Theoretical Science Group

理論科学グループ

部報 **312** 号
— 駒場祭パンフレット号 —

目　次

展示企画 **1**

HTML5 Canvas と JavaScript を使ったアニメーション . . . . . 【lip_of_cygnus】 1

TSGCTF . . . . . 【yamaguchi】 2

一般記事 **3**

正規表現超絶技巧 . . . . . 【moratorium08】 3

それでも僕は Ruby で CTF をする . . . . . 【cookies】 11

東方文花帖を AI に自動でクリアしてもらう話 . . . . . 【satos】 18

Green Hackenbush の必勝法について . . . . . 【dai】 28

展示企画

HTML5 Canvas と JavaScript を使ったアニメーション

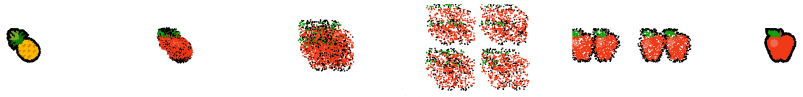
lip_of_cygnus

展示内容の説明

パソコンの画面用内に円 (粒子) が数百個から数千個描かれています。粒子は壁に当たると跳ね返ります。ある一定時間がたつとそれらの粒子が集まって文字を形成します。しばらくすると粒子が移動し別の文字が形成されます。

技術

まず表示させたい文字を Canvas を利用して画像に変換します。その画像情報を利用して一定以上の透明度がある点の座標のリストを取得します。次に、それらの点の座標のリストの中から 1000 個程度ランダムに取得して円を動かす関数の引数に代入します。ただし、本当にランダムに点を選ぶと偏りが発生するためある程度均等に選ばれるようにしています。すると粒子を動かす関数内で円が最終的にたどり着く目標地点が定まるのであとは現在の位置から逆算して粒子の動きを決めます。一連の計算や操作には JavaScript を用いています。下の画像はパイナップルからリンゴに変化する例です。絵文字も使用できるためこのような表現が可能になっています。



おわりに

ブラウザ上で動くことから、多種多様な環境でも動かせることが面白いです。去年に引き続き図形の描画を行う展示になりましたが進歩したのではないかと思っております。

TSGCTF

yamaguchi

はじめに

情報科学科二年の yamaguchi です。オンサイト CTF というほど立派なものではありませんが、初心者向けの簡単な問題とガチ勢向けのまぁまぁ難しい問題の二本立てでやろうと思っています。できればスコアサーバも作りたい…。全完すると粗品が出るかもしれません。

経緯

去年から今年にかけて、TSG で CTF が熱い！ CTF 分科会自体一年前が最初の開催で、そこからたまにオンライン CTF にチームで出たりしています。オンサイト CTF をやろうと思ったのは、オンサイト CTF がとても楽しかったからです。

CTF とは？

Capture The Flag の略。オンラインやオフラインのセキュリティを題材にしたいわゆるハッキングコンテストです。オンライン (世界中のチームが参加する) だと jeopady というチームで与えられた問題を解く形式が一般的で、今回もこれに則ります。日本で有名なコンテストだと SECCON があります。テレビなどで「ハッキングのコンテスト！」と紹介されるのはたぶんこれです。

開催について

今回、初心者向けに簡単な問題には部報とは別途解説を付けるつもりではありますが、もし CTF に興味を持ってくれた人がいたら以下のリンクを参考にしてみてください。初めての人向け常設 CTF http://www.cpaw.site/ 初〜中級者向け常設 CTF http://ksnctf.sweetduet.info/
オンライン CTF まとめサイト https://ctftime.org/

一般記事

正規表現超絶技巧

moratorium08

この記事では Ruby 2.3.0 を用いたときの正規表現の例について書いています。他の正規表現エンジンとは挙動が違うことが “実際に” 存在し、必ずしも以下に示す正規表現が他のエンジンで動くことを保証するものありません。

前書き

先日 HITCON CTF 2016 という CTF 大会に参加した際に出た問題に、正規表現の可能性を感じさせられたために書いた文章です。Ruby の正規表現の機能の概要と使い方について書いています。何も知らない人でも一通り読めば正規表現の概要とその可能性はつかめるだろうと思います。

正規表現とは

文字列が、ある文字列の集合に入るための条件を表す表現のことである。例えば、

`^[0-9]+$` は、1 桁以上の数の連続にマッチする。このような、メタ文字と文字の組み合わせによって、正規表現は構成される。基本となるメタ文字は`^ $ * + . ? | [] ()` である。

正規表現の基本

正規表現	意味
^	文頭を表す
$	文末を表す
	その前の部分の連続 0 回以上を表す
+	その前の部分の連続 1 回以上を表す

正規表現	意味
?	その前の部分が 1 回または 0 回あることを表す
.	任意の文字を表す
|	選言に使う

また、文字クラスという概念があり [] でくくることで、文字の集合を定義でき、集合内に存在する任意の文字にマッチする。例えば平成 x 年 (x は半角の数) というのをマッチしようと考えると

```
^平成([0-9]+|元)年$
```

のように扱う。また、半角英数文字+アンダーバーを表す \w や数字の集合を表す \d のように、よく使うクラスに関しては別途に特別な表現が存在する。正規表現は基本的に、これらの文字を組み合わせることで達成される。

正則文法の限界

これまでの枠組みで表せるものは「正則文法」と言われる。これだけでも世の中の多くのルールを表現することが可能である。例えば、1 つの「,」を挟んで任意の数字列が並んでいるというような正規表現を考えるとする (e.g. 123,5555 のような)。これは例えば

```
^\d+,\d+$
```

によって表現される。ではここにさらに、「『同じ文字数だけ』「,」を挟んで任意の数字列が並んでいる」という表現を考えたいとする。これは、ここまでの枠組みでは表現不可能であり、さらに広い枠組みである文脈自由文法が必要となる。では、どう達成するのか。

田中哲スペシャル

Ruby の正規表現には、さらに強力な道具として、再帰と後方参照がある。用いられる記号は次の三つである。

名前	正規表現	意味
名前つき捕獲式集合	(?<name>)	グループを name で名付ける
部分式呼び出し	\g<name\>	name の表現を再帰的に参照する
後方参照	\k<name>	name にマッチした最も最近の値を参照する

注. 別名として、部分式呼び出しは「田中哲スペシャル」と呼ばれる。なお、以下では短く「再帰」と呼ぶ（短く、またわかりやすいと思うので）。

また、名前だけでなく何番目のグループか、でもまた参照可能である。これは\g<1>や\k<1>のように表現する。\k<1>はさらに省略して、\1 のように表記可能である。

そして、後方参照は、通常は参照された名前や番号について、最も直近のものを参照するが、再帰と組み合わせた時に「再帰のネストレベル」によって参照が可能でありこれは、\k<name+1>や、\k<name-1>のように自分からのオフセットによって指定ができる。

さっきの問題をこれらを使って表現してみると、

```
^(?<recursion>(\d\g<recursion>\d|\d,\d))$
```

と表現できる。これは、一番外側に recursion と名付けたグループを構成し、その内側に、「数字 1 文字+ S + 数字 1 文字」または「数字 1 文字 + , + 数字 1 文字」にマッチするような再帰を置く。ここで、S は recursion と名付けたグループが再帰的に構成されることを意味する。この表現は先に説明した「何番目のループか」を用いるとさらに短くできる。また、見栄えのために「,」1 文字のみも、「『同じ文字数だけ』「,」を挟んで任意の数字が並んでいる」に含まれていると考えると

```
^(\d\g<1>\d|,)$
```

（この表現は「,」にマッチするという点で先の表現と同値ではない）

ここまで短くできる。この番号による参照は、正規表現ショートコーディングには必須技術であり、今後もまず、名前付けで考えやすくした表現を、最後は番号による参照で短縮することにする。

さらに、この問題はもっと短くすることができる。今回の駒場祭で展示している Regexp ショートコーディングゲームに挑戦してもらいたい（僕としてはこれを書いている段階で 10 文字まで短くできた）。

先読み・後読み

先読み、後読みとは、マッチの結果を汚さずに文字列を判定する方法である。具体的には次の通りである。ただし X は正規表現とする。

正規表現	意味
(?=X)	肯定的先読み
(?!X)	否定的先読み
(?<=X)	肯定的後読み
(?<!X)	否定的後読み

イメージとしては、先読みは「ちらっと見て戻す」、後読みは「行き過ぎてから確認する」である。具体的には、

```
(?=1)\d+
```

とすると、1 から始まる数字列にマッチする。このとき例えば

```
(?=1)(?<val>\d+)
```

に対して、「1234」をマッチさせたときは、val には 1234 がそのまま入る。マッチを汚さないというのはつまりこのことで、先読みは 1 であることは確かめるが、それをそっと戻すのである。

HITCON CTF 2016 RegExpert/moRE

僕が正規表現の “” つよさ “” に気づいたのは、この CTF 大会で正規表現のショートコーディングに挑戦したためであり、面白い問題が多いので、いくつか問題と解法を紹介することにする。

Palindrome

原文は以下である。

```
============= [Palindrome] ==============
Both "QQ" and "TAT" are palindromes, but "PPAP" is not.
```

