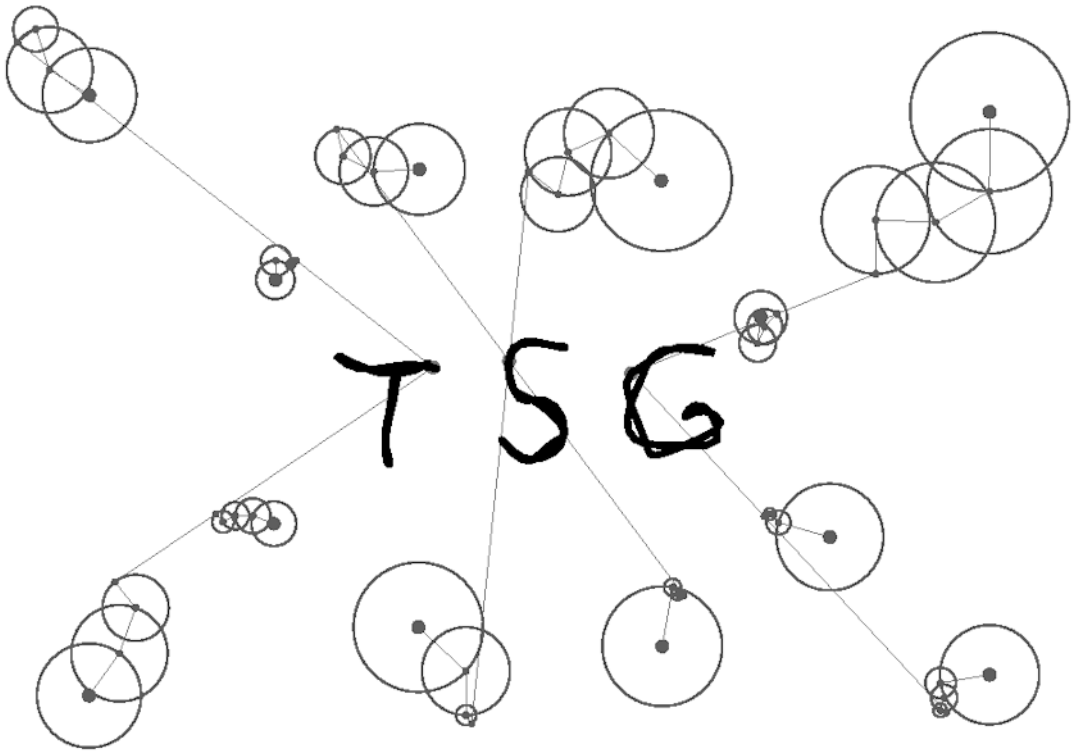


Theoretical Science Group

理論科学グループ

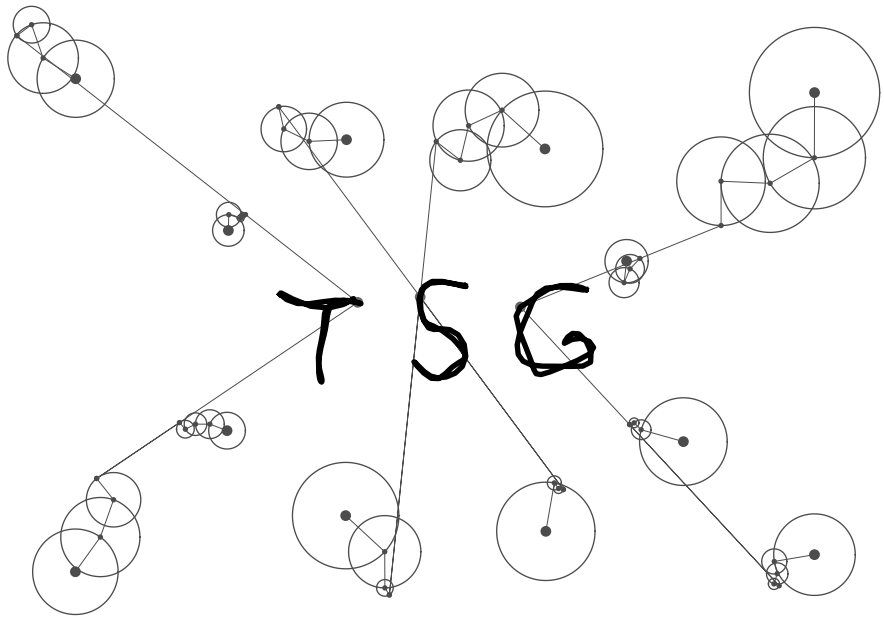
部報 313 号
— 駒場祭パンフレット号 —

目　次

展示企画

円運動だけで文字を書く

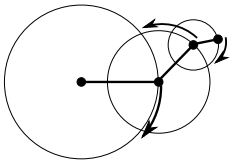
dai



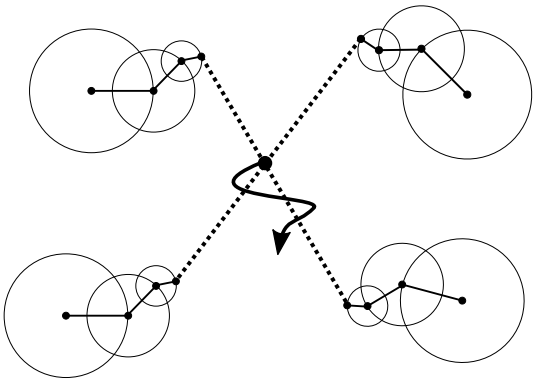
日本語では説明しにくいので図を載せました。駒場祭では動いている様子を展示します。

これは何？

まず固定された点の周りを小さな点が回っていて、さらにその点の周りを別の点が…を繰り返します。



この単位をユニットと呼ぶと、2 つずつユニットの一番外の点を結び直線をつくります。さらにこの直線を 2 つ用意して交点を取れば、その点は複雑な動きをします。



この複雑な動きを使って文字を書こうとしたのが冒頭の図です。

作り方

各ユニットにおいて、(x_0, y_0) にいる恒星 (一番内側の動かない点) の周りをまわる点 (惑星) の位置 (x_1, y_1) は時刻 t に対し次のように動きます。

$$x_1 + y_1 i = (x_0 + y_0 i) + r_0 e^{i(\omega_0 t + \phi_0)}$$

さらにこの周りをまわる衛星の位置 (x_2, y_2) は

