとしていることをコメントに残しておくのが良いです。また、なにかバグを発見したが今すぐにデバッグできないようなときにも、コメントでどういうバグが起きているのかを残しておくことは有効です。

- あるコードで主人公のお小遣いの値を入れる定数を定義したとします。名前からこれにお小遣いの値が入っていたことが分かったとしても、なぜこの値なのかはわかりません。そういう時にはコメントを使いましょう。例えば、「これくらいが年相応です」などなど。

他にも、いろいろなところで、コメントは使えます。でも、どの場合においても、読み手の事を考えて書くことが重要になってきます。なので、コメントも名前と同じで簡潔で、明確、そして正確である必要があります。コメントを書くときにもちゃんと考える必要があるのです。

他にも、読みやすいコードにするために工夫できるところはたくさんあるのですが、私はまだ未熟者なので全てを説明することはできません。なので、まずはこの 2 つに気をつけてみるといいと思います。たぶん、この 2 つに気をつけるだけでも十分読みやすいコードになると思います。他の工夫の仕方を知りたい人は、『リーダブルコード』を読んでください。

5　jQuery、来訪

その後、AI をサーバーで動かすにあたって Ajax という技術を導入しました。データを取得するのにフォームから送信してページを丸ごと読み込むのではなく、必要なデータだけバックグラウンドで読み込み、取得したデータを利用して処理をする、というのが主な流れです。しかし、この Ajax を自前で実装するのは結構面倒です。例えば hoge.php に user が hoge、text がぴょんぴょんとなるデータを送信する例です。

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'hoge.php', true);
xhr.onreadystatechange = function() {
    if (this.readyState !== 4) {
        return;
    }
    console.log('complete!');
    if (this.status === 200) {
        console.log('success');
        console.log(this.responseText);
    } else {
        console.log('error');
    }
};
```

```
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
xhr.send('user=hoge&text=%E3%81%B4%E3%82%87%E3%82%93%E3%81%B4%E3%82%87%E3%82%93');
```

xhr.send のところがわかりにくい (これだけ見てもどんなデータを送ってるのかわかりづらい) のでこうしてもいいんじゃないですかね。(間違ってても責任は取りません。)

```
encodeForm = function(data) {
    var param;
    var paramArray = [];
    for (param in data) {
        part = encodeURIComponent(param) + '=' + encodeURIComponent(data[param]);
        paramArray.push(part);
    }
    return paramArray.join('&').replace(/%20/g, '+');
};

xhr.send(encodeForm({
    user: 'hoge',
    text: 'ぴょんぴょん'
}));
```

どっちにしろ長いですね、はい。とにかく記述が長くなることだけ分かってもらえるといいです。というか書いてることが理解できる人がこの部誌を読むと間違いをたくさん指摘されそう。

ここで jQuery の出番です。jQuery を使っただけで

```
$.post('hoge.php', {
    user: 'hoge',
    text: 'ぴょんぴょん'
}).always(function() {
    console.log('complete');
}).done(function(data) {
    console.log('success');
    console.log(data);
}).fail(function(error) {
    console.log('error');
```

```
});
```

こうなります。(console.log というのはデバッグの時に使う関数で、コンソールに何かを出力します。) 短くなったのはもちろん、always や done、fail といったキーワードが出てきたので「hoge.php に post でデータを渡し、終わったらいつも (成功でも失敗でも)'complete' と出力し、成功したら'complete' と出力したあとに結果を出力する。失敗したら'error' と出力する」という流れが分かりやすくなっています。流れが分かりやすくなるという点は次の章でもう一度触れます (めんどくさくなって部誌を書くのをやめたりしない限り)

jQuery は HTML の要素を操作するのにも長けています。画像や表示されている文字が変わったり、一部が非表示になったりするページはよく見かけますが、そのような処理は jQuery を使うことで簡潔に分かりやすく書けます。

```
$('.hoge').removeClass('hoge');
```

これは hoge というクラスが付いた要素のクラスから hoge を取り除くコードです。生の JS で書くととても長くなります。長くなりすぎるし、書いたコードが正しいか自分でも分からなくなるので ~~というよりめんどくさいので~~ 例は省略します。

~~面倒になってきたし、~~ コードとかでちょっと長くなったのでこれくらいにします。

6　瞬間、処理、重ねて

非同期プログラミング

JS においてはネットワークアクセスなどが非同期で動作します。さっきの Ajax もそうでした。非同期の利点は

- 重い処理を止まって待つのではなくほかの作業をしながら待てる
- いくつかの重い処理を並列にできる

などです。(多分)Ajax はユーザーの操作を妨げずにバックグラウンドで通信できるためユーザーが待たされない点が好ましいとされています。サイトの一部を先に読み込むと読み込みが速くなったように感じるのではないでしょうか。

一方、非同期プログラミングは複雑になりがちです。結果が返ってくるのがいつか分からず、複数の非同期プロセスを同時に進行させるとどのプロセスが先に終了するかわからない。さらに、エラーをキャッチするのも難しい。また、後で見ていきますが、非同期プロセスを順番に実行しようとすると関数のネストが深くなる傾向になります。

例えば、setTimeout は非同期に動作するタイマー関数です。setTimeout(func, time) の形で time(単位ミリ秒) 後に func を実行します。ここで大事なのは、タイマーをセットするだ

けで待ちはしないことです。

```
setTimeout(function() {
    console.log('hoge');
}, 100);
console.log('huga');
```

このとき、hoge と fuga はどちらが先に出力されるでしょうか？ setTimeout はタイマーをセットするだけで、そのまま次に進んでしまうので fuga が先に出力され、100 ミリ秒後に hoge と出力されます。

KA-FU-「えっ sleep みたいにその場で待ってくれないんすか。えっえっ意味分かんない。」

というように生粋の JS マンである (Ruby や PHP や C++ もちょっとだけ書けますが) 僕には理解できない言葉を KA-FU-は言いました。setTimeout は非同期なのがあたりまえです、はい。しかし、1 秒後に hoge と出力し、その 1 秒後に fuga と、その 1 秒後に piyo と出力するプログラムを書いてみると

```
setTimeout(function() {
    console.log('hoge');
    setTimeout(function() {
        console.log('fuga');
        setTimeout(function() {
            console.log('piyo');
        }, 1000);
    }, 1000);
}, 1000);
```

一気にネストが増えて見づらくなりました。なお、KA-FU-はネストが増えても別に気にしない人なのでこのコードをみても

KA-FU-「えっ別になんとも思わないしまだ見やすい方なのでは。」

というように生粋の JS マンである僕には (ry)。これを世の人々は「コールバック地獄」(callback hell) と呼んでいます。(コールバックについては後ほど)Ruby の end 地獄と同じくらい読みにくい状態だと思います。これを解消するためにいろいろな偉い人達が頑張っ

てきました。

コールバック

onload や addEventListener などのメソッドででコールバックを設定するタイプと、関数の第二引数で関数を指定するタイプの二種類に別れます。XMLHttpRequest は前者です。

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'url');
xhr.onload = function() {
    if (this.status !== 200) {
        console.log(this.statusText);
    } else {
        var result = this.responseText;
        //result を使っていろいろする
    }
};
xhr.onerror = function() {
    console.log('error');
};
xhr.send();
```

setTimeout は後者です。後者は Node.js の標準ライブラリでよくある形ですね。

```
var fs = require('fs');
fs.readFile('path/to/file', { encoding: 'utf-8' }, function(error, data) {
    if (error) {
        console.log(error);
    } else {
        console.log(data);
    }
});
```

ファイルを読んでエラーが起きなければファイルの内容を、エラーが起きればエラーの内容を表示するだけのサンプルです。fs.readFile はオプションがある場合は第三引数、オプションがない場合は第二引数に関数を取ります。動作が完了した後に呼ばれる関数をコールバックと呼んでいます。どちらにしろ、さっきのコールバック地獄は避けられません。

onload や addEventListener などの引数である関数の中で二回目の非同期プロセスを呼ぶ必要があります。また、複数の非同期プロセスを並列に実行し、全てが終わったら別のことをする、なんて複雑なことをしたらプログラマが死にます。

例えば、ディレクトリのファイルを調べ、それらのファイルを 1 つずつ読んですべて終わったら結果を配列にして出力する、という例を考えてみます。(Node.js 前提です) 正直、こんな処理書きたくないです。fs(File System) というライブラリに readdir という関数がありますが、この関数はディレクトリも含めて返してきます。そこで 1 つずつファイルかどうか調べ (fs.stat という関数で調べられます)、ファイルのものだけ読む (fs.readFile)。全部読み終わったら結果を配列にして出力する。これらをすべてコールバックでやるなんて無茶です。正直やりたくないです。そこで、悩める子羊のために新しい JS(正確に言うと標準の ES6) で Promise というものが正式に導入されました (導入されます)。

Promise

Promise に関する説明は他の本やサイトにゆずります。(そこまで詳しく理解してるわけではないですし) ここでは XMLHttpRequest を Promise を返す関数にしたいと思います。

```
function get(url) {
    return new Promise(function(resolve, reject) {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', url);
        xhr.onload = function() {
            if (this.status !== 200) {
                reject();
            } else {
                resolve(this.responseText);
            }
        };
        xhr.onerror = function() {
            reject();
        };
    });
}
```