つまり、回文を表すような正規表現をかけ、という問題である。これもまた文脈自由文法であり、したがって再帰を必要とする。ややくだけた表現になるが、

- S -> xSx (x は同じ文字)
- S -> x
- S -> ε (空文字)

のようなイメージで構成をすればよい。つまり、再帰をしていく中で、どこか “中点” があって、そこでは任意の 1 文字または空文字であり、その周りに同じ文字をくっつけていく、というイメージである。これを正規表現で表せば

```
^(?<rec>((?<char>(.))\g<rec>\k<char+0>|.?))$
```

のようになる。ここで \k<char+0>は同一ネストレベルにおける char を参照していることを表す。さらに、これを短くしてみると、

```
^((.)\g<1>\2|.?)$
```

となる。ここでは、上に示した後方参照の省略記法を用いている。

x^p

```
================= [x^p] =================
A prime is a natural number greater than 1 that
has no positive divisors other than 1 and itself.
```

x という文字が素数回繰り返される場合にのみマッチするような正規表現を考えろという問題である。素数は、1 とその数以外に正の約数を持たない数であり、これはすなわち、1 より大きい正の数二つの積で表せないことを意味する。これを正規表現で表すことができる。

```
^(?!(?<val>(xx+))\k<val>+$)xx+$
```

一見すると少しわかりにくいのでいくつかに分解してみる。

```
(?<val>(xx+))\k<val>+
```

は、合成数にマッチする。(xx+) は x の 2 以上の連続を表し、これを val という名前でグループ化している。その後に、後方参照によって、少なくとも一回以上このグループにマッチした文字列が連続しているものにマッチする。これは、すなわち val グループを約数として持つかをテストしているに他ならない。具体的に言えば、「xxxxxxxxx」という文字列は val グループには「xxx」が入り、これが 3 回 (\k<val>+の連続としては 2 回) 連続するものとしてマッチする。ただし、\k<val>+のあとに、$がないと、文字列の一部をとって分解できるものを探してしまうので注意する。

問題は、素数であること、すなわち合成数でないことであるから、否定先読みを用いて、これを否定する。さらにマッチすべきものとして、xx+を与えることで全体の回答が完成する。まとめると、

- x の合成数個分の連続でない
- x が 2 文字以上である

という条件付けを、正規表現で表したことになる。正規表現と素数が関連づけられるというのはなかなかに興味深い。これもさらに短くすることが可能で

```
^(?!(xx+)\1+$)xx+$
```

とできる。

x^(n^2)

```
=============== [x^(n^2)] ===============}
We like squares: ■□■□■□■□■□■□
```

x を平方数回繰り返したもののみにマッチするような正規表現を考える、という問題である。つまり xxxx は良いが、xxxxx はダメである。これは、x の個数を数列の総和のように捉えるという発想をする。すなわち

$$x^2 = \sum_{i=1}^{x}(2i-1) \tag{3.1}$$

という等式を思い出せば、

$$f(x) = 2x - 1 \tag{3.2}$$

なる数列と同じ個数の x にマッチするように前から構成していけば良いとわかる。

```
(?<term>(xx\k<term+0>|^x))+$
```

あとはこれを短縮して、

```
(xx\k<1+0>|^x)+$
```

とできる。

Complement

```
============= [Complement] ==============
Please match x,~x where x is a binary string.
Like "0,1" or "1001,0110".
```

最後に、この問題を紹介する。この問題に関しては、非常に面白いので、解答を以下の文章でネタバレを食らうともったいないかもしれない。また、少し考えてみないとなぜこの問題が難しいかが捕らえにくいかもしれない。

この問題の困難さは、もはや文脈自由文法ですらないという点である（実は、これ以外にも文脈依存文法な問題はあったが、先読み・後読みで対処可能だった）。したがって、今までのような再帰で表すことはできない。この問題に対しては「ネストの数をポインタとして後方参照をする」というテクニックを用いる。

少し難易度を下げた問題として、「, を挟んで同じ文字列が二つ並んでいる」というような正規表現を考える（例えば abc,abc）。また、簡単のために、「,」のみの入力はなく、二つの並んでいる文字列のそれぞれの長さは同じとする。この設定の上で「ネストの数をポインタとして後方参照をする」テクニックについて説明する。まず要件を満たす正規表現は

```
^(?<former>(|(?<word>(.))\g<former>)),(?<latter>(|\k<word+0>\g<latter>))$
```

これは、例えば「ab,ab」にはマッチするが、「ab,aa」にはマッチしない。前半の

```
(?<former>(|(?<word>(.))\g<former>))
```

では、1 文字ずつ前から文字を読んでいき、1 文字につき、1 つの再帰を行う。すなわち、1 文字につき、1 つネストが深くなる。後半では、このネストの深さを活用する。すなわち、

```
(?<latter>(|\k<word+0>\g<latter>))
```

において、同じネストの位置にある前半部分の文字 \k<word+0>と一致するものとマッチし、また、後半も 1 文字調べるごとに 1 つネストを深くしている。この方法により、ネストの深さを用いれば、離れている場所の文字列とも巧妙にマッチさせることができるということがわかる。元の問題に戻れば、1 か 0 の文字列が二つ「,」を区切って渡されて、それらの補数、すなわち 1 と 0 を逆転したものにマッチすればよい。今回、1 か 0 しかないので、これはつまり 1 文字ずつ見ていき、同じでないことを確認すれば良い。よって、

```
(?<former>(|(?<number>(0|1))\g<former>)),
(?<latter>(|(?!\k<number+0>)[01]\g<latter>))$
```

これを縮小して、

```
(|((0|1))\g<2>),(|(?!\k<3+0>)[01]\g<5>)$
```

となる。さらに、今まで前提としていた、「,」でない、と「前後が同じ長さの文字列である」という過程を先読みで判定する処理を加えて、

```
(?!,)(?=^(.\g<1>.|,)$)(|((0|1))\g<2>),(|(?!\k<3+0>)[01]\g<5>)$
```

となる。この問題を考えてみると、後方参照と再帰を組み合わせることによって、メモリのような機能を果たしている、という様相が非常に強力であるとわかる。

そのほかの **Tips**

控えめなマッチ

通常のマッチでは、マッチできる範囲ならば、できるだけマッチしようとする。一方で、控えめなマッチでは、マッチできる範囲でできるだけマッチしないようにマッチする。これは、主要な繰り返し制御文字に対して対応するものが存在し次の通りである。

最大マッチ	最小マッチ
*	*?
+	+?
?	??
{min,max}	{min,max}?

最大マッチと最小マッチの違いは、例えば aabb という文字列に対して、次の二つの正規表現は

```
[ab]+?b
[ab]+b
```

前者はマッチするのが「aab」であるのに対して、後者は「aabb」までマッチする。

範囲指定

範囲指定とは、その前の文字セットの繰り返し回数を指定する方法であり、具体的には次のようにする。

```
{cnt}
{min,max}
```

の形式で、指定する。前者は cnt 回の時に、後者は min 以上 max 以下の時にマッチする。例えば

```
^A{3}$
^A{1,4}B$
```

前者は、AAA のみにマッチする。後者は AAAAB にはマッチするが、B や AAAAAB にはマッチしない。これを用いた応用例として、関数的定義をすることが挙げられる。これは

```
(?<function>x){0}
```

と書くことで、名前として、function というグループが作られるが、0 回の連続、すなわちマッチには影響させない、という状態になる。例えば、「n 文字の連続が二種類の文字で連続し (a^nb^n のような形)、これが 3 連続する」にマッチするという正規表現を考えたい (つまり aaabbbccddef など)。

```
^(?<x>.){0}(?<y>.){0}(?<set>(?=\g<x>\k<x>+\g<y>)){0}
(?<f>\k<x>\g<f>?\k<y>){0}\g<set>\g<f>\g<set>\g<f>\g<set>\g<f>$
```

のように書く事ができる。このように書く事から見えることとして、\g は再帰というよりもむしろ関数呼び出しのように扱えるということである。これはよりルールが複雑化したときに正規表現の煩雑さを抑えるのに有効である。

あとがき

書き終わってみると、超絶技巧というのは少し誇張のような気もしてきたのですが、タイトルは少しかっこつけたかったので許して下さい。もっと一般論ができるとよかったなあと思っています。

それでも僕は **Ruby** で **CTF** をする

cookies

前置き

CTF

競プロくらい有名になってくれると紹介しなくてもいいんだけど、これをまともに読もうとするような人にも CTF を知らない人は全然いる気がするので紹介しなければならない。

CTF は Capture The Flag の略で、一応コンピュータセキュリティ技術の有無を計れるようなクイズ [1]、ゲームと言われている。コンピュータセキュリティ技術に関連するといっても内容は様々で、例えば実際にマルウェアを扱う研究所・会社では行われているであろう機械語が埋め込まれた実行可能ファイルの解析も含まれるし、最近の話題から/実行可能ファイルを眺めて/ソースコードを眺めて/試行錯誤から/勘と経験 [2] で脆弱性を探して突いてみるものもあるし、ある人物についての情報収集をさせる問題もあるし、普通に使う分には見過ごしてしまうファイルの中のメタ情報を話題にする問題もある。

全てに共通するものではないが、ひとつ CTFer の好きなものがあって、コンピュータ技術であまり人が気にもしない重箱の隅や、普通の人ならブラックボックスとしてしまう中身なんかを理解しようとすると良い。