$$\begin{aligned} x_2 + y_2 i &= (x_1 + y_1 i) + r_1 e^{i(\omega_1 t + \phi_1)} \\ &= (x_0 + y_0 i) + r_0 e^{i(\omega_0 t + \phi_0)} + r_1 e^{i(\omega_1 t + \phi_1)} \end{aligned}$$

つまりユニットの点の動きは円運動の線形結合で表せます。

直線を作って交点を…とやるともう少し複雑な式になりますが、一応 t について微分できる形です。そこで目的の文字からのずれを誤差として、chainer を用いて円の半径や位置や点の角速度などを最適化しました。chainer が偏微分やもろもろをやってくれるので、かなり負担は軽かったです。(やり方が悪いと全然学習してくれなくて困りましたが…)

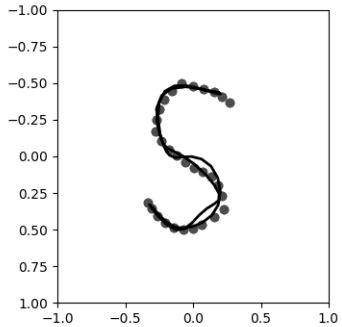


図 1.1: お手本の文字 (点の集まり) と最適化された点の軌道（曲線）

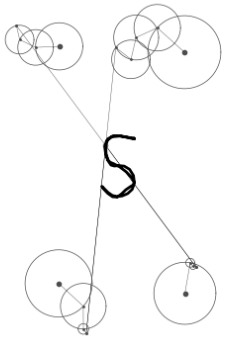


図 1.2: 学習後のユニットと交点が書く軌道

各ユニットで使う円の数を 4 つとすると、単純なアルファベット (S とか) なら 2,30 秒で最適化ができました。文字に折れ線や枝分かれがあると少してこずりますが、それでも 2,3 分回せばよいものが得られました。Adam はすごいなあ。

苦労した点

初めのころはあまりゴチャゴチャしても困ると思い、各ユニットが持つ円の数は 3(つまり衛星の衛星まで) に制限していました。しかしこれだと全然学習しませんでした (右図)。長時間学習しても改善しそうな雰囲気ではなかったので、少ない円で最適化するのはあきらめました。