使用法

```
get('/hoge/fuga').then(function(data) {
    console.log(data);
}).catch(function() {
    console.log('error');
});
```

とにかく、

```
hoge().then(/* 成功した場合*/).catch(/* 失敗した場合 */);
```

という形です、はい。(then の引数に成功した場合、失敗した場合と 2 つ関数を渡すこともできます)resolve, reject の引数となった値は then や catch の引数の関数に渡されます。resolve(this.responseText) としているので then の引数の関数で data として参照できます。resolve した値が次の関数に渡されています。

ところで、前出の jQuery を使った Ajax のコードを引っ張ってきます。

```
$.post('hoge.php', {
    user: 'hoge',
    text: 'ぴょんぴょん'
}).always(function() {
    console.log('complete');
}).done(function(data) {
    console.log('success');
    console.log(data);
}).fail(function(error) {
    console.log('error');
});
```

似てますね。jQuery でも Promise に似た仕組みがあり、Deferred と呼ばれています。

例:setTimeout を Promise や Deferred を使ってわかりやすくする。

```
function waitByPromise(time) {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve();
        }, time);
```

```
    });
}

function waitByDeferred(time) {
    var deferred = jQuery.Deferred();
    setTimeout(function() {
        deferred.resolve();
    }, time);
    return deferred.promise();
}
```

Deferred については説明が面倒なので省略します。

また、並列実行が簡単になる Promise.all という関数があります。Promise の配列を引数に取り、すべての Promise が resolve した時に resolve する Promise を返します。ちなみに一つでも reject されると reject し、resolve する値はそれぞれの Promise が resolve した値の配列となります。

```
Promise.all([waitByPromise(2000), waitByPromise(7000)]).then(function() {
    console.log('ok');
});
```

この場合、2 秒後に resolve される Promise と 7 秒後に resolve される Promise の配列が Promise.all に渡されるので、7 秒後、全てが resolve された後に ok と出力されます。また、then の関数の中で値を返したときは次の then の引数にその値が渡されますが、その返り値が Promise オブジェクトだった場合はその Promise オブジェクトが resolve するまで待たされます

さて、Promise を使って

> 例えば、ディレクトリのファイルを調べ、それらのファイルを 1 つずつ読んですべて終わったら結果を配列にして出力する

という例を考えてみます。天下の Promise さんならできるはずです。Node の fs には同期の関数もありますがあえて使いません。(非同期縛りです) まず、readdir や readfile を Promise を返す関数にします。また、ファイルかどうか判定する関数も追加します。

```
function readdir(path) {
    return new Promise(function(resolve, reject) {
```

```
        fs.readdir(path, function(error, files) {
            if (error) {
                reject(error);
            } else {
                resolve(files);
            }
        });
    });
}

function readFile(path, options) {
    return new Promise(function(resolve, reject) {
        fs.readFile(path, options, function(error, data) {
            if (error) {
                reject(error);
            } else {
                resolve(data);
            }
        });
    });
}

function isFile(path) {
    return new Promise(function(resolve, reject) {
        fs.stat(path, function(error, stats) {
            if (error) {
                reject(error);
            } else {
                resolve(stats.isFile());
            }
        });
    });
}
```

これらは単純に失敗したらエラーを reject、成功したら結果を resolve しています。これらを組み合わせて実装します。

次に、ファイルやディレクトリのの絶対パスの配列を渡すとファイルのパスだけ集めてくれる関数を追加します。

```
function collectFilePath (filepaths) {
    return Promise.all(filepaths.map(isFile)).then(function(result) {
        var ret = [];
        result.forEach(function(fileResult, index) {
            if (fileResult) {
                ret.push(filepaths[index]);
            }
        });
        return ret;
    });
}
```

これらを組み合わせて実装します。

```
readdir(root).then(function(files) {
    return files.map(function(filename) {
        return path.join(root, filename);
    });
}).then(function(filepaths) {
    return collectFilePath(filepaths);
}).then(function(filepaths) {
    return Promise.all(filepaths.map(function(filepath) {
        return readFile(filepath, {
            encoding: 'UTF-8'
        });
    }));
}).then(function(fileData) {
    console.log(fileData);
});
```

return がやけに多い黒魔術みたいなコードになりましたが、コールバックでやるよりはよほどマシでしょう。

(参考にさせていただいたサイト http://www.html5rocks.com/ja/tutorials/es6/promises/)

Generator パターン

ES6(新しい JS などの標準) で Generator というものが追加されました。途中で処理を止められるというのです。そこで、偉い人は考えました。

とある偉い人「非同期のコードを実行してる間は処理止めちゃえばいいじゃん」

実際はこんなに簡単では無いのですが基本的なアイデアはこんな感じです (たぶん)。yield というキーワードで処理を止めることができます。co というライブラリがいろいろ裏でやってくれるので楽できます。

co を使った例です。

```
function readFiles(files, options) {
    return Promise.all(filePaths.map(function(filepath) {
        return readFile(filepath, options);
    }));
}

co(function*() {
    var fileOrDirectory = yield readdir(root);
    var fileOrDirectoryPaths = files.map(function(filename) {
        return path.join(root, filename);
    });
    var filePaths = yield collectFilePath(fileOrDirectoryPaths);
    var fileData = yield readFiles(filePaths);
    console.log(fileData);
});
```

なんということでしょう、return が多用され黒魔術みたいだったコードが、非同期はそのままに同期処理の書きやすさを取り入れたコードになったではありませんか！

ところで、これがブラウザで使えるようになるのは何年後なのでしょうか。10 年後には古いブラウザを気にせず使えるといいですね。でも Microsoft の新しいブラウザが Generator を導入するのが 5 年後くらいかな？でも最近 Microsoft 頑張ってるしもっと速く導入してくれると嬉しいです。むしろ他のブラウザのほうが Microsoft の新しいブラウザ

よりも導入が遅いかもしれません。

ここまで書いた感想

　これ、なんの部誌だっけ...？たしかオセロの部誌だった気がするなぁ (遠い目)。半分くらいが Node.js での話になってるんじゃないですかね、どうしてこうなった。まあ、Promise の話はブラウザでもそれ以外でも同じですし、も、問題無いですね (震え)。なんでこんなに非同期について長々と書いてんだよ jQuery の時よりもはるかに長いよ。しかもこないだ勉強したばっかのことだし正確であることを保証できないし信頼性ゼロだよ。
　というわけで (なにが"というわけで"だよ全くまとめてねえよ) みなさんはもっと真っ当なサイトや本を読んで勉強しましょう。参考程度に留めていただけると幸いです。

とある一般人「なぜか『jQuery、来訪』と『瞬間、処理、重ねて』だけ気合入った文章じゃないですか？」

そこに気づくとは... やはり NPCA の部誌を読む人は天才か。あ、えっと、それはですね、書いてる人が違うからです。(まぁ編集したのは私 (KA-FU-) ですが。) さて、書いてるのは誰でしょう？というか待って、よく見たらほんとにすごい内容じゃないですか。これは私の立場が危うい...。
(ヒント：この部誌のタイトルの下には 2 つ名前が書いてあります。)

7　SQL インジェクション、そして

　オセロはちゃんと遊べるようになりました。やったぜ。しかし、hinata が言うにはセキュリティ的にダメなところがあるらしい。えっまじで、セキュリティとか考えたことなかったよ、Web 怖っ。(※セキュリティ問題は気にするのが普通です。)

hinata「SQL インジェクションによる攻撃に対策しないといけません。」
KA-FU-「は、はい...？」
hinata「prepare という関数を使えば対策できます。」
KA-FU-「へー。って、え、ごめん話の流れつかめてない。」
hinata「やるだけなので頑張ってください。」
KA-FU-「はい！？」

IE 対応はやってくれたのに...。はっこれが時に優しく時に厳しくというやつですねわかります。まぁ、調べたらわかりましたよ、はい。データベースを操作するときには、

```
select * from hoge.huga where foo = 1
```

のような SQL 文というものを実行します。しかし、素直にこれを毎回そのまま書くとプログラミング言語の性質を利用されたりなんやかんやで色々と怖いそうです。なので、prepare という関数などを使って

```
select * from hoge.huga where foo = ?
```

としておいて、実行するときに?の部分にさっきの例でいえば 1 などにして実行するという方法をとればいいそうです。あー Web 怖っ。では、次に行きましょうってぐわっ hinata 何をする、やめろ、くぁ w せ drftgy ふじこ lp

KA-FU-がログアウトしました。

ちょっと KA-FU-の説明が適当すぎるので僕 (hinata) が少し補足します。

```
select * from users where id ='$id' and password = '$password'
```

このような文字列を直接実行していたとします。このとき、$id = “”, $password = “’ or ‘a’=‘a” だとすると、文字列は

```
select * from users where id ='' and password = '' or 'a'='a'
```

となります。‘a’ = ‘a’ が常に真なのですべてのデータが合致し、合致するデータがあるためパスワード認証をスルーすることができます。また、複文を利用した SQL インジェクションなど、いろいろな攻撃方法があります。(徳丸本 [2] を参考) 専門の本を読んだほうがわかりやすいと思うのでそちらを参照してください。