たとえば、Windows のごみ箱ってあれ C:\ から辿れるのかとか [3]。たとえば、みんな大好き表計算ソフト Excel の拡張子は xls,xlsx,xlsb などがあるが、これらの違いをご存知だろうか。[4] レベルを上げると、たとえば、C 言語でもなんでもいいけど、rand 関数の中身とか。MD5 とか SHA1 がなにしてんのかとか。コンパイルって何をして、何が出てきているんだろうとか。

実際、被害の大きな脆弱性って、たくさん使われているんだけど、その中であんまり人が気にもしてない部分から出てきたりするわけだ。

[1] これは jeopardy 型 CTF になるだろうか。

[2] いわゆるエスパー。あまりに高いエスパー力を要求する問題は、「エスパー問」と呼称され、あまりよくない評価をされることが多い。

[3] これは僕が好きなだけで CTF に関連したところを見たことはありません。

[4] 実のところ、私もよくは知らない。そして私は Excel は好きなんだけども今パソコンには Excel が入っていないので調べることもしづらい。

タイトルの意味

CTF をするにあたって、わりとプログラミングをする場面は多い。どうせ全てのデータは紙とかではなく（それが画像データにせよ）データで与えられるので。プログラミングしなくてもよいのに問題を早く正確に解くためにすることも多々ある。テキストデータで 50 個の数字が与えられたときに、その和を計算するために普通の電卓に打ち込むことはない。そんなとき、それこそ高性能電卓的にスクリプト言語がよく使われる。スクリプト言語というと、JavaScript とか Python とか Perl とか Ruby とか PHP とか HSP とか Scheme なんかが思いつく。しかし、なぜだろうか、CTF をするときには Python を使う人が圧倒的に多い。write-up[5] を読んでも、C 言語を除けば Python がほとんどではないだろうか。理由の一つとして、「Python を使う人が多いから」というのがある。これは間違っていなくて、それゆえに CTF に使えるような Python 製ライブラリも多い。[6]

しかし私は、Ruby に慣れてしまって今のところ手放す気にもならないので、ずっと使い続けている。そういうわけで、CTF ならでは Ruby を書く人の気をつけるべきことや便利なものをここに記そうというわけだ。

本題

環境

ここに書く話は基本、ArchLinux 64bit, Ruby 2.3 を普段使っている状況で書く。Ruby の console は irb より **pry** がよい。補完も強いし、色がついてわかりやすい。

バイナリデータの取扱い

CTF では、情報を扱うときに、それがバイナリデータとして表現されていることを強く意識させられることがよくある。アルファベットにせよ日本語文字にせよどんなファイルにせよ、数値として保存されいるわけで、そのようにして扱うことを要求される。

エンコーディング

Ruby で文字列を表す String class では、インスタンスが encoding を持つ。バイナリデータとして見ようというときは、むしろ邪魔だ。’ あ’.size が 1 になってしまう [7]。

そこでバイナリデータを表す encoding が **BINARY** だ。

ソースコードをファイルに保存しているときは、magic comment として、ファイル先頭に

[5] CTF の大会が終わったあと、各自がブログなどに反省を込めて解法を公開するもの。
[6] peda, pwntools, xortool, scapy など。
[7] ruby 1.9 以降

```
# encoding: BINARY
```

と書いておく。

console で作業している場合などは、読み込んだ文字列オブジェクトに対し、随時

```
|force_encoding('BINARY')|
```

を呼ぶ。

これで文字列結合、部分文字列参照などがすんなりといく。

よく使うデータの変換

一文字と **ASCII** コードの変換 `'a' <=> 97`

`'a'.ord == 97`

`97.chr == 'a'`

文字列と **1byte** 数値の変換 `'abcd' <=> [97, 98, 99, 100]`

`(=>) 'abcd'.bytes.to_a` が 1 番簡単だろう。.bytes までで Enumrator のオブジェクトが取れる。

`(<=) [97,98,99,100].pack('C*')` pack は **pack** テンプレート文字列というやつを引数に取る。次の節で詳しく述べる。

文字列と **4byte** 数値の変換 `"xafxbexadxde" <=> 0xdeadbeef <=> "xdexadxbexef"`

これも pack テンプレート文字列 V,N を使う。

数値の底変換 `'1100001' <=> 97 <=> '61'`

それぞれ、'a' に対応する ASCII 文字コードの、2 進表示、16 進表示、10 進整数である。

`'1100001'.to_i(2) == 97`。8 進数、16 進数のみ特別に `String#oct, #hex` が用意されている。`map` を使うときなんかは、`to_i` を使うと `map(&:hex)` という風にできないから、ちょっと役に立つ。引数に 0 を与えると、prefix から判断されるようになる。

`97.to_s(16) == '61'`。

どちらも、底は 36 までが許される。36 とは、26+10 であり、alpha numeric の限界である。

pack テンプレート文字列

公式ドキュメント [8] の例のセクションから以下引用する。

> pack を使用しなくても同じことができる場合はその例も載せています。 pack は暗号になりやすい面があることを考慮し、pack を使いたくない人 に別解を示すためです。

[8] https://docs.ruby-lang.org/ja/latest/doc/pack_template.html 現在、Ruby2.3.0 版

はい。

pack テンプレート文字列というのは、`Array#pack, String#unpack` の引数に使えるパターンなのだが、よく C 言語なんかの `printf` に与える書式文字列がまあまあ似ているものかもしれない。

引用したとおり、使うと暗号になりがちなのだが、各アルファベットに相当な意味がこもっていて、なかなか覚えられないというのが大きい理由だろう。私はそのうち「C, V, N, Q, B, H」くらいしか覚えていないし CTF では使わない。[9]

C
: 前に書いたとおり。1byte 文字を unsigned char と対応付ける。

V, N, Q
: V,N は 32bit 整数、Q は 64bit 整数と対応付ける。どれも unsigned。
V,N の違いはエンディアンで、V がリトルエンディアン、N がビッグエンディアンすなわちネットワークバイトオーダー。
Q はリトルエンディアン。
pwn 問題でよく使う。

B, H
: 2 進、16 進文字列をそれの示すバイナリ文字列と対応付ける。
1byte は 8bit のため、`['1111'].pack("B*").unpack("B*")` は `['11110000']` と言った風に可逆でないので注意。

これらの文字のあとに数字をつけると、それだけ文字を繰り返したことになる。`pack("B4")` は `pack("BBBB")` となる。さらに、数字の代わりに、「*」をつけると、その位置から残りすべてを表す。

ひとつ、よくはまるのが、たとえば整数 12345 を V に通して文字列を得たいとき、`12345.pack("V")` としてしまいがちだが、`[12345].pack("V")` が正しい。Integer に pack なんてない。

Array の取扱い

Array をうまく調理できるのも大事。

順列・組み合わせ

あるデータを並べ替えたもので総当たり攻撃、なんてことをしたくなったとき使える。

`Array#permutation, combination, repeated_permutation` がある。どれも要素数を引数に取る。メソッド名から分かる通り、それぞれ順列、組み合わせ、重複組合せとなる。

たとえば'flag' のアナグラムをすべて得るには、

```
'flag'.chars.permutation('flag'.size).map(&:join)
```

[9] ただ、これを極めて見るのも面白いかもしれない。

とかする。

たとえば英数字からなる長さ 3 の文字列全ては、

```
[*?a..?z,*?A..?Z,*?0..?9].repeated_permutation(3).map(&:join)
```

とか。

添字を覚えていたい

sort とかをしてしまうと、もともと何番目のデータだったのかを忘れてしまう。そのためには、添字を各配列に加えておいて、ソートすればよい。添字をつけるのに、`each_with_index` というのを使う。

n 個ずつまとめてとる

`[1,2,3,4,5,6,7,8,9,10,11,12]` から、`[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]` とかとりたくなるとき `Array#each_slice` を使う。

度数分布

```
data = [1,2,1,1,3,8,6,1,3]
```

一番多いのはなんでしょう。`data.max_by{|x| data.count x}`

それぞれいくつあったでしょう。`data.group_by{|x|x}.map{|x,y|[x,y.size]}`[10]

にぶたん

二部探索を手軽にできる。

```
(0..1000).bsearch{|x| x**10 >= 2064377754059776}
```

探せるのは、条件を満たす最小値であるので、不等号を逆にするとバグる。

文字列どうしのバイナリ的 **XOR** をとりたい〜

よくある。前の章にも関連するけれど。

`'igohj'.bytes.zip([*1..5]).map{|x,y| x^y}` とかいつもやっていますがどうでしょう。

[10] もっといいのがありそう。

エンコーダ/エスケープ/ハッシュ

Base64

```
require 'base64'
Base64.encode64 'TSG'
Base64.decode64 'VFNH'
```

uuencode

実は pack テンプレート文字列にある。`'u'` が対応する文字。

URL escape

`URI.escape` というのがあるが、obsolete であるし、何がエンコードされるのかよくわからないので、`CGI.escape` を使うべき。

といっても、それでもエスケープが足りない時があるので、別にすべての文字をエスケープするんでいいんじゃないかと思っている。

```
string.bytes.map{|b| '%%02X'%b }.join
```