そこで円の半径に L1 正則化をかけて、学習中に円の数を減らす (一部の半径を小さくして見えなくする) 方針を取りました。これはそれなりに作用しました。

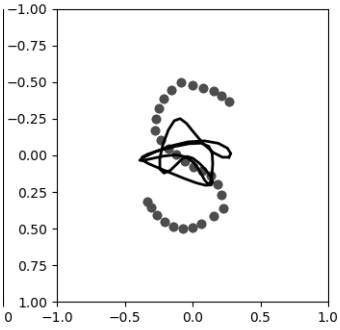


図 1.3: うーん？？？

おわりに

任意の文字を学習できるようにしたいです。あと半径等を連続的に変化させれば文字のモーフィングみたいなことができるのかな？ どうなるか試してみたいです。

Octas

kurgm

はじめに

2 年のくろごま (@kurgm) です。駒場祭では、以前作った「octas」という対戦型ボードゲームを展示します（予定)。これは僕が高校生の頃に友達と紙の上で遊んでいたゲームをパソコンやスマホなどで遊べるようにしたものです。夏休みに開催された TSG の開発合宿で gasin さんが AI を書いてくれたため、AI と対戦できるようになったのでその成果を展示します。

ゲームのルール

octas のルールを図 2.1 に示します。

序盤は両者がゴールを目指し合いジグザグな軌跡を描く状態が続きますが、盤面がある程度複雑になってくると、図中のルール 5 を使って一気に盤面を大きく移動することができるようになります。自分のゴールに近づくルートを探しながら、相手のルートをどう塞ぐかを考える必要があるのがこのゲームの面白いところだと思います。

AI の戦略

現時点の AI は、盤上の点を評価する関数（自分のゴールに近い点は値が高く、相手のゴールに近い点は値が低い）を用意して、4 手[1]先まで全探索をして最も評価値が高い点に到達できるルートを選んでいます。ただし、4 手読みは盤面によってはとりえる手の数が膨大になり全探索が終わらないことがあるため、10 秒経っても探索が終わらない場合は探索を中断して 2 手読みで探索をやり直すようになっています。

今後の課題としては、全探索をやめて計算量が小さい手法をとるとか、評価関数が盤面の状況を考慮するようにするとかを考えています。とはいっても、現時点ですでに AI が強すぎて勝てないのですが……。