なお、他の脆弱性に関して、今回は信頼出来ないデータを元に表示するような部分がなかったため XSS は理論上発生し得ないはず... です。

8　せめて、ログらしく

KA-FU-がログインしました。

はぁ、はぁ、hinata 怖かった...。SQL インジェクションをどうにかしたのですが、実はもうひとつ対策しなきゃいけない大きなセキュリティ問題があります。いやぁ...Web、恐ろしい...。オセロ本体は JS で書かれています、最初のほうで説明したとおりこれはブラウザ上、つまりオセロで遊ぶ人のパソコンやスマホで動くので、JS の部分は自由にいじれちゃ

うわけです。すると、JS がわかる人ならいわゆるチート[4]ができてしまうのです。きゃー怖い。しかも今のままではチートを使われてもそのままデータベースに保存しちゃいます。これだと AI がバグを起こしてエラーが大量発生！？なんてことにもなりかねません。なので、PHP でゲームのログ[5]が普通にプレイされたものかチェックすることにしました。データベースに保存するために JS から PHP にログが送られます。その内容は、どちらかがコマを置いた場所とその後の盤面の情報です。それを元にして PHP 側でログチェックをします。例えば、あるときコマを置いた場所を A、その後の盤面を B、その後コマを置いた場所を C、その後の盤面を D とします。このとき、ログチェックは B の C にコマを置いた時、D と盤面がおなじになるか、という方法で行っています。もちろんオセロのルールで置けないはずの場所においていてもダメです。これで問題点は全部クリアしました。やったぜ、成し遂げたぜ。

[4] いわゆる反則技のこと
[5] 記録のこと

第 IV 章

まとめ

1　世界の中心で Lisp を叫んだもの

まえがきにも書きましたが、Web-othello 作成中のエピソードに関連させてあれこれ書いただけの部誌で、一貫したテーマがなくまとめるようなことは特にありません。なので、私が Web-othello の開発を通して感じたことをここでは書かせていただきます。

第 III 章の 1 にも書いたように、簡単な気持ちでオセロの Web 化を決めました。しかし、実際には Web というのはそんな簡単なものではなく、付け焼き刃の知識では見た感じいいものになるだけでちゃんとしたものは作れませんでした。ちゃっかり Web マンである hinata に Web 方面の仕事を任せたのが幸いして、無事完成しました。餅は餅屋といいますが、まさにそのとおりだと実感しました。

では、この部誌のまとめは私が Web-othello 作成を通じて一番感じたことにさせていただきます。

まとめ
Lisp が一番使いやすい！

第V章

あとがき

最後まで読んでいただいた方、そして一緒にこの部誌を書いてくれたhinata、本当にありがとうございました。よく考えてみたら、私はネタ成分しか書いていないような気がします。hinataからは「ネタ部分がいないと部誌は成り立たない」と言ってもらえたのですが、さすがにちょっとという気が否めません。きっとこの部誌は私の黒歴史の一部となるでしょう。でももし、私の部誌をお楽しみいただけたのなら嬉しい限りです。おそらく来年も部誌を書かせていただくことになると思いますので、もしよければまた部誌を手にとっていただけると嬉しいです。そして最後に、まとめに心を打たれた方、「は？何いってんのこいつ」と思った方、その他の方はぜひLispを勉強しましょう。

では、ありがとうございました。

参考文献

[1] Dustin Boswell・Trevor Foucher リーダブルコードーより良いコードを書くためのシンプルで実践的なテクニック 須藤 功平 解説、角 征典 翻訳、オライリージャパン
[2] 徳丸 浩 体系的に学ぶ 安全な Web アプリケーションの作り方ー脆弱性が生まれる原理と対策の実践 ソフトバンククリエイティブ

第 V 部

木に対する一般的なテク達

@okuraofvegetabl

第 1 章

はじめに

こんにちは、69 回生の okuraofvegetable です。競技プログラミングが大好きで、普段は競プロの問題ばかり解いています。npca の部誌は昨年に引き続き 2 回目です。昨年の部誌では、グラフの基本的な知識や問題について書きました。私はグラフ理論の問題が好きです。見かけはグラフに見えなくてもグラフの問題に落としこめる問題などもとても好きです。今回はグラフの中でも、更に的を絞って"木"に対する問題のテクニックなどを紹介したいと思います。中でも、木に対するたくさんのクエリに高速に答える問題において威力を発揮するであろうテクニックを紹介します。木の根や形が変わるような動的木については扱いません。そちらについて詳しく知りたい方は 2 年前の部誌の catupper さんの記事に詳しく載っていますのでそちらを参照してください。昨年の部誌と同様に、文章中のソースコードではすべて C++ を用いています。また、ソース中の整数型でほとんど int 型を用いていますが、実際に問題を解く際は制約に注意して適宜 long long int 型などを用いてください。ソースコードをコピペして使って頂いても構いませんが、何があっても自己責任でお願いいたします。

第 II 章

知識編

1　木の用語

グラフの基本的な用語や知識 (辺、頂点などの用語、DFS などのアルゴリズム) については割愛させていただきます。もしわからない場合は調べるか、去年の部誌の私の記事を参照してください。

木

木というのは下の図のような特殊なグラフのことで N 個の頂点と $N-1$ 本の辺がある連結なグラフのことです。実はこの条件だけで、木には閉路が含まれないことがわかります。背理法で証明してみましょう。$M(0 < M < N)$ 頂点の閉路があると仮定します。閉路の中には M 本の辺が含まれています。残り $N-M$ 頂点と閉路を連結にするためには最低 $N-M$ 本の辺が必要です。閉路に含まれる辺とあわせて最低 N 本の辺が必要です。これは辺が $N-1$ 本であることに矛盾します。よって木の中に閉路は存在しません。木においては任意の頂点から他の頂点までの行き方 (パス) はただひと通りに定まります (もちろん往復など無駄な動きをしない場合)。図を見ればなぜこれが木と呼ばれるのかわかると思います。この記事のソースコード中で、木は普通の無向グラフと同様に std::vector を用いた隣接リストで表現します。

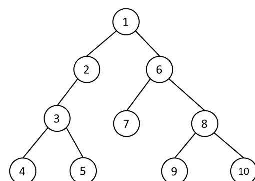


図 II.1　木

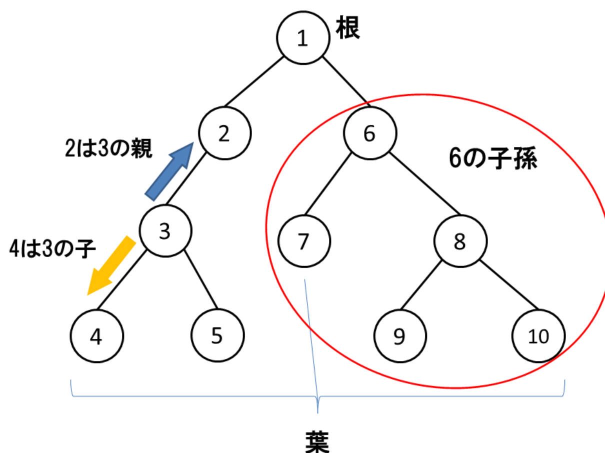


図 II.2　根、葉、子孫、先祖

根、葉

ある頂点を図の一番上に固定して考えると色々と便利なことがあります。その際、一番上に固定した頂点を根と呼びます。また、根を固定した木を根付き木と呼びます。また、子のいない頂点を葉と呼びます。

深さ

頂点 v の深さは、根から v までの辺の本数です。木の深さは全ての頂点の深さの最大値です。

子、親、子孫、先祖

根付き木において、根以外のある頂点 v に直接繋がっている頂点のうち根に近い頂点を v の親、それ以外を v の子と呼びます。根以外の任意の頂点はただ一つの親を持ちます。そして、頂点 v の子、子の子、子の子の子、…をすべてあわせて v の子孫と呼びます。[1]。同様に、頂点 v の親、親の親、親の親の親、…をすべてあわせて v の先祖と呼びます 各頂点の親や深さは dfs することで $O(N)$ で求まります。以下のソースでは、木を辺の集合として入力し DFS で頂点 0 を根とした時の各頂点の深さと親の頂点番号を求めています。[2]

---

[1] 図では v 自身も v の子孫に含んでいます。以後問題文中などでも v の子孫といった場合、v 自身も含める事にします

[2] 頂点番号は 0〜$N-1$ と仮定します。

```
#include <cstdio>
#include <vector>
using namespace std;
#define MAX_N 100100
#define pb push_back
vector<int> g[MAX_N];//tree
int N,root=0,parent[MAX_N],depth[MAX_N];
void dfs(int v,int p,int d)
//v...current vertex , p...parent of v ,d...depth of v
{
    depth[v]=d;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]==p)continue;
        dfs(g[v][i],v,d+1);
    }
}
void add_edge(int u,int v)
{
    g[u].pb(v);
    g[v].pb(u);
}
int main()
{
    scanf("%d",&N);//number of vertexes
    for(int i=0;i<N-1;i++)// edges
    {
        int u,v;
        scanf("%d %d",&u,&v);
        add_edge(u,v);
    }
    dfs(root,-1,0);
}
```

2 木の問題について

そもそも木に対するクエリに高速に答える問題というのはどのようなものなのでしょうか?