MD5,SHA

```
require 'digest/md5'
require 'digest/sha1'
require 'digest/sha2'
Digest::MD5.hexdigest('TSG')
Digest::SHA1.digest('TSG')
Digest::SHA512.hexdigest('TSG')
```

digest と hexdigest がある。他の言語同様で、digest にするとバイナリデータが、hexdigest にすると 16 進文字列でててくる。

SHA3 は標準ライブラリにはない。gem でも引っ張ってくること。

あとがき

おわり。もう少しつらつら書いても良いけれど、そろそろ双方飽きてきたし、提出の締切も過ぎてるし。この辺にする。

これが Ruby を普段からしている人が CTF を始めるきっかけになったら嬉しいと思いつつ、そんなことはなさそうとも思いつつ。

最後は EKOPARTY CTF 2016 Bleeding(Pwn50) の攻撃 Ruby スクリプトでも貼って終わる。

```
1   #encoding: BINARY
2   require 'socket'
3
4   sock = TCPSocket.new("4ff0eff1d46c1d74d152aaf36de6f2799020bdbc.ctf.site
        ", 50000)
5
6   data = 'hogefuga'.unpack("C*")
7   size = 10000
8   buf = [0x37333331,size,data].flatten.pack("VVC*")
9   key = 'deadbeefcafebabe'.scan(/../).map(&:hex)
10  # key = ['deadbeefcafebabe'].pack("H*").bytes でも可
11
12  encrypted = buf.bytes.each_with_index.map{|x,i| x^key[i%8]}.pack("C*")
13  sock.print encrypted + "\n"
14  puts sock.read
```

東方文花帖を AI に自動でクリアしてもらう話

satos

まえがき (あとがき)

小説は冒頭の書き出しが一番難しいといいますが、まさに今書き出しに苦労していて、結局一番最後にこの部分を書いています。最初は、人工知能は人間を単純労働から解放するだのなんだの書こうとしていましたが、そもそもそんな高尚な考えから作ろうと思ったのではなかったので没にしました。

僕が文花帖の AI を作ろうとしはじめたのは、過去のソースコードの更新日時によれば、約 3 年前のことだそうです。その当時に、参考文献 [3] の東方紅魔郷を自動でプレイする AI の動画を見て、自分も AI を作ろうと思い立ったのでした。だから、僕が AI を作ろうと思い立った理由は、動画を見て驚いたからだとか、AI が非人間的な動きで弾を避けているのが面白かったからだとか、それぐらいの単純で直接的なものだったのでしょう。

僕の駒場祭での展示が誰かに驚きを与えること、この文書が AI を作ろうとする誰かの助けになることを願っています。

解析

東方文花帖について

上海アリス幻樂団 [1] の製作した、シューティングゲームシリーズ『東方 Project』のうちの 1 つです。どんなゲームかは、インターネットでプレイ動画を調べるなり、もし駒場祭中なら展示されている AI のプレイ風景を見るなりすれば、多分一番手っ取り早く正確に分かると思います。

このゲームの AI を作るにあたって知っておくべきことは、まず、敵を攻撃して倒していくのではなく、敵から大量に飛んでくる弾を避ける類のシューティングゲームなので、避けるのを主目的として AI を作ること、また、文花帖は、東方シリーズの中でもシステムが特殊であり、敵の『写真』を撮れば [2] ステージクリアとなるゲームであること、この二つくらいだと思います。[3]

[1] http://www16.big.or.jp/~zun/top.html

[2] 主人公が新聞記者であり、敵の攻撃を避けつつスクープ写真を撮ることが目的のゲームです。敵の近くで、チャージの溜まった状態で撮影ボタンを押すと、敵の写真が撮れ、指定枚数以上を撮影するとステージクリア、というシステムです。

[3] あと、他の東方シリーズに比べて、1 ステージあたりの長さが 1 分程度と短く、デバッグしたり AI の強さを評価したりするのにかかる時間が短くて済む、という利点があって東方文花帖を選んだりしています。

解析の前に

東方文花帖は、Windows 上で動く x86 の exe ファイルです。もともとのゲームプログラムのソースコードはもちろん得られませんが、exe ファイルには動くための情報がしっかり残っています。そのため、バイナリを解析することにより、プログラムの内部挙動を知ることができます。

実際に解析を行うにはプログラムがどのように解釈されて動いているかを理解しないといけないので、機械語とかアセンブラとか PE ファイルの構造とか DLL とかマルチスレッドとか WindowsAPI の仕組みやそれを使ったプログラミングの方法とか、について知っているほうが解析する分には楽ですが、最近はインターネット上に資料がわんさか転がっているので、知らない状態で始めても、適宜必要な情報をググっていけば解析することができると思います。知識がなくてもインターネットがあればどうにかなります。[4]

解析に用いるツールについて

プログラムの解析法には、動的解析と静的解析の 2 種類の方法があります。動的解析は、プログラムを動かしつつ入力を試したりメモリをいじったりして、そのときの挙動や呼び出される外部関数などから内部の構造を探る方法です。静的解析は、プログラム中の文字列やリンクされる外部関数、逆アセンブリなどを調べることにより、プログラムの構造を調べる方法です。

例えば、以下で述べるもののうち、「IDA」はどちらかといえば静的解析用で、「うさみみハリケーン」「OllyDbg」「GDB」はどちらかといえば動的解析用です。実際の解析の際には、これらの方法をうまいこと組み合わせつつやっていきます。

また、既存のツールだけでは面倒なときは、デバッガなどを自作する [7] のもよいかと思います。

IDA

超高性能ディスアセンブラです。デコンパイル機能もついてます。おためし版が無料で手に入るので、とりあえず使ってみるとよいでしょう。64bit バイナリは有料版 [5] でないと解析できませんが、文花帖は 32bit なので無料版でも解析することができます。

うさみみハリケーン

プロセスメモリエディタ兼デバッガです。特定の値でメモリの検索をしたり、ハードウェアブレークポイントを仕掛けたりできるので便利です。

[4] まあ、実践をする前に基礎をある程度やっていたほうが効率が良いかとおもいますが。CTF のリバーシングと呼ばれる分野をやるとそのへんの知識が楽しみながら身につくのでよいです。

[5] 1000 ドルくらいします。ほしいのでだれかください。

OllyDbg , GDB

デバッガです。どちらもちょっとしたプログラムの解析には便利ですが、大物になると大変で、今回はあんまり役立ちませんでした。

挙動を把握する

main 関数からトップダウン式にやる方法と、個別の関数からボトムアップ式にやる方法とがあります。前者はすなわち、main 関数の挙動を調べ、その中で呼ばれている制御関数や描画関数の挙動を調べ... といった風に、全体の構造を一度に調べていく方法です。CTF の pwn 問で出るような本当に小さなバイナリなら上から調べて行っても十分すぐに全挙動を把握することができますが、ゲームの挙動となるとそうはいきません、年が暮れてしまいます。そこで後者の方法を取るわけです。キーボード入力の受け取りかたや弾と自機の当たり判定、ステージデータが内部でどのように処理されているか、など、要所の処理をしている関数を見つけることにより、ピンポイントで知りたい挙動を得ることができます。[6]

さて、問題はどうやってその要所の関数を見つけるかですが、これには様々な方法があります。以下では僕の使った方法を紹介していきます。

外部からの入力を得ている位置を得る

後々説明しますが、DLL インジェクションで乗っ取る API を探すために中の挙動の解析が必要になりました。具体的には、なにがしかの外部状況を読んでいる WindowsAPI があるので、それがどのように呼ばれているか調べる必要があります。理想的には、ゲームが 60fps なので、1 秒間に 60 回ほど呼ばれている API が望ましいです。

API を探すのは、例えば適当な PE ヘッダビューワーでヘッダを見るなり、IDA や OllyDbg の外部関数呼び出し一覧から調べるなりでできます。また、API モニターなどを使うとどれくらいの頻度で何が呼ばれているかを確認できます。

まず、当初はキーボード入力を調べるような、 GetKeystate などの API を探していました。見つかるには見つかったんですが、1 秒間に数回しか呼ばれていませんでした。よくよく考えると、キーボードの状態は WM_KEYDOWN メッセージとか、そういうのがメッセージループに送られた時にだけ調べられるので、外部からメッセージを送りつけなければ関数が呼ばれず、却下となりました。そこで別の手段として、 joyGetPosEx という API を乗っ取ることにしました。これはジョイパッド [7] の状態を調べるやつなのですが、ジョイパッドからメッセージループに送られるメッセージがないので、この API は毎フレームごとに呼ばれていました。また、ジョイパッドが刺さっているかどうか判定しているメモリ [8] をいじることにより、ジョイパッドが刺

[6] これは CTF のバイナリ問にも共通する考え方だと思います。バイナリを素早く解析するこつは、いかにバイナリを読まずに挙動を把握するか、にあると思います。

[7] 文花帖はジョイパッドにも対応しています

[8] joyGetPosEx を呼んでる付近のアセンブラを読めばどこをいじればよいか分かります。

さっていると誤認させ、ここから AI の入力をさせることができるようになりました。

自機と敵弾の位置を取得する

AI が弾の位置を知るには、メモリ上のどこに、どの形式で値が載っているのかを知る必要があります。また、弾と自機の当たり判定の計算方法も知っておいたほうがよいでしょう。