[1]ルール 5 に拠って一度に複数回移動する場合、複数回の移動をまとめて 1 手として数えます。

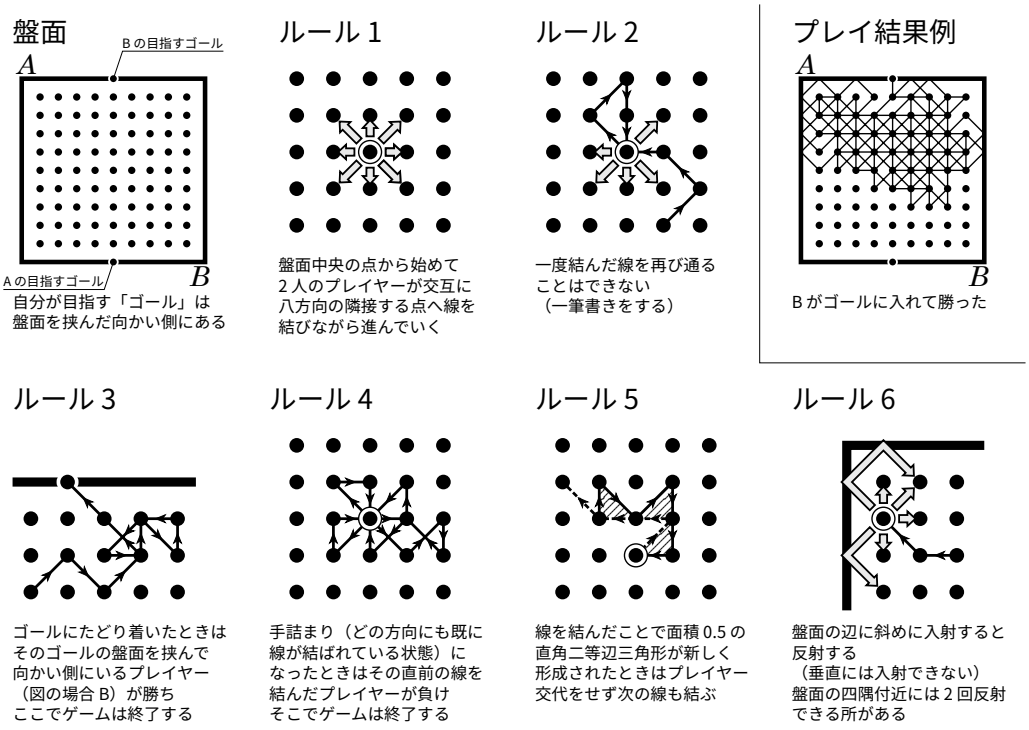


図 2.1: octas のルール

おわりに

octas は、パソコンやスマホなどのブラウザで `http://sig.tsg.ne.jp/octas/` にアクセスすれば遊ぶことができます。ただし、動作確認は Google Chrome でしか行っていないので、それ以外のブラウザだと正しく動かないかもしれません。

また開発は `https://github.com/tsg-ut/octas/` でオープンソースでおこなっています。この原稿を書いている時点では、画面に勝ち負けの表示が出ない、世界で同時に 1 人しかプレーできないなど課題が山積しているので、しあさって（！）の駒場祭に向けてこれから頑張って実装していきたいと思います。(｀・ω・´)

MNEMO Battler

moratorium08

展示しているゲームについて簡単な解説をします。ゲームは題して「MNEMO Battler」といい、TSG が作った MNEMO というゲームに対戦要素を加えたものになっています。まずはこのゲームの背景を簡単に説明します。

背景

既存のゲームとして、与えられたいくつかの数字から目標となる数字を四則演算を用いて作るいわゆる「四則」というゲームがあります。MNEMO Battler のイメージをわかりやすくするために、有名ですが、このゲームについて少し説明します。

このゲームは、まず最初にいくつかの数字が与えられます（ここではまず 4 つ与えられるとしましょう)。この 4 つの数字を一度ずつ使い、四則演算「足し算」「引き算」「掛け算」「割り算」を組み合わせて、目標となるある値（ここでは 10 としましょう）になるような計算方法を回答するというゲームです。

具体例を見て確認します。例えば、3,4,5,6 という 4 つの数字を使って 10 を作ることを目指します。この例だと例えば「5 - 4 + 6 + 3」が正解になります。数字には重複もありえて、例えば「1,2,2,3」という場合は、「1 * 2 * (2 + 3)」が正解になります。

もう一つ元となったゲームがあります。それが、昨年から今年にかけて TSG の人々が開発を行なっている「MNEMO」というゲームです。「MNEMO」は、プログラミング体験パズルゲームで、パズルのように演算子ブロックを組み合わせることで、計算を行い、正しい回答を導くゲームです。詳しい内容は、この会場でも展示が為されていると思うので、ぜひそちらも遊んでみて確認して下さい。

MNEMO Battler は、この「四則」というゲームと「MNEMO」というゲームの二つから派生した、「パズル型の四則」であると言えます。

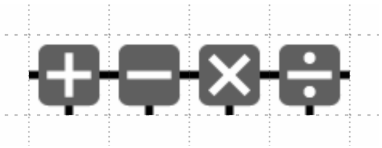
ゲーム説明

用語の説明

以下のゲーム説明に出てくる用語を簡単に説明します。

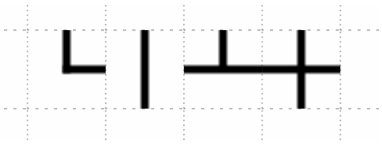
演算子ブロック

「+」、「-」、「x」、「/」のいずれかの演算が可能なブロックです。二つの値を受け取り、その結果を返します



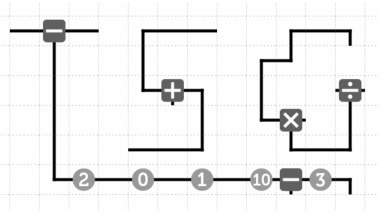
ワイヤー

設置された演算子ブロックをつなぎ合わせるものです。ワイヤーにはいくつかの種類があり、分岐するワイヤーは数が二つにコピーされることに注意が必要です



回路

演算子ブロックとワイヤーを合わせた全体で何らかの演算を行う集まりのことです



ゲームの概要

このゲームは対戦ゲームです。プレイヤーは二人いて、ターン制でゲームが進行します。

ゲームの開始時に、入力となる 4 つの数字と、お互いのプレイヤーがそれぞれ目指すべき二つの数字が与えられます。各プレイヤーは回路がこの数字を計算することができるように組み上げていくことになります。