クエリとは

クエリというのは言わば質問のようなもので、「〜な頂点は何個ありますか?」や「〜と…の間の距離はいくらですか?」のようなものが多いです。

問題の難しさ

実際に競技プログラミングのコンテストで出題されるような問題では、木の頂点数が 10^5 個、クエリの数が 10^5 個程度のものが多いです。あらかじめ、任意のクエリに対する解を前計算し、クエリを読み込むたびに $O(1)$ で答えられる問題もありますが、そうでない場合 Time Limit が 1,2 秒であることを考慮すると、1 クエリあたり 100〜500 回の計算で答えを出さなくてはなりません。頂点数に対して圧倒的に小さい回数の計算で答えを出すためには様々な前計算やテクニックが必要なのです。先に言ってしまうと、今後 $O(logN)$,$O((logN)^2)$[3]といった計算量がたくさん出てくると思いますが定数倍も考慮すると 1 クエリあたりこれくらいの計算量で解かなければなりません。

3 データ構造

一度、木から離れて考えてみます。少ない計算量で答えを出すためには効率良くデータを管理して効率良くデータの変更、読み出しを行わなくてはなりません。ここでは Segment Tree と BIT いうデータ構造を紹介します。これらは、数値の列に対するデータ構造です。木を扱わなければならないのに列のデータ構造なんて役に立つのか?と思われる方もいらっしゃるかもしれませんが後々紹介するテクを知ればなんでこれが木を扱うのに利用できるかわかると思います。

[3] 普通底の 2 は省略されます

Segment Tree

Segment Tree は応用範囲が広く、色々な機能を実現させられますがその中でも有名なのが RMQ(Range Minimum Query) です。RMQ では、N 要素の数列 A がある時に、

- 更新クエリ　　A_x の値を変更する.
- 最小値クエリ　区間 $[l, r)$ の最小値、すなわち $min\{A_i \mid l \leq i < r\}$ を求める.

という操作をそれぞれ $O(logN)$ で実現できます。$[l, r)$ は半開区間といい、$l, l+1..., r-1$ を表します RMQ を少し変更するだけで区間の max をとったり、区間の和をとったりもできます。Segment Tree は、下図に示すように完全二分木で、それぞれの頂点がある区間に対応する値を持っています。図のように各頂点を番号付けすると、葉以外の頂点 k の左の子は頂点 $2k+1$, 右の子は頂点 $2k+2$ となります。頂点 k から親にアクセスしたいときは頂点 $(k-1)/2$ を見ればいいです。[4]このように番号付けすると配列で管理でき、実装が楽です。[5]図を見る限り、2 の冪乗個の要素数の数列しか扱えないようですが、そうでないときは適宜クエリの答えに影響しないような値をつけ足して要素数を 2 の冪乗個にすればよいです。initialize は、N 回値の変更をすればよいです。[6]図から明らかなように数列の要素数が 2^N 個の時、Segment Tree の頂点数は $2^{N+1}-1$ 個になります。また、Segment Tree の深さは N です (根の深さを 0 とする)。すなわち要素数に対し、木の深さはおおよそ $logN$ なのです。RMQ では、次のページの図のようにいくつかの区間の最小値が計算されている

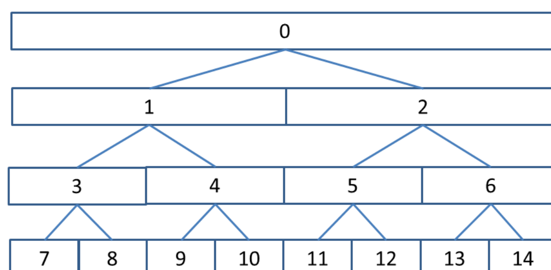


図 II.3　Segment Tree

状態を維持します。これらから、与えられた区間の最小値を区間全体を見ずに求めるので

[4] 整数型どうしの割り算では切り捨てのため
[5] これは個人の感想で、ポインタを用いてちゃんと木構造っぽく書くこともできます
[6] $O(NlogN)$ なので問題ない

す。それぞれのクエリについて順に見て行きましょう

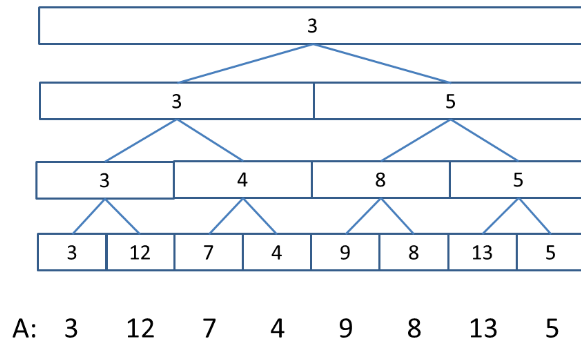


図 II.4　RMQ

更新クエリ

　図のように更新したい頂点から親にたどっていき、更新していきます。数列の要素数を $SIZE$ とすると、数列の x 番目 (0-indexed) は Segment Tree の頂点 $x + SIZE - 1$ に対応します。対応する頂点の値を変更し、そこから親をたどっていきながら更新していけばよいです。計算量は Segment Tree の深さ分なので $O(logN)$ です。

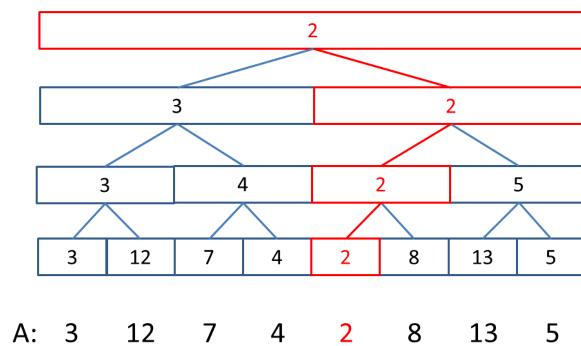


図 II.5　RMQ の更新

最小値クエリ

図のように根 (図で一番上の頂点、すなわち頂点 0) から順にたどって行きましょう。クエリが $[a, b)$, 今見ている頂点に対応する数列の区間を $[l, r)$ とすると、3 つの場合が考えられます。

- 2 つの区間が交わらない時、すなわち $r \leq a$ かつ $b \leq l$ の時
- $[l, r)$ が $[a, b)$ に完全に包含されている時
- それ以外の時

最初の場合は、答えに影響しない数 (今回の場合とても大きな数 INF) を返します。2 つ目の場合、答えが今見ている頂点の値より大きくなることはないのでそれを返します。最後の場合は、左の子と右の子について再帰的に計算してやって、それらの小さいほう (同じならその値) を返します。これで最小値クエリに答えることができます。さて、計算量は各深さにおいて見る頂点が定数個なのでこちらも $O(logN)$ です (詳しい評価は省きます)。最小

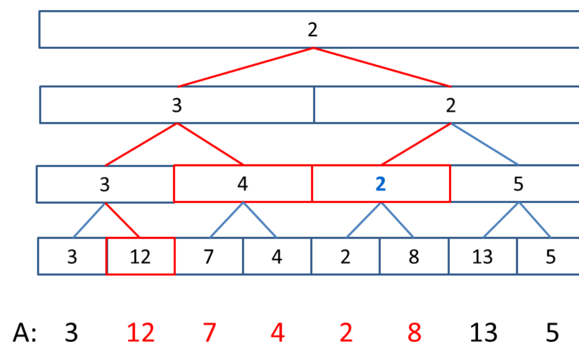


図 II.6　最小値クエリ

値クエリの実装では、今見ている頂点に対応する区間のインデックスも一緒に引数として渡してやると楽かもしれません。これで、基本的な Segment Tree が使えるようになりました。以下は RMQ の実装例です。

```
#include <algorithm>
using namespace std;
#define INF 2000000000
const int SIZE = 1<<20;
struct RMQ
{
    int seg[SIZE*2];
    void update(int k,int x)
    //k...index of sequence
    //x...value
    {
        k += SIZE-1;
        seg[k]=x;
        while(k)
        {
            k = (k-1)/2;
            seg[k]=min(seg[k*2+1],seg[k*2+2]);
        }
    }
    int query(int a,int b,int k,int l,int r)
    //a,b...indexes of sequence corresponding query
    //k...index of Segment Tree
    //l,r...indexes of sequence corresponding k-th vertex of Segment Tree
    {
        if(r<=a||b<=l)return INF;
        else if(a<=l&&r<=b)return seg[k];
        else return min(query(a,b,k*2+1,l,(l+r)/2),query(a,b,k*2+2,(l+r)/2,r));
    }
};
```

ここで突然ですが次のような問題を考えて見ましょう。

要素数 N の数列 A がある。次の 2 種類の Q 個のクエリに答えよ

- $[l, r)$ に x を足す
- $min\{A_i \mid l \leq i < r\}$ を求める

$N \leq 10^5, Q \leq 10^5$

最小値クエリは前と同じですが更新クエリが前と少し異なっています。一見 $[l, r)$ に対して一つずつ x を足してやればいいように見えますが、これでは 1 回のクエリでの計算量が最悪 $O(NlogN)$ となり本末転倒です。実は、この問題も Segment Tree に少し工夫をするだけで各クエリ $O(logN)$ で解けてしまいます。次ではその具体的な方法について見ていくことにします。