まず、先行研究 [4] によれば、東方地霊殿 [9] において、自機や弾の位置の内部表現は float 型だそうです。文花帖もおそらく同じようにしているでしょう。

ここで、うさみみプロセスエディタの変動検索機能を利用します。これは、メモリのスナップショットをとってその差分を比べ、特定の動きをしているアドレスのみを絞り込んで検索してくれる、というものです。例えば、自機を右に動かしたあとに、値の増加したメモリのみに絞り込み、自機を左に動かして減少したもののみに絞り込み... というのを繰り返していくと、自機の動きと同期してメモリ上の値が増減しているもののみにメモリのアドレスを絞り込むことができます。数回やっていれば、たかだか 1,2 か所程度に絞り込めます。[10] 自機の左右の動きと同期して値の増減しているメモリには、おそらく自機の x 座標の値が入っているはずなので、これで自機位置を取得することができました。これと同じようなことを行い、ある弾の座標データのメモリ上でのアドレスを得ます。[11]

その後、メモリブレークポイントを仕掛けて、そのメモリを読み込んでいる実行コードの該当部分を探します。[12] これは複数か所あったりするのですが、そのうち 1 か所は弾と自機の当たり判定をしているところだろう、と見当をつけて見ていくと、if 文の連鎖が見えたりします。[13] 全判定を潜り抜けるようなところにブレークポイントを仕掛けておき、その状態でゲームを数回プレイします。で、キャラクターが被弾した瞬間のみにブレークポイントに引っかかるなら、そこが当たり判定を行っているところだと想像がつきます。[14]

これで、当たり判定をしているところが分かりました。さらに、その周辺のアセンブラを頑張って読んでいくと、自機のデータがどこに格納されていてどんな風に当たり判定をしているかとか、

[9] 東方シリーズのひとつ

[10] 1 か所だけでないのは、たいてい、内部状態を持っているもともとのデータ以外にも、描画のためにコピーされたデータとかがあったりするためです。

[11] と、軽く書いてはいますが、こちらは自機位置ほど簡単ではないです。というのも、自機は自分で動きを制御できるのに対し、敵弾は勝手に動いていくので、絞り込むのが大変なのです。僕は、1-1 の蛇行して進む弾を利用することにより、絞り込みに成功しました。

[12] x86 のブレークポイントには、ソフトウェアブレークポイントとハードウェアブレークポイントがあります。ソフトウェアブレークポイントは、プログラム中のブレークしたい命令の部分を int3 という命令に書き換えることによって、そこを実行したときにデバッグイベントを発生させてブレークするものです。(ブレークポイントを仕掛けたあとに該当のコード部が書き換えられるとだめなので、シェルコードのデバッグではこれは使えません) ハードウェアブレークポイントは、x86 のデバッグレジスタという機能を用いて仕掛けるものです。対象アドレスの実行時以外にも、書き込み時や読み書き時にブレークさせる、といった設定をすることが可能なので、今回のような場合などに役立ちます。

[13] このへんは雰囲気で頑張ってください。例えば、IDA とかだと、if 文が連鎖しているところは階段状にブロックが並んでいるので、見た目でなんとなく分かります。他にも、for 文っぽい形とか、switch 文っぽい形とかがあります。慣れです。

[14] たとえば、その判定部を回避するようなパッチをあててやれば、弾が当たってもキャラがやられないように改造することもできるでしょう。

[15] 他の弾データがどのように格納されているか、[16] とかを知ることができます。また、弾の座標に書き込みをしているあたりを探すと、弾の座標を更新している部分が見つかり、そこを読んでいくと、弾の速度ベクトルが格納されている位置などが分かります。[17] 自機や敵機についても、同じような方法でデータを集めることができましたので、これらのデータをもとに、どうやって回避すればいいかを探索します。

DLL インジェクション

AI にゲームを自動プレイさせるには、AI がゲームの状態を把握できることと、AI がゲームに入力を伝えられることが必要です。

去年の駒場祭で僕がやった T-Rex Player では、状態の把握はゲーム画面のキャプチャを解析することによって、ゲームへの入力は SendMessage という WindowsAPI を用いることによって行いました。が、今回はどちらも同じ方法ではうまくいきませんでした。[18]

そこで、参考文献の [2] のスライドで行っている、DLL インジェクションという方法を用います。

原理

この方法の理解には、DLL を呼び出す方法や、PE ヘッダの IAT(インポートアドレステーブル)についての知識が必要になります。[19]

IAT の上書き

Windows には、DLL という実行時に動的にリンクされるライブラリがあります。このライブラリは、プログラムの実行時に PE が展開された後、その下に展開されるわけですが、このとき、本体から DLL への関数呼び出しを解決してやる必要があります。これを愚直にやろうとすると、プログラム中の call 命令全てのアドレスを調べて適宜書き換える必要があるので非常にロードに手間がかかります。そこで、DLL 上の関数アドレスを書き込むスペースを PE ヘッダ上に作ってお

[15]画面上にはさまざまな形の弾が飛んでいるように見えますが、文花帖の場合は、実際のデータは bullet と laser の 2 種類のみでした。bullet の当たり範囲は画面に水平な正方形でした。(このため、円形に見える弾では、真横のほうが角より避けやすかったりします) laser の当たり範囲は、矩形を適当な角度分傾けたものでした。それぞれ、当たり範囲内に自機が入ると当たったと判定されます。

[16]弾やレーザーのデータはリストに繋がれて管理されているので、リストを走査することにより全ての弾とレーザーのデータを取得することができました。

[17]弾のデータは、弾の中心位置と移動速度、当たり判定の大きさなどが得られました。レーザーのデータは、レーザーの中心位置と角度、縦横の当たり範囲と中心座標の移動速度、角度の角速度などが得られました。

[18]キャプチャを頼りにする場合、画面から自機や敵機や弾の位置を識別する必要がありますが、背景がわりと騒がしいので大変そうです。また、 SendMessage で WM_KEYDOWN を送りつける方法ではキー入力が認識されませんでした。

[19]理解せずに使おうとするとうまいこと動かなかったときにデバッグのしようがないので、面倒でも一度学んでおきましょう。

き、呼び出す際にはそこのアドレスを参照して呼ぶことにすれば、書き換えるのはそのスペース分だけで済みます。この関数アドレスを書き込んでおく部分のことを IAT(Import Address Table) と言います。[20]

で、ここを書き換えてやると、プログラムに介入することができます。たとえば、GetKeyboardState 関数のアドレスを自作関数のアドレスで上書きしてやると、プログラム中で GetKeyboardState が呼び出されるたびに、代わりに自作関数が呼び出されることになります。また、自作関数の返り値をうまいこといじってやれば、あたかもキーボードが押されているかのようにプログラムに錯覚させることができます。

DLL の注入

IAT を上書きしても、自作関数のデータが対象プロセスと同じメモリ上に載っていないと動いてくれません。そこで、CreateRemoteThread という WindwsAPI を用います。これは外部から対象プロセス内に新たにスレッドを立てる、というものです。これを使って LoadLibrary に自作 DLL のパスを渡したものを呼んでやると、プロセスがあたかも自分から DLL をロードしたかのように思って、自作の DLL を読み込んでくれます。DLL は読み込まれる際に DllMain という関数が呼ばれることになっているので、対象プロセスに自分のコードを実行させることができます。あとは、DllMain 内で IAT を上書きしてやればよいですね。

やりかた

具体的な方法は、たとえば参考文献の [6] などに載っています。ほかにもネット上にいろいろ資料が転がっているので参考にするとよいでしょう。

今回の場合

今回は、前述の JoyGetposEX を上書きしました。返り値としてジョイパッドの入力を偽装してやると、あたかもジョイパッドが操作されているかのようにすることができます。[21] 更に、対象プログラムと同一プロセスなので、いちいち ReadProcessMemory [22] することなくゲーム内部のメモリを読むことができます。

以下、自分のコードより IAT の上書きを行う部分の抜粋です。[23] 上書きするべきアドレスは、ImageDirectoryEntryToData とかを使って真面目に調べてもよいですが、今回は IAT の目標アドレスをそのままベタ書きしています。[24]

[20] Linux の実行ファイルである ELF にも GOT という IAT とだいたい同じような目的のものがあります。IAT を書き換えるのは GOT overwrite のようなものです。

[21] 手元にジョイパッドがあれば、その挙動を動的解析することにより簡単に入力が真似できますが、無かったので、MSDN の JOYINFO structure について調べたりして入力を再現しました。 Xpos , Ypos の値域を初期化しないとうまく動かないのでやるときに気を付けてください。