ゲームのターンでの操作

次に、プレイヤーがそれぞれ回ってくるターンで何ができるかについて説明します。各プレイヤーは、自分のターンで、次の二つの操作のうちどちらかを選び行うことができます

1. ボードの上に演算子ブロックを一つ選んで配置する

2. 回路が完成したことを宣言し、ワイヤーで演算子ブロックを配置して、実際に回路が正しく動くことを示す

1,2 それぞれ制限時間内に終えることができなければ、ブロックがランダムに配置されターンが相手にうつります

その他

このゲームの開発は夏休みの合宿において行われましたが、ゲーム実装自体は、hakatashi さんが行い、簡単な AI に関する実装を僕が積み残していましたが、多分駒場祭当日にはできていると思います。

このゲーム、お互いが紳士的に打ち合えば（ゲームを壊す目的でゲームをしなければ)、予想外の形での回路や超絶技巧的回路が作れたりするので、楽しいです。まだゲームとしては発展途上のような部分もあるため、もっと改善したら良さそうな部分があれば、気軽に声をかけてくださるとうれしいです。

Taxi

fiord

冗長な前置き

さて突然ですが、Esolang って聞いたことがありますか？ これは難解プログラミング言語のことで、意図的にプログラムが扱いづらいように言語仕様を設定しています。有名どころでは半角スペースとタブ文字、改行だけでコーディングするため画面が真っ白な「Whitespace」やコンパイラが 100B 前後の「Brain(自主規制)[1]」といったところでしょうか。他にも、コードを 2 次元で扱ったりソースが画像になっていたりします。といっても、実際にはただ難解なだけではなく、ネタ性も重視されています。プログラマの一種の遊びで、TSG でも部内大会を開いたりしています。

私は Esolang には TSG に入ってから触れるようになりました。私が触れた言語の 1 つに「Taxi」という言語があります。文字通り、タクシーを運転しながら、プログラムを進める、というものです。この言語の特徴として、「意味の通った英語でプログラムを書く」というものがあります。これは実際にコード[2]を見てもらった方が早いでしょう。

```
50 is waiting at Starchild Numerology.
Go to Starchild Numerology: west 1st left, 2nd right, 1st left, 1st left, 2nd left.
Pickup a passenger going to Joyless Park.
Go to Joyless Park: west 1st right, 2nd right, 1st right, 2nd left, 4th right.
Go to Post Office: west 1st left, 1st right, 1st left.
...(この後途方もない量が続きます)
```

これは英語で実際にタクシーを操作しています。値を設置し、タクシーを移動させて回収、別の場所へと値を動かしています。これもプログラミング言語ですが、一般的に考えられる上に挙げたようなものと比べると、自然言語が扱われており、かなり親しみやすいように感じます。代償としてコードはとても冗長なものになってしまうのですが、私はこの言語を書いていて楽しかったのです（ところで、この時点で普通の人は狂気を感じるのでしょうか。私は普通ですよね…？）。

さて、楽しかった私は、「実際にタクシーを運転してもらうゲーム」という形式でこの言語を知ってもらおうと思ったのです。

[1] 自主規制なので分かりづらさ全開ですが察してください。

[2] 今年の夏に行った第 3 回コードゴルフ大会で私が書いたものの冒頭です。コード全容は hyoga.hatenablog.com に落ちています。

因みに、私は普段競技プログラミングや CTF を主にやっていますが、特にすごい知識を持っているプロという訳でもなく、両方とも既に部内にプロが居て記事を書くと叩かれそうなので、安全そうな Esolang から出します。が、こちらもプロがいるので、叩かれて灰になっているかもしれません。