遅延評価

さきほどの Segment Tree では予め特定の範囲の最小値が計算された状態を維持することで、少ない回数の計算で様々な区間の最小値を求めることができました。今回は、値が区間に対して足されるのでそれをいくつかの区間に分けてまとめて管理したいのです。なので、先ほどまで、Segment Tree の各頂点はそれぞれの頂点に対応する区間の最小値を持っていましたがさらに"その対応する区間に一様に足された値"を持たせてみます。(正確には、一様に足された値のうち、まだ反映されていない値です。) 以降、各頂点が持っている対応する区間の最小値を minimum, 区間に一様に足された値を lazy と言う事にします。上の図のようにさきほどと同様な初期状態の Segment Tree があるとします。それぞれのクエリにつ

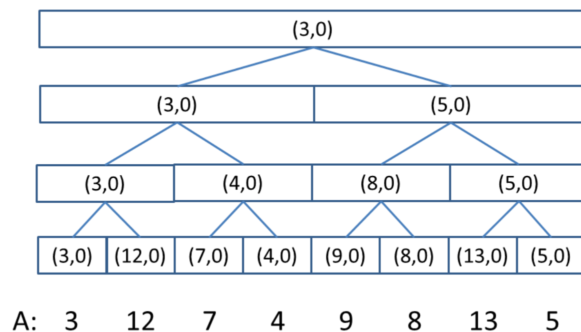


図 II.7 (minimum,lazy)

いて見ていきましょう。

更新クエリ

さきほどの RMQ の最小値クエリと同様の場合わけをします。1 つ目の場合はなにもせず終了。2 つ目の場合は今見ている頂点の lazy に値を加算。3 つ目の場合は**今見ている頂点の lazy を子に伝搬**し、して左右の子に対して再帰的に計算する。そして自分の子をもとに、自分の値を更新。これでうまく行きます。下図で、実際にどのような動きをしているのかを見ればなぜこれで良いのかわかると思います。先ほど太字で示したところが遅延評価と呼ばれるものです。

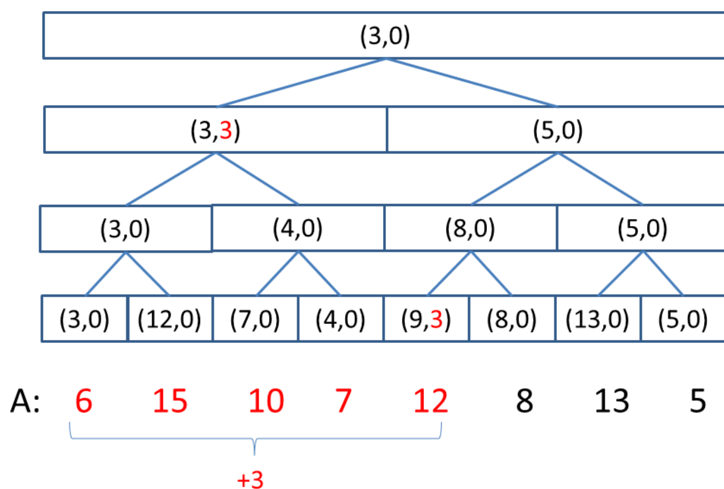


図 II.8　遅延評価 SegmentTree の更新 1

最小値クエリ

こちらは一つ前の Segment Tree とほとんど同じです。

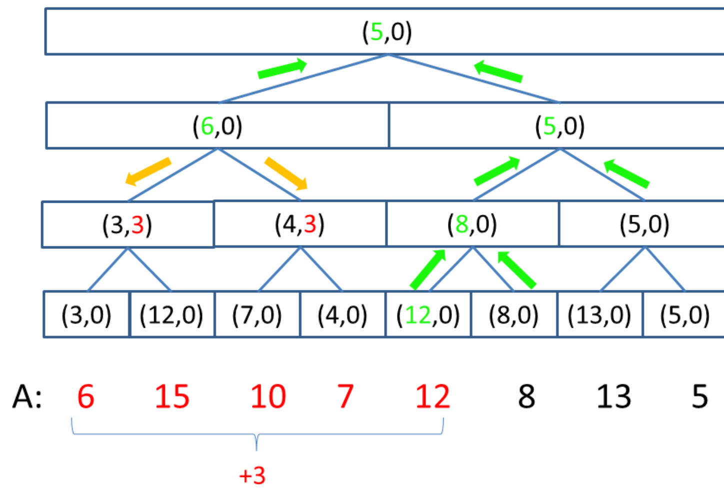


図 II.9　遅延評価 SegmentTree の更新 2

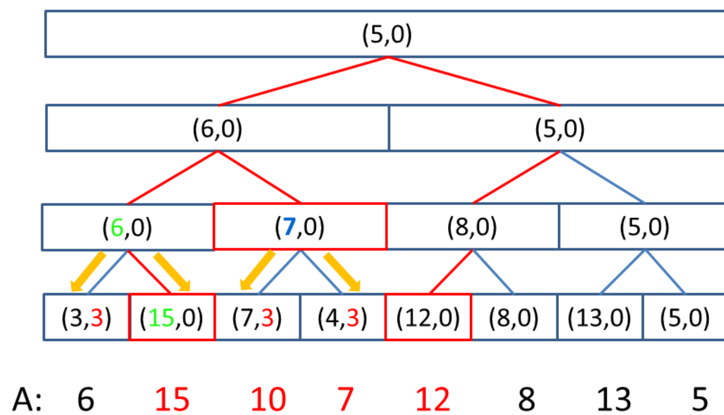


図 II.10　最小値クエリ

```
#include <algorithm>
using namespace std;
const int SIZE = 1<<18;
struct segtree
```

```
{
    int minimum[SIZE*2],lazy[SIZE*2];
    void lazy_evaluate(int k)
    {
        minimum[k]+=lazy[k];
        if(k<SIZE-1)// k isn't leaf of SegmentTree
        {
            lazy[2*k+1]+=lazy[k];
            lazy[2*k+2]+=lazy[k];
        }
        lazy[k]=0;
    }
    void update(int a,int b,int k,int l,int r,int x)
    {
        if(r<=a||b<=l)return;
        if(a<=l&&r<=b)
        {
            lazy[k]+=x;
            lazy_evaluate(k);
        }
        else
        {
            lazy_evaluate(k);
            update(a,b,k*2+1,l,(l+r)/2,x);
            update(a,b,k*2+2,(l+r)/2,r,x);
            minimum[k] = min(minimum[k*2+1],minimum[k*2+2]);
            return;
        }
    }
    int query(int a,int b,int k,int l,int r)
    {
        lazy_evaluate(k);
        if(r<=a||b<=l)return 0;
        if(a<=l&&r<=b)return minimum[k];
        else
        {
            int lch = query(a,b,k*2+1,l,(l+r)/2);
```

```
            int rch = query(a,b,k*2+2,(l+r)/2,r);
            return min(lch,rch);
        }
    }
};
```

Binary Indexed Tree

BIT(Binary Indexed Tree) は

- A_x の値を変更する
- $A_0 + A_1 + \cdots + A_x$ を求める

が $O(logN)$ でできるデータ構造です。おわかりの通り、Segment Tree で同じ機能を実現できます。なので細かいことは省いて実装だけ載せておきます。なぜこれでうまく動くのか知りたい方はぜひ調べてみてください。実装はとても軽いです。ただ、SegmentTree は 0-indexed でしたが BIT は 1-indexed なので注意してください (内部で 1 ずらしてやれば見かけ上 0-indexed として利用できます)。

```
#define MAX_N 100100
struct BIT
{
    int bit[MAX_N+1];
    void add(int i,int x)
    {
        while(i<=MAX_N)
        {
            bit[i]+=x;
            i+=i&-i;
        }
        return;
    }
    int sum(int i)
    {
        int res=0;
        while(i>0)
        {
```

```
                res+=bit[i];
                i-=i&-i;
            }
            return res;
        }
    };
```

いよいよ、木に対する様々なアプローチ、テクニックを紹介していこうと思います。

4 木に対するテクニック

LCA(Lowest Common Ancestor)

根付き木において、ある 2 頂点 u,v の両方の先祖である頂点のうち最も深さの大きい頂点を *LCA(Lowest Common Ancestor)* と言います。LCA を求めるためのアルゴリズムには

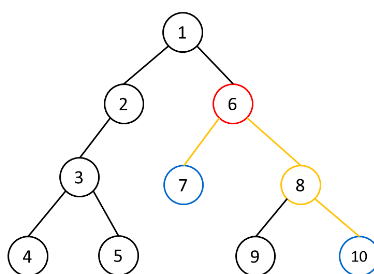


図 II.11 頂点 7 と頂点 10 の LCA は頂点 6

いろいろありますが、今回はそのうちの一つのダブリングを紹介します。ダブリングというのは、LCA 以外でも用いられる汎用的なアルゴリズムです。まず、前処理として次のようなテーブルを計算しておきます

$parent[v][i]$:= 頂点 v から 2^i回親をたどった頂点 (なければ -1)

ここで $parent[v][i+1] = parent[parent[v][i]][i]$ より
$par[\sim][i]$ が計算できれば $par[\sim][i+1]$ が計算でき、i は log(木の深さ)[7]しか必要ないので時間計算量も空間計算量 (メモリ使用量) も $O(NlogN)$ です。このテーブルが求まれば、頂点 v から x 回親をたどった頂点というのは、x を 2 進数表記した時に i 番目のビットが立っていたら 2^i 回親をたどるということをして $O(logx)$ で求まります。これが求まれば後は簡単で、まず u,v の高さを揃えるために深いほうの頂点から $|depth[u] - depth[v]|$ 回親をたどります。$LCA(u,v)$ が u,v から x 回親をたどった頂点だとすると、u,v それぞれから x+1 回以上たどった頂点もすべて一致します。なので二分探索をすることで、その境目である LCA を求める事ができます。