[22] これを使うと別に injection せずともメモリを読むことができますが、いかんせん速度が重いのです。

[23] 自作 DLL をロードさせる部分は、参考文献 [1] に載っていたものを使っているのでそちらを参照してみてください。

[24] たいていは固定アドレスであり、ベタ書きの方が楽です。

```
void WINAPI attach(){
        DWORD dummy;
        VirtualProtect(gotpos, sizeof(gotpos), PAGE_EXECUTE_READWRITE, &dummy);
        /* ここで、書き込むとこのプロテクトを外す。 */
        mempo = (Jgpex)(*gotpos);
        (*gotpos) = (PROC)Joygetposex_wrapper;
        VirtualProtect(gotpos, sizeof(gotpos), PAGE_EXECUTE_READ, &dummy);
        /* ここで、書き込むとこのプロテクトを戻す。 */
        /* 後略 */
}
void WINAPI detach_joygetposex(){
        DWORD dummy;
        VirtualProtect(gotpos, sizeof(gotpos), PAGE_EXECUTE_READWRITE, &dummy);
        /* プロテクトを外す。 */
        (*gotpos) = (PROC)mempo;
        SendMessage(debhwnd,WM_CLOSE,0,0);
        Sleep(3000);
        /* ここにスリープを入れないと、自分のラッパ関数が呼ばれている最中に */
        /* DLL がデタッチされようとしてプログラムが落ちる */
        VirtualProtect(gotpos, sizeof(gotpos), PAGE_EXECUTE_READ, &dummy);
        /* プロテクトをはめる。 */
}
```

AI を作る

さて、敵機、弾、自機の位置とそれらの移動速度を得ることができました。これらを用いて弾を回避して写真を撮る AI を作ります。

現状の AI

弾がそのまま直線運動すると仮定して盤面予測を行っています。自機が被弾しないように気をつけつつ、避けているだけではいけないので適当にボスの方向に動きつつ、といった感じの AI にしています。個人的なイメージはビームサーチですね。

以下は本質部分から適当に省略しつつ抜粋したものです。

```
void getd(revdata d,int& ry,int& rx,int& button){
        ry=rx=button=0;
        /* ステージによっては、ここに開始時にランダムに動くような処理が入っていたりする */
        d.select(50.0);
        d.step();
        if(!d.isdie()){
                /* もし、次フレームで被弾せず、チャージが溜まっているなら撮影する */
                if(d.my.fil>=1.0){
                        if(abs(d.my.p.dist(d.bos.p)) < 140.0){
                                button |= 1;
                        }
```

```
                }
        }
        typedef pair<mp,fpos> mfp;
        vector<mfp> vs;
        vs.push_back(mfp(mp(-2,-2),d.my.p));
        /* beam search みたいな感じで、死なないパターンを列挙していく。 */
        rep(i,20){
                vector<mfp> tv;
                for(mfp pa : vs){
                        vector<mfp> tps = enummypos(d,pa.sec);
                        for(mfp pb : tps){
                                d.my.p = pb.sec;
                                d.my.normalize();
                                if(d.isdie())continue;
                                if(pa.fir.fir>=-1)pb.fir = pa.fir;
                                pb.sec = d.my.p;
                                tv.push_back(pb);
                        }
                }
                swap(tv,vs);
                if(vs.size()>=500)break;
                d.step();
        }
        /* 後略 */
        /* あとは、 vs に n 手先読みした場合の初手と最終距離が入っているので、 */
        /* 25 パターンくらい生存しているような初手のうち、 */
        /* ボスにある程度の距離まで近づいていくような初手を選ぶ。 */
}
```

現状

現状、前述のアルゴリズムを用いると、全 85 ステージ中 49 ステージをクリアできます。半分は超えてますが、Ex を出すには 66 ステージクリアしないといけないのでまだまだ工夫しないといけません。

人間が苦手だが自分の **AI** の得意な面

- 9-6「金閣寺の一枚天井」
  弾がある程度ランダムに、それなりのスピードで大量に上から降ってくるので、その間を避ける必要があります。人間がやる場合はなかなか集中力が要りますが、AI は関係なくクリアしてくれます。
- 8-5「八千万枚護摩」
  弾が横並びで飛んできて、一見回り込まないと避けきれないように見えるのですが、実は少しだけ弾の間に隙間があり、AI はそこをするりと通り抜けて楽々クリアすることができます。

物量主義的な弾幕に対しては、AI はそれなりに上手く避けきることができます。

自分の **AI** の苦手な面

- 5-4「フロギスティックレイン」
  8-5 と違って弾の間に隙間がなく、チャージを素早くして撮影し、弾を消して道を作らないといけないので、だめです。[25]
- 8-2「死出の誘蛾灯」
  弾がそれなりにきつい角度で曲がるので、AI の予測が外れて被弾してしまいます。[26]
- 9-4「エイジャの赤石」
  いきなり初見殺しのビームが飛んでくるので、ビーム生成関数のあたりを読んでやらないといけません。[27]

全体的に、突然敵弾が飛んでくるか、高速チャージしないといけないか、弾の密度が高くかつ弾が微妙に曲がって飛んでくる、ような場合に失敗している感じです。

今後の展望

未解析の部分[28] はまだまだあり、するべき課題は山積みです。例えば、突然発生する弾や初見殺し的なスピードの弾に対しては、現状の AI では対処しようがないですが、内部でどのように弾を発生させて動かしているかをより正確に解析すれば、避けられるようになるかもしれません。また、低速移動や高速チャージなどの操作も組み入れていないので、まだまだ実装する必要があります。個々の盤面に対してアドホックに対応するのではなく、強化学習なども取り入れて、もっと汎用的に、自分から学習してくれるような AI を組んでみるのもよいでしょう。

「我々は、地球を探検するには遅すぎ、宇宙を探検するには早すぎる時代に生まれてしまった」という言い回しがありますが、計算機の中にはまだまだ未知の領域が広がっています。

参考文献

東方の自動プレイについて

全て、この記事を書いてる時点 (2016/11/16) でのものです。

[25] AI に高速チャージを組み込まないといけないのだけれどまだやってないです。
[26] 弾の速度ベクトルをいじっている部分があるはずなので、そこを読む必要があるのですが、まだ読んでいません。
[27] まだ読めてないです。できれば逆アセンブラ結果を読まずにことを済ませたいです。
[28] その中にはあまり触れない方がよろしい部分もあります。たとえば、解析していく中でデータの暗号化を外している部分なども見つかるわけですが、今回の目的はデータの抽出ではないので読む必要はなく、またなにがしかの法に触れないとも限らないので無闇矢鱈と解析するものでもないでしょう。

1. http://d.hatena.ne.jp/aki33524/20131110/1384031374 , 『東方の当たり判定を描画してみるよ！ - aki33524 の日記』
2. http://www.slideshare.net/aki33524/ai-32089294 , 『東方紅魔郷 AI』
東方紅魔郷の自動プレイについてのブログやスライドです。これを参考に DLL インジェクションを行いました。
3. http://www.nicovideo.jp/watch/sm23043946 , 『【AI リプレイ】紅魔郷 Ex ノーショットノーボムノーミスクリア 1/2 - ニコニコ動画』
[1] の AI が実際にプレイしている動画です。
4. http://www.usamimi.info/~ide/programe/touhouai/report.pdf , 『東方地霊殿の自動プレイプログラムの作成』
東方地霊殿の自動プレイについての pdf です。内部でのデータの持たれ方などについて参考にしました。

解析とか **DLL** インジェクションとかについて

5. 新井 悠、岩村 誠、川古谷 裕平、青木 一史、星澤 裕二 著 ,『アナライジング・マルウェア』, オライリー・ジャパン , 2010 , https://www.oreilly.co.jp/books/9784873114552/
DLL インジェクションのやりかたについての記述があります。
6. 姜 秉卓 著/金 凡峻 訳/金 輝剛 監修 , 『リバースエンジニアリングバイブル』 , インプレスジャパン , 2013 , http://book.impress.co.jp/books/1113101030
Windows バイナリを解析するときのコツについて書かれています。
7. Digital Travesia 管理人 うさぴょん 著 , 『デバッガによる x86 プログラム解析入門【x64 対応版】』, 秀和システム , 2014 , http://www.shuwasystem.co.jp/products/7980html/4205.html
デバッガを自作する際に参考にしました。

Green Hackenbush の必勝法について

dai

はじめに

1 年の dai です。二人対戦ゲーム ‘Green Hackenbush‘とその必勝法を紹介します。
ちなみに、この文章は最後に載せてある論文を適当に和訳しながら説明しているだけなので、厳密な議論はそっちを読んだほうが良いと思います。

Green Hackenbush

場には枝分かれした木のようなものが生えており、二人のプレイヤーは交互に枝を切っていきます。このとき、地面から離れてしまった（孤立した）枝は一緒に消えてしまいます。これを繰り返し、切る枝がなくなってしまったほうが負けとなります。（下図参照）

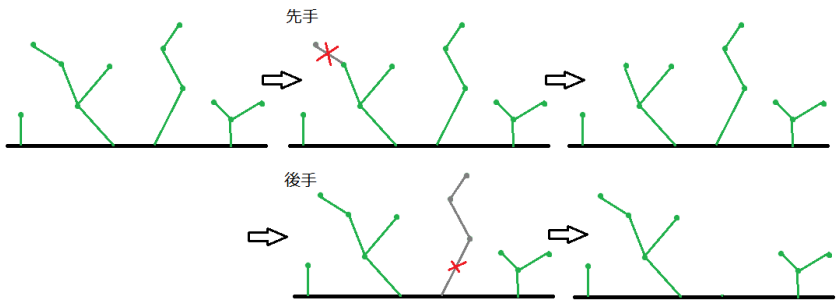


図 6.1: ゲームの流れ

これは二人零和有限確定完全情報ゲームの一種であり、このようなゲームは引き分けがない場合必勝法が存在することが知られています。そしてこの Green Hackenbush の必勝法は、人の頭の中でもなんとか計算できるものになっています。