ゲーム説明

皆さんにはタクシーを運転してもらいながら、指定された操作（例えば「数字を 2 つ渡すので、これらの和を求めてください」など）を行ってもらいます。地図上のタクシーを様々な場所へと移動させていくのですが、その際に客を拾って運転していきます。客は 1 人に 1 つ値を持っていて、彼らを目的地に下すと、その目的地に応じた演算が行われ、その結果を客として拾うことも出来ます。

さて、簡潔に書こうとして Esolang 並に難解になってしまった説明をした訳ですが、出来る操作は以下の 2 つです。

- 客を拾う。その際、客の行先を指定する
- 移動する。客を乗せていて、その客の目的地に着くと、客は代金を払って降り、演算が行われます。

Taxi Garage というところに到着すると操作終了で、正しい操作を行ったか判定が行われます。途中でガソリンが無くなる（ガソリンスタンドでお金を払って追加できます）、間違った答えを出力するなどをすると社長にクビにされます。答えが正しければクリア。ランキング機能付きで、「移動距離」と感想に述べている「コード量」の 2 種類のランキングを用意しています。コード量については、操作中に確定した部分のコードが表示されるので、その中でどうすれば短くなるのか考えていってください。

感想

ゲームで行う内容は Taxi 言語に置換できるようになっていて、裏でこっそり言語化を行ってそれをランキングとして用いております。ただ、本来実装されているループを行う際は、そのコードをゲーム中に表示して挿入する、という感じになるしかないかな…と思っています（つまり未実装で、ゲームの内容は全て定数時間で解けるものです）。ただ、そうするとどうしても敷居が上がってしまうので、プログラマでなくても気軽に楽しめる、という前提の下で考えたいです。まあ、実際には Hard 以外地図をオリジナルで実装しているので、そのまま Taxi 言語として動かせる訳ではありません。因みに、逆に任意の Taxi コードをゲーム操作に置き換えることは出来ませ

ん。Taxi は途中で操作ミスをすると即クビ=GameOver になってしまうという仕様なので、これをゲームに入れると鬼畜ゲー化してつらいです。

ゲームを製作する中で、結構 mnemo からアイデアを引っ張ってきてしまったので、先輩方ごめんなさいという感じが物凄いです。まだデザインを中心に未完成なので、完成していなかったらごめんなさい。大人しくプログラマ展示 (制作の終わらなかった人々がその場で実装している様を展示する企画) になっています。完成したとしても Hokkaido Univ.&Hitachi 1st New-concept Computing Contest 2017 があるので、いずれにせよプログラマ展示になっていそうですね · · ·

一般記事

SA-IS アルゴリズムの空間計算量

gasin

世にも有名なアルゴリズム、SA-IS の空間計算量をできるだけ削ってみた話です。

とはいえ、このアルゴリズムは計算量、空間計算量共に線形なので所詮は定数倍です。

プログラムには通常、メモリの制限があるため、空間計算量を削ると大きい入力にも対応できるようになるのがうれしいところの一つです。

SA-IS アルゴリズム

文字列が与えられたとき、線形時間でその接尾辞配列を求めるアルゴリズムのことです。

接尾辞配列は接尾辞をソートしたものなので、文字列の比較に O(size) かかることや数字のソートに O(NlogN) かかることを考えるとこれが線形時間で行えることのすごさがわかると思います。

SA-IS のアルゴリズム自体は調べれば資料がありますし、ここで改めて同じことを説明する意味もないと思うので省略します。

改善結果

改善結果としては以下の図ような感じです。

実は図 5.1 には入出力文の配列も含まれており、図 5.2 には含まれていないので多少の嘘があるのですが、とりあえず extra memory space を O(2*SIZE) にすることができました。

計算量を落とさないまま空間計算量を落とすのはこれ以上は厳しいように僕は感じましたが、ちょっとの工夫でできる気もするので興味のある人はやってみるといいと思います。

```
int buf[2*SIZE+100];
int SA[2*SIZE+100];
int SL[2*SIZE+100];
int is_LMS[SIZE+10];
int st[2*SIZE+100];
int en[2*SIZE+100];
```

図 5.1: 初期のメモリ使用量

```
int buf[SIZE+10];
int pos[SIZE+10];
```

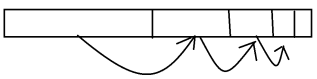