[7] 一般の木の場合、木の深さは最悪 N です。

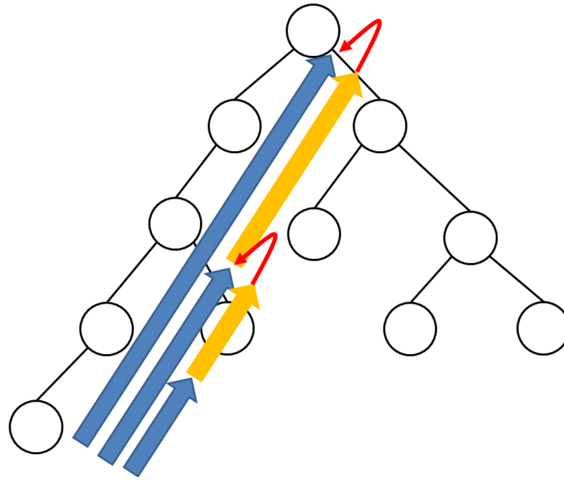


図 II.12　ダブリングのイメージ

```
#include <algorithm>
#include <vector>
using namespace std;
#define MAX_N 100100
#define MAX_LOG_N 20
int N,root;
vector<int> g[MAX_N];
int depth[MAX_N];
int par[MAX_N][MAX_LOG_N];
void dfs(int v,int p,int d)
{
    par[v][0]=p;
    depth[v]=d;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]==p)continue;
        dfs(g[v][i],v,d+1);
    }
}
void fill_table()
{
    for(int i=0;i<19;i++)
```

```
    {
        for(int j=0;j<N;j++)
        {
            if(par[j][i]==-1)par[j][i+1]=-1;
            else par[j][i+1]=par[par[j][i]][i];
        }
    }
}
int lca(int u,int v)
{
    if(depth[u]>depth[v])swap(u,v);
    for(int i=19;i>=0;i--)
    {
        if(((depth[v]-depth[u])>>i)&1)v=par[v][i];
    }
    if(u==v)return u;
    for(int i=19;i>=0;i--)
    {
        if(par[u][i]!=par[v][i])
        {
            u = par[u][i];
            v = par[v][i];
        }
    }
    return par[u][0];
}
int main()
{
    dfs(root,-1,0);
    fill_table();
    return 0;
}
```

Euler Tour

木を扱うにあたって最重要といっても過言ではないテクニックがオイラーツアー (*Euler Tour*) です。オイラーツアーというのは, 図のように木を根から DFS する時に通る頂点を一直線に並べたものです。 各頂点番号が最初に登場するインデックスと最後に登場するイ

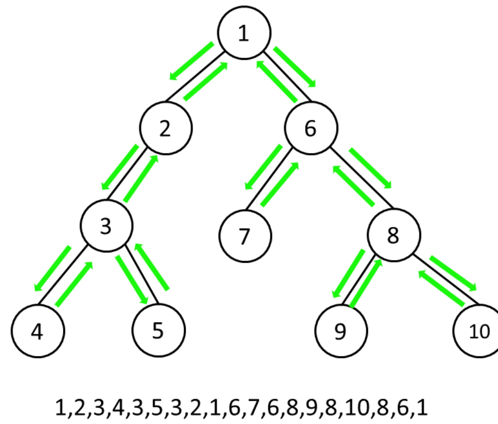


図 II.13　オイラーツアー

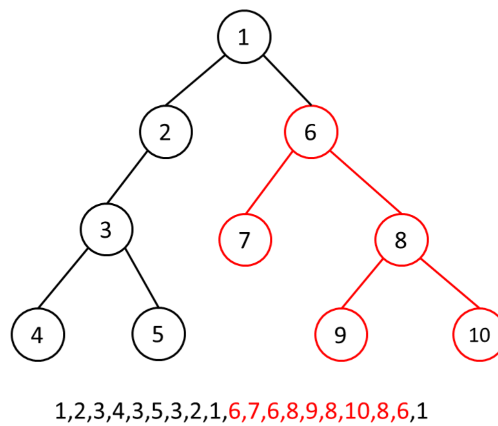


図 II.14　部分木が列に対応

ンデックスの間の区間はちょうど元の木の v の部分木に対応します。ついに木を列として扱うことできるようになりました。オイラーツアーに対して前述したデータ構造が使えるのです。例えば頂点が値を持っていて、

- 頂点 v 以下全ての頂点の値に x 加算する

- ある頂点の値を求める

というクエリがたくさんくるという問題なら、オイラーツアーを考えることで列の問題に変換でき、これは遅延評価 SegmentTree を用いて簡単に実現することができます。[8]オイラーツアーの実装は非常に簡単で、DFS で通る順にならべるので、実際に DFS するだけで O(N) でできます。オイラーツアーの要素数は、各辺を 2 回ずつ通るので植木算的に $2N-1$ だとわかります。

以下の実装で $begin[v], end[v]$ はそれぞれ v がオイラーツアー上で最初と最後に現れるインデックスです。(正確には $end[v]$ は最後に現れるインデックスの次) すなわち、$[begin[v], end[v])$ が v を根とする部分木に対応しているということです。

[8] 実は累積和を考えると遅延評価 SegmentTree ではなく BIT で解けます。基本的に、区間に同じ値を足す、区間の和を求める、というタイプならば BIT を用いればよいので遅延評価は必要ないです。

```
#include <vector>
using namespace std;
#define pb push_back
#define MAX_N 100100
vector<int> g[MAX_N];
vector<int> euler_tour;
int begin[2*MAX_N],end[2*MAX_N];
int k=0,root=0;
void dfs(int v,int p)
{
    begin[v]=k;
    euler_tour.pb(v);
    k++;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]!=p)
        {
            dfs(g[v][i],v);
            euler_tour.pb(v);
            k++;
        }
    }
    end[v]=k;
}
int main()
{
    dfs(root,-1);
}
```

パスに対するクエリ

オイラーツアーは部分木だけでなくパスに対するクエリもうまくデータをもつことで処理できます。木の辺にコストがついている時に、

- 辺のコストを変える
- u から v へのパスのコストを求める

という 2 つのクエリに答えたいとします。u, v 間のパスは、$u, lca(u, v)$ 間のパスと $lca(u, v), v$ 間のパスに分割できます。$lca(u, v)$ は u, v 両方の先祖なので、深さが単調に増加するパスのコストさえ高速に求められればこの問題を解くことができます。なので、ここでは u は v の先祖であると仮定します。まず図のオイラーツアーを眺めて見ると、部

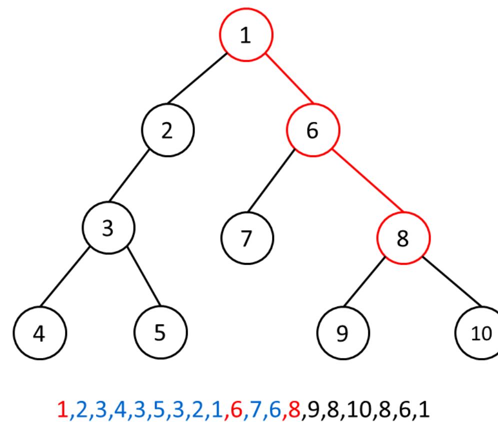


図 II.15　無駄な往復

分木の無駄な往復が含まれているので SegmentTree(または BIT) 上で $[begin[u], begin[v]]$ の和を求めてもパスのコストはわかりません。しかし、同じ辺でも根から離れる方向に移動するときはコスト正、戻る時は負にしてやると、部分木のコストは打ち消されてパスのコストが求まります。このようにコストに符号をつけて相殺するテクを知っていれば対応できる問題が増えると思います。

第 III 章

実践編

ここからは実践編と題して、問題の解説を書きたいと思います (あまり詳しい解説ではないのでわからなかった方は、各問題の公式の Editorial を参照してください)。今回は以下の問題を扱います。せっかくなら自分で解きたいという方は是非チャレンジしてみてください。おまけ以外はほぼこの記事で扱ったテクニックで解けます。各問題の詳しい概要、制約は下のリンクからご確認ください。

- Propagating tree (Codeforces Round #225 Div1 C)
  http://codeforces.com/contest/383/problem/C
- On Changing Tree (Codeforces Round #232 Div1 C)
  http://codeforces.com/contest/396/problem/C
- Running Away from the Barn (USACO 2012 December Gold)
  http://www.usaco.org/index.php?page=viewproblem2&cpid=213
- A and B and Lecture Rooms (Codeforces Round #294 Div2 E)
  http://codeforces.com/problemset/problem/519/E
- Xenia and Tree (Codeforces Round #199 Div2 E)(おまけ)
  http://codeforces.com/contest/342/problem/E