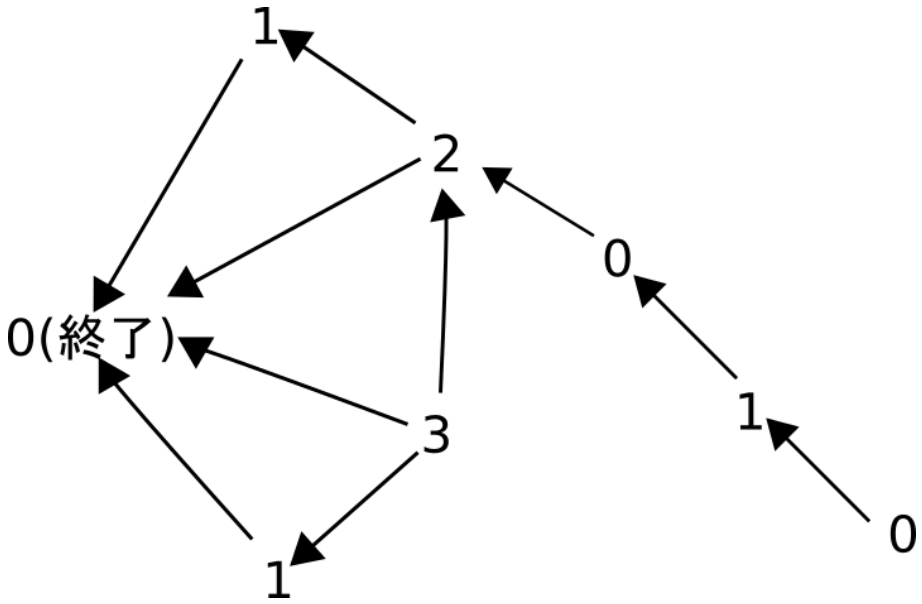


図 6.2: Grundy 数の推移の例。

三山崩しの必勝法

本題に入る前にまずより簡単な「三山崩し」について考えます。このゲームでは、場にコインが三つの山に分けて積んであり、プレイヤーはどれか一つの山を選択してそこから好きなだけ取り去ります。そして最後にコインが取れなくなったほうが負けになります。このゲームにももちろん必勝法が存在し、プレイヤーは「Grundy 数」と呼ばれる数が 0 になるようにコインを取ることによって絶対に負けることがなくなります。
この Grundy 数とは、盤面の状態を非負整数で表したもので、最後の盤面（今回は机の上に何も載っていない状態）を 0 とします。そしてほかの盤面は、そこから変化しうる盤面の Grundy 数を除く最小の整数をその盤面自身の Grundy 数とします。こうすることで、各盤面は必ず別の Grundy 数に変化し、そしてその盤面より少ない Grundy 数をもつ盤面へ移動できることが保証されます。よって、Grundy 数を 0 にして相手に番を渡すことで、最後のコインを自分がとれることも保証されます。

そしてこのゲームにおいて Grundy 数は各山のコイン数の排他的論理和（xor）を取った数なのですが、なぜ山ごとで xor を取るのかは全て省略して（自分自身よく分かっていません。ごめんなさい！ 小さな例から始めて、数学的帰納法を利用して任意の山の証明をするらしいのですが……)、とにかくこのようにすると必ず勝てることが、実際にやってみるとわかります。

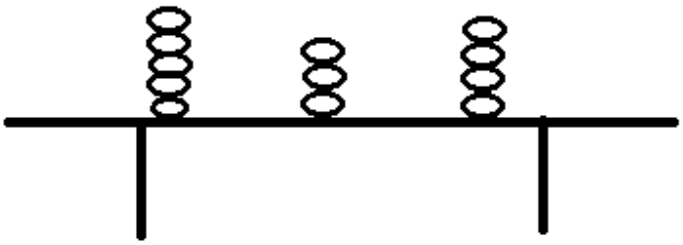


図 6.3: 三山崩し。この場合、中央の山から二つ取り一つにすることで Grundy 数を 0 にできる。（5 xor 1 xor 4 == 0）

Green Hackenbush の必勝法

この必勝法が、似たゲームであるこの Green Hackenbush に応用することができます。このゲームにも Grundy 数が存在し、これを 0 にすることで勝つことができます。しかしこのゲームでは単に山ごとで xor をとるだけではうまくいきません。
では、Green Hackenbush 特有の形状における Grundy 数の求め方を考えてみましょう。

枝分かれ

まず、次の図は、先に挙げた三山崩しの図と同じ状態であることがわかります。

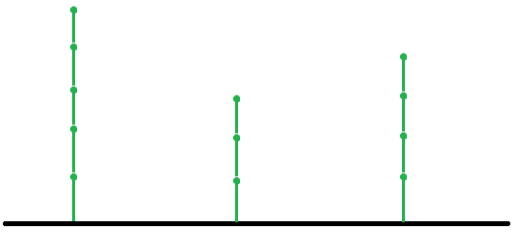
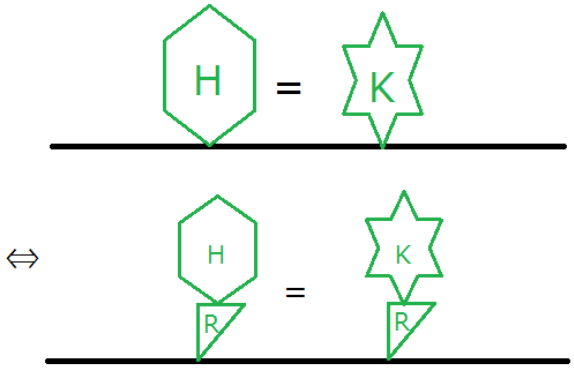


図 6.4: この場合も中央の枝を一本にすればよい

では枝分かれはどうなるでしょう。これを考える前に、次の定理について考えてみます。

定理 **1 (the Colon Rule)** グラフ H,K があり、H=K(Grundy 数が等しい。以降単に「等しい」と表す) のとき、地面に接している部分に新たにグラフ R を追加したグラフ H+R と K+R は等しい。



これを示すために、H+R と K+R のグラフを並べて一つのゲームとしてみます。このとき $H+R=K+R$ ならば、xor より場全体の Grundy 数は 0（後手必勝形）となるはずです。そして後手番は次のような戦略をとります。

1. もし先手が R に属する枝を切った場合、後手はもう一方の R の同じ枝を切る。
2. もし先手が H(または K) に属する枝を切った場合、(H xor K=0 より)H' xor K' が 0 になるように枝を切る。このとき切る枝は H または K に属している。

このようにすることで、先手番では常に H'+R' と K'+R' が並ぶため、場の枝をすべて取り去り対称的なグラフにすることができなくなります。つまり先手は絶対勝つことができないため、必然的に後手が勝つことになります。以上より、(H+R)xor(K+R) のグラフは後手必勝 ($\Leftrightarrow Grundy=0 \Leftrightarrow H+R=K+R$) であることが示せました。
「同じものを足しても不変」なら「同じものを引いても不変」なので、これを利用することで枝分かれの Grundy 数を比較によって求めることができます。

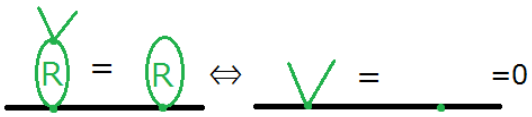


図 6.5: 比較によって Grundy 数を求める

この図を見れば、枝分かれは地面上の山同士の計算と同様に xor で処理できることがわかります。

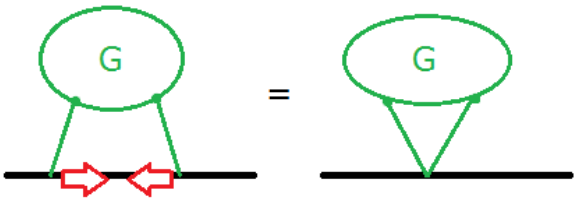
ループ

Green Hackenbush の特徴としてもう一つ、ループがあります。これも枝分かれと同様変形前と変形後の xor が 0 になることを示すという方法を用います。ここで次のような操作を考えます。

定理 **2 (the Fusion Principle)** ループ上の節は結合してよい。このとき Grundy 数は変化しない

この定理を証明するために、結合前と結合後のグラフを並べたゲームが後手必勝である（Grundy 数が 0 である）ことを示すのですが、ループには様々な種類があります。

地面上の節結合



これは見たまんまですね。くっつけてもグラフの振る舞いは全く変わりません。

空中の節結合

問題は地表以外で節を結合する場合です。ここで、結合する節の一方から他方へ通る道は複数通りありますが、ゲーム中でうまく枝を切ることで一番シンプルなループ（道が 2 通り）の問題に置き換えることができます。
例えば、下の図では、R=R’ を示すためにはこれが後手必勝であることを示さなければならないのですが、先手が切った枝に対応する枝を切ることで、1 巡後には 2 通りの道しかないループに変えることができます。つまり、その先が後手必勝であることが示せたら、元の図形も後手必勝であることが言えます。