1 Propagating tree (Codeforces Round #225 Div1 C)

問題概要

各頂点が値をもった N 頂点の根付き木が与えられる。

- 頂点 x に val を加算し、x の子に $-val$ を加算、x の子の子に val を加算…
- 頂点 x の値を求める

という M 個のクエリに答えよ

加算のクエリが少し変ですね。+-交互にしないといけないので、一気にまとめて扱うのは難しそうです。ですが、x の子孫で、x からの距離が偶数のものには全て val 加算, 奇数のものには $-val$ 加算です。この性質を用いてうまくまとめて処理することを考えてみます。オイラーツアーを考えてみると、オイラーツアーで隣あう頂点の深さの差は 1 なので、深さの偶奇が同じ頂点はひとつ飛ばしに現れます。そこで、Segment Tree を 2 本用意して偶奇に分けて考えてみると、それぞれの x 以下の部分木に対応する区間にそれぞれ val,$-val$ を加算すればよく、値を知りたいときは適切なほうの SegmentTree の値を見ればわかります。なので遅延評価 SegmentTree を 2 本用意して、偶奇にわけて処理してやるとうまくいきます[1]。計算量は各クエリ $O(logN)$ なので全体で $O(MlogN)$ です

ソースコード中のオイラーツアーを生成する関数 dfs は省略します。

```
#include <cstdio>
#include <vector>
using namespace std;
#define pb push_back
const int SIZE=1<<19;
const int MAX_N = 200100;
int n,m,k=0;
int a[MAX_N];
vector<int> g[MAX_N];
int begin[MAX_N*2],end[MAX_N*2];
struct segtree
{
    int sum[SIZE*2],lazy[SIZE*2];
    void lazy_evaluate(int k,int l,int r)
```

[1] もちろん先ほどと同様に累積和をかんがえることで BIT で解けます。

```
{
    sum[k]+=lazy[k]*(r-l);
    if(k<SIZE-1)
    {
        lazy[2*k+1]+=lazy[k];
        lazy[2*k+2]+=lazy[k];
    }
    lazy[k]=0;
}
void update(int a,int b,int k,int l,int r,int x)
{
    if(r<=a||b<=l)return;
    if(a<=l&&r<=b)
    {
        lazy[k]+=x;
        lazy_evaluate(k,l,r);
    }
    else
    {
        lazy_evaluate(k,l,r);
        update(a,b,k*2+1,l,(l+r)/2,x);
        update(a,b,k*2+2,(l+r)/2,r,x);
        sum[k] = sum[k*2+1]+sum[k*2+2];
        return;
    }
}
int query(int a,int b,int k,int l,int r)
{
    lazy_evaluate(k,l,r);
    if(r<=a||b<=l)return 0;
    if(a<=l&&r<=b)return sum[k];
    else
    {
        int lch = query(a,b,k*2+1,l,(l+r)/2);
        int rch = query(a,b,k*2+2,(l+r)/2,r);
        return lch+rch;
    }
```

```
    }
    void update(int a,int b,int x){update(a,b,0,0,SIZE,x);}
    int query(int a,int b){return query(a,b,0,0,SIZE);}
};
segtree odd,even;
int main()
{
    scanf("%d %d",&n,&m);
    for(int i=0;i<n;i++)scanf("%d",&a[i]);
    for(int i=0;i<n-1;i++)
    {
        int u,v;
        scanf("%d %d",&u,&v);
        u--;v--;
        g[u].pb(v);
        g[v].pb(u);
    }
    dfs(0,-1);
    for(int i=0;i<n;i++)
    {
        if(begin[i]%2==0)even.update(begin[i],begin[i]+1,a[i]);
        else odd.update(begin[i],begin[i]+1,a[i]);
    }
    for(int i=0;i<m;i++)
    {
        int type;
        scanf("%d",&type);
        if(type==1)
        {
            int x,val;
            scanf("%d %d",&x,&val);
            x--;
            if(begin[x]%2==0)
            {
                even.update(begin[x],end[x],val);
                odd.update(begin[x],end[x],-val);
            }
```

```
            else
            {
                odd.update(begin[x],end[x],val);
                even.update(begin[x],end[x],-val);
            }
        }
        else
        {
            int x;
            scanf("%d",&x);
            x--;
            if(begin[x]%2==0)printf("%d\n",even.query(begin[x],begin[x]+1));
            else printf("%d\n",odd.query(begin[x],begin[x]+1));
        }
    }
    return 0;
}
```

2 On Changing Tree (Codeforces Round #232 Div1 C)

問題概要

各頂点が値をもった N 頂点の根付き木が与えられる。

- 頂点 v に x を加算し、v の子に $x-k$ を加算、v の子の子に $x-2k$ を加算…
- 頂点 x の値を 10^9+7 で割った余りを求める

という Q 個のクエリに答えよ

この問題も加算のクエリが少し変なタイプの問題です。この問題では深さの違う頂点に足される値は全て異なります。先ほどと同じアイデアで、深さごとにまとめて処理しようとしても木の深さは最悪頂点数と同じになるなので 1 つのクエリあたり $O(N)$ 以上かかってしまい間に合いません。足される値の性質に注目してみると深さごとに足される値が等差ですが、この性質をどう使えるのかはイマイチ見えてこないと思います。そこで足される値をもう少し正確に定式化してみましょう。v の子孫 u に足される値は $x-k*(depth[u]-depth[v])$ です。これを展開してみると $x+k*depth[v]-k*depth[u]$ となります。そこで 2 つに分けて、$x+k*depth[v]$ を v 以下の頂点すべてに足し、足しすぎた $k*depth[u]$ を引くことを考えてみます。$depth[u]$ は定数なので足し過ぎた k の合計値がわかれば足し過ぎた値を求められます。なので加算クエリが来た時に v の子孫 u について $k(=$ 足し過ぎた値$/depth[u])$ を加算をしたいということになります。これは、オイラーツアーの区間に値を足すだけですね。

まとめると、今回も 2 つの SegmenTree(または BIT) を 2 本用意して、片方には頂点 v の部分木に対応する区間に $x+k*depth[v]$ を、もう一方には k を加算することで加算クエリを $O(logN)$ で処理できます。値を求める時は、上記の計算をするだけなのでこちらも $O(logN)$ でできます。

BIT のほうが定数倍が軽いので、BIT で解けるときは BIT を使うことをおすすめします。$SegmentTree$ と dfs は省略しています。

```
#include <cstdio>
#include <vector>
using namespace std;
typedef long long ll;
#define pb push_back
#define MOD 1000000007
const int SIZE=1<<20;
const int MAX_N = 300100;
```

```
int n,q,K=0;
vector<int> g[MAX_N];
int depth[MAX_N];
int begin[MAX_N*2],end[MAX_N*2];
segtree sum,over;
int main()
{
    scanf("%d",&n);
    for(int i=1;i<n;i++)
    {
        int v;
        scanf("%d",&v);
        v--;
        g[i].pb(v);
        g[v].pb(i);
    }
    dfs(0,-1,0);
    scanf("%d",&q);
    for(int i=0;i<q;i++)
    {
        int type;
        scanf("%d",&type);
        if(type==1)
        {
            int v,x,k;
            scanf("%d %d %d",&v,&x,&k);
            v--;
            sum.update(begin[v],end[v],((ll)x+(ll)k*depth[v])%MOD);
            over.update(begin[v],end[v],k);
        }
        else
        {
            int v;
            scanf("%d",&v);
            v--;
            ll ans = sum.query(begin[v],begin[v]+1)
                     -(ll)depth[v]*over.query(begin[v],begin[v]+1);
```

```
            ans = ((ans%MOD)+MOD)%MOD;
            printf("%d\n",(int)ans);
        }
    }
    return 0;
}
```

3 Running Away from the Barn (USACO 2012 December Gold)

問題概要

各辺に正のコストがある N 頂点の根付き木が与えられる。各頂点について、子孫のうち距離 L 以下の頂点の数を求めよ。

これはクエリ形式の問題ではないですが N 頂点全てに対する答えを求めないといけないので、それぞれ独立に計算するとすれば一頂点あたり $O(logN)$ 以下で求めなければ間にあいません。

まず、各頂点について子孫のうち L 以下になるものを探すのは大変そうなので自分を距離 L 以下の子孫とする頂点たちのカウンタを増やす、という方向で考えていきましょう。頂点 v を子孫とするような頂点は根から頂点 v のパス上の頂点のみです。また、辺のコストは正なのであるところを境に、L 以下かそうでないかがわかれます。なので、パス上を二分探索することで $O(logN)$ で境目をみつけることができます (DFS で各頂点の根からの距離を求めておけば、距離は O(1) で求められる)。あとはパス上に 1 を足すだけです。ある頂点 x の値をオイラーツアーの x の部分木にあたる列の和に対応させると、$u, v(depth[u] < depth[v])$ 間に 1 を足したい時、BIT の $begin[u]$ に 1 を、$begin[v] + 1$ から 1 を引くことで実現できます。全体で計算量は $O(N(logN)^2)$ になります。[2]

```
#include <cstdio>
#include <vector>
using namespace std;
typedef long long ll;
#define MAX_N 200100
#define pb push_back
BIT bit;
```