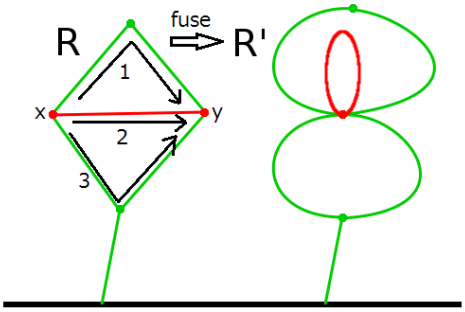


図 6.6: 結合前（左）と結合後（右)、この場合 x から y へ 3 通りの道がある。

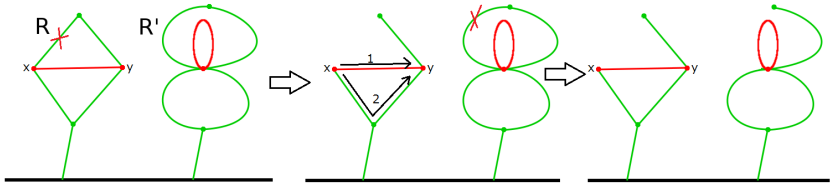


図 6.7: 後手はこのような戦略をとる。一番右の全体の Grundy 数が 0 であればよい。

もちろんこれは 4 通り以上の場合でも同じことを繰り返して 2 通りのループに帰着できます。

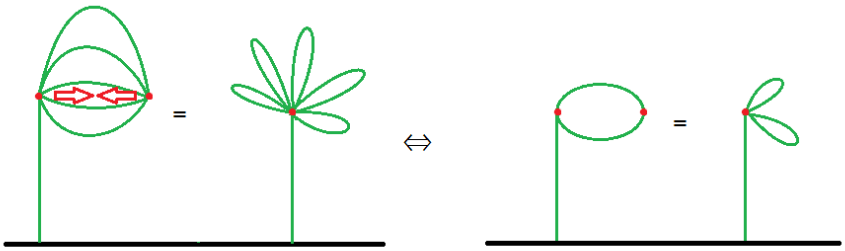


図 6.8: どんなに複雑でも良い。

そして単純になったループを **Colon Rule**(定理 1) で地面上に持ってきて比較して終わりです。... と行きたいところですが、ここでまた面倒事が起こります。

地上 ⇔ 空中の節結合

こいつを行ってよいのかまだ示していません。これさえ証明すれば、自動的に前述の節結合がすべて正当だと言うことができます。

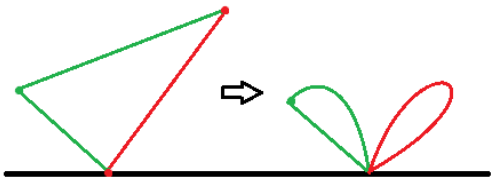


図 6.9: これは可能か?

ループ上（これを「橋」と呼びます）には枝分かれやほかのループがくっついている可能性がありますが、それらはすでに述べたような方法で同じ Grundy 数を持った一本の「糸」に置き換えることができます。そしてループは、単純なもので試すとわかるのですが、すべての節を結合すると、ループを構成する枝が偶数個なら Grundy 数 0、奇数個なら Grundy 数 1 になります。今までと同様に結合前と結合後を並べます。
まず橋が偶数本の場合を考えます。

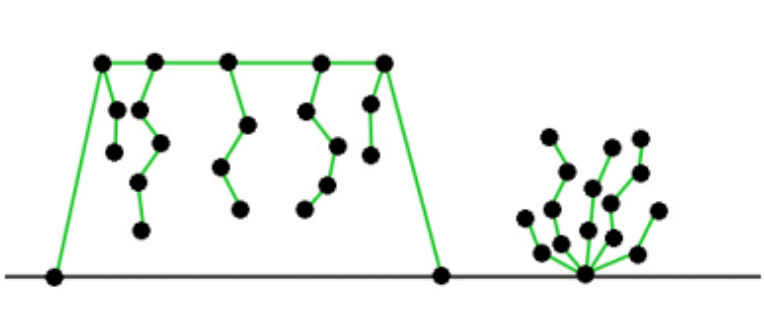


図 6.10: 論文より引用。結合前（左）と結合後（右）。左にぶら下がっているものが「糸」である。これらが右のグラフに対応する。

このゲームでもし先手が「糸」を切った場合、後手はそれに対応するものを切るだけです。問題は「橋」を先手が切った場合です。この瞬間ループが切れ、枝分かれのみになるので Grundy 数がすぐ求まるのですが、これが 0 でないこと（後手に回ってきたとき Grundy は 0 以外でなければならない）を証明するにはどうすればよいでしょうか。ここでもう一つ定理を挙げます。

定理 **3**　少なくともループの存在しないグラフにおいて、Grundy 数の奇偶と枝の本数の奇偶は一致する。

枝を一つずつ地上から積むことを考えます。枝が 0,1 本のときは自明に Grundy 数は 0,1 です。次の枝からは「どこかから枝分かれさせる」か「どこかの先端に積む」の 2 通り考えられますが、どちらも Grundy 数の奇偶を反転させていることがわかります。よってこの定理は正しいと証明されました。

ではこれを利用して先ほどの議論を続けましょう。橋の本数は偶数であったので、先手がその枝を切った今、橋だったものは合計で奇数本のはずです。糸に対応する枝はピッタリ同じ本数が左右に存在するので、グラフ全体では枝が奇数本、すなわち Grundy 数は 0 でないことがわかりました。

最後に橋が奇数本の場合です。橋を結合すると Grundy 数は 1 になるので、図は次のようになります。

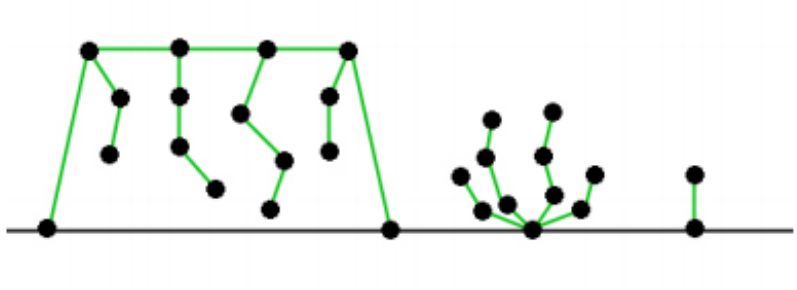


図 6.11: 論文より引用。一番右の一本が橋だった物。

大体は先ほどと同じですが、先手が橋を切ったときこんどは偶数本になってしまいます。しかし今回は右に一本枝が立っているので、結局全体の枝は奇数本であるため、後手番において Grundy 数は 0 でないことがわかります。これでループを結合しても Grundy 数が変わらないことが証明されました。

以上で Green Hackenbush に関する定理は終わりです。実際に対戦するときは、まずループを結合し枝分かれのみのグラフにした後、xor を用いて Grundy 数を求めることによって、どんなグラフでもそのグラフが後手必勝か、次にどこの枝を切ればよいかを判断することができます。

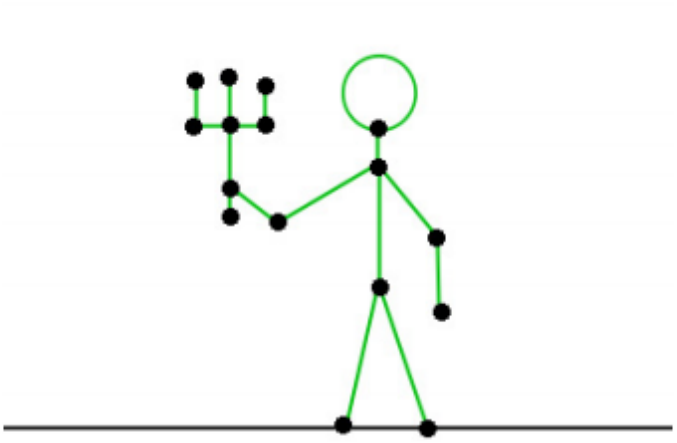


図 6.12: 論文より引用。一見複雑だが、結局は枝分かれとループの集合体である。

おわりに

このゲームについて日本語の記述があまりなかったので、勉強ついでに今回このテーマで書くことにしました。このゲームを基にした問題がプログラミングコンテストでたまに出るそうなので、知っていると役に立つかもしれません。ちなみにこのゲームは Hackenbush と呼ばれるものの亜種で、ほかに Red-Bkue Hackenbush や、R-G-B Hackenbush などがあります。下の論文ではそれらのゲームにも触れているので、興味があったら読んでみてください。

参考文献

Padraic Bartlett, ‘A Short Guide to Hackenbush.‘ *VIGRE REU 2006*
Available at: http://www.link.cs.cmu.edu/15859-s11/notes/Hackenbush.pdf

編集後記

★ 直前でエラーが大量に発生するとは思ってもいませんでした…
★ なんとか完成できてよかった…

理論科学グループ　部報 第 **312** 号

2016 年 11 月 25 日　発行
発行者　佐藤聡太
編集者　秋本慎弥
発行所　理論科学グループ
〒 153–0041　東京都目黒区駒場 3–8–1
東京大学教養学部内学生会館 313B
Telephone: 03–5454–4343

理論科学グループ部報 第 **312** 号
— 駒場祭パンフレット号 —
2016 年 **11** 月 **25** 日

THEORETICAL SCIENCE GROUP