[2] 実装を工夫すれば $O(NlogN)$ にできます

```
struct edge
{
    int to;
    ll cost;
    edge(int to,ll cost):to(to),cost(cost){}
};
int N;
ll L;
vector<edge> g[MAX_N];
int par[MAX_N][20];
ll dist[MAX_N][20];
int depth[MAX_N];
int begin[MAX_N],end[MAX_N];
int K=1;
void dfs(int v,int p,ll d,int dep)
{
    begin[v]=K++;
    depth[v]=dep;
    par[v][0]=p;
    dist[v][0]=d;
    for(int i=0;i<g[v].size();i++)
    {
        edge e = g[v][i];
        if(e.to == p)continue;
        dfs(e.to,v,e.cost,dep+1);
        K++;
    }
    end[v]=K;
}
ll search_dist(int v,int x)
{
    ll res = 0ll;
    for(int i=19;i>=0;i--)
    {
        if((x>>i)&1)
        {
            res += dist[v][i];
```

```
                v = par[v][i];
            }
        }
        return res;
    }
    int search_parent(int v,int x)
    {
        for(int i=19;i>=0;i--)
        {
            if((x>>i)&1)v = par[v][i];
        }
        return v;
    }
    int main()
    {
        scanf("%d %lld",&N,&L);
        for(int i=1;i<N;i++)
        {
            int p;
            ll l;
            scanf("%d %lld",&p,&l);
            p--;
            g[i].pb(edge(p,l));
            g[p].pb(edge(i,l));
        }
        dfs(0,-1,-1,0);
        for(int i=0;i<19;i++)
        {
            for(int j=0;j<N;j++)
            {
                if(par[j][i]==-1)
                {
                    par[j][i+1]=-1;
                    dist[j][i+1]=-1;
                }
                else
                {
```

```
                par[j][i+1]=par[par[j][i]][i];
                dist[j][i+1]=dist[j][i]+dist[par[j][i]][i];
                if(par[j][i+1]==-1)dist[j][i+1]=-1;
            }
        }
    }
    for(int i=0;i<N;i++)
    {
        int l = 0, r = depth[i]+1;
        while(r-l>1)
        {
            int mid = (l+r)/2;
            if(search_dist(i,mid)<=L)l = mid;
            else r = mid;
        }
        int p = search_parent(i,l);
        bit.add(begin[p],1);
        bit.add(begin[i]+1,-1);
    }
    for(int i=0;i<N;i++)printf("%d\n",bit.sum(begin[i])-bit.sum(end[i]));
    return 0;
}
```

4 A and B and Lecture Rooms (Codeforces Round #294 Div2 E)

問題概要

辺のコストが全て 1 の N 頂点の木が与えられる。

- 頂点 u,v からの距離が等しい頂点の数を求める

という M 個のクエリに答えよ

まず、u, v 間のパスの長さが奇数の時、感覚でわかると思いますが、答えは明らかに 0 です。念の為証明してみましょう。木では隣あう頂点に移動した時に必ず深さは 1 ずつ変化します。なので u, v のパスの長さが奇数の時、u, v の深さの偶奇は異なります。ゆえに u, v から同じ距離の頂点の深さの偶奇が一致しないので、そのような頂点がないことがわ

かります。次に偶数の時を考えてみましょう。u, v 間のパスの真ん中の頂点 x は u, v からの距離が等しいです、また x の子のうち,u, v の先祖でない頂点を根とする部分木の頂点全ても u, v からの距離が等しいです。これだけで大丈夫なように見えますが頂点 x が丁度 u, v の LCA と一致するときは x の先祖も u, v からの距離が等しくなります。また、いやらしいコーナーケースですが u, v が同じ頂点の時は u, v 自身も u, v からの距離が等しくなります。[3]

```
#include <cstdio>
#include <vector>
using namespace std;
#define pb push_back
#define MAX_N 100100
int n,m;
vector<int> g[MAX_N];
int depth[MAX_N],child[MAX_N];
int par[MAX_N][20];
int dfs(int v,int p,int d)
{
    par[v][0]=p;
    depth[v]=d;
    child[v]=1;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]==p)continue;
        child[v]+=dfs(g[v][i],v,d+1);
    }
    return child[v];
}
int search(int v,int x)
{
    for(int i=19;i>=0;i--)if((x>>i)&1)v=par[v][i];
    return v;
}
int main()
{
```

[3] 要するに全頂点への距離が等しいということです

```
scanf("%d",&n);
for(int i=0;i<n-1;i++)
{
    int a,b;
    scanf("%d %d",&a,&b);
    a--;b--;
    g[a].pb(b);
    g[b].pb(a);
}
dfs(0,-1,0);
fill_table();
scanf("%d",&m);
for(int i=0;i<m;i++)
{
    int a,b;
    scanf("%d %d",&a,&b);
    a--;b--;
    if(a==b)
    {
        printf("%d\n",n);
        continue;
    }
    if(depth[a]<depth[b])swap(a,b);
    int c = lca(a,b);
    int dist = depth[a]+depth[b]-depth[c]*2;
    if(dist%2!=0)
    {
        printf("0\n");
        continue;
    }
    dist/=2;
    int u = search(a,dist);
    int v = search(a,dist-1);
    if(dist==depth[a]-depth[c])
    {
        int w = search(b,dist-1);
        printf("%d\n",n-child[v]-child[w]);
```

```
        }
        else printf("%d\n",child[u]-child[v]);
    }
    return 0;
}
```

5 Xenia and Tree (Codeforces Round #199 Div2 E)

問題概要

辺のコストが全て 1 の N 頂点の木が与えられる。初め、根は赤く、その他は青く塗られている

- 頂点 v を赤く塗る
- 頂点 v からもっとも近い赤い頂点までの距離を求める

という M 個のクエリに答えよ

おまけです。この問題は、おそらくいままで紹介した手法では解けないと思います。ですが、紹介した手法も使います。まず、ナイーブな方解法を考えてみましょう。最初に思いつくのは、更新が起こるたびに BFS をして $O(N)$ で最短距離を更新する、という解法だと思います。また、更新クエリが連続して来る時、それらはまとめて $O(N)$ で処理することができます。[4]この性質をうまく使ってやることは出来ないでしょうか。更新クエリをある数 (B とおく) ずつに分割して、更新クエリは各まとまりごとに処理することにします。そうすると、前から順番にクエリを処理していったときに、距離を答えるクエリの前に来る更新クエリのうち、最短距離に反映されていない更新クエリの個数はたかだか B-1 個しかないという状態になります。また、まとめて処理したものからの最小距離と、反映されていない更新クエリの頂点との距離の最小値のうちの小さいほうが答えとなります。2 頂点間の距離は LCA を用いて $O(logN)$ で求められるので、計算量は $O(N*M/B+M*B*logN)$ になり、B を $\sqrt{N}$ 程度にしてやると、時間内に解くことができます。このような手法はクエリの平方分割と呼ばれたりします。[5]以下のソースコードでは $dfs, fill_table, lca$ を省略しています。

```
#include <cstdio>
#include <algorithm>
#include <vector>
```

[4] 最初にキューに入れる始点を増やすだけです。

[5] クエリの個数 N のとき、B を $\sqrt{N}$ 程度にするとうまく行く事が多いため。バケット法とも言います。

```
#include <queue>
using namespace std;
#define pb push_back
const int SQR = 334;
int N,M;
vector<int> g[100100];
int depth[100100];
int par[100100][20];
int d[100100];
int dist(int u,int v)
{
    int l = lca(u,v);
    return depth[u]+depth[v]-depth[l]*2;
}
vector<int> lazy;
void calc()
{
    queue<int> q;
    for(int i=0;i<lazy.size();i++)
    {
        d[lazy[i]]=0;
        q.push(lazy[i]);
    }
    while(!q.empty())
    {
        int v = q.front();
        q.pop();
        for(int i=0;i<g[v].size();i++)
        {
            if(d[g[v][i]]>d[v]+1)
            {
                d[g[v][i]]=d[v]+1;
                q.push(g[v][i]);
            }
        }
    }
    lazy.clear();
```

```
}
int main()
{
    scanf("%d %d",&N,&M);
    for(int i=0;i<N-1;i++)
    {
        int a,b;
        scanf("%d %d",&a,&b);
        a--;b--;
        g[a].pb(b);
        g[b].pb(a);
    }
    dfs(0,-1,0);
    fill_table();
    for(int i=0;i<M;i++)
    {
        int t,v;
        scanf("%d %d",&t,&v);
        v--;
        if(t==1)
        {
            lazy.pb(v);
            if(lazy.size()>=SQR)culc();
        }
        else
        {
            int ans = d[v];
            for(int i=0;i<lazy.size();i++)ans = min(ans,dist(v,lazy[i]));
            printf("%d\n",ans);
        }
    }
    return 0;
}
```

第 IV 章

おわりに

私の記事を最後まで読んでいただきありがとうございます。本当は Heavy-Light Decomposition などもう少しむずかしめのテクニックまで扱う予定だったのですが締め切りに追われてしまい結局書くことができませんでした。また、実践編の解説では自分の日本語力が足りないために非常にわかりにくい説明になってしまい申し訳ありません。ですが、この記事を通して、「木って奥深いなぁ」とか「グラフ理論おもしろそう」と感じて頂いたり、「競技プログラミングを始めてみたい」と思っていただけたなら幸いです。ここで紹介できなかった問題もいろいろな Online Judge にありますので、是非挑戦してみてください。もし間違ってる箇所の指摘などがありましたら@okuraofvegetabl まで連絡していただけると嬉しいです。

参考文献

[1] 秋葉拓哉, 岩田陽一, 北川宣稔『プログラミングコンテストチャレンジブック 第 2 版』(マイナビ,2010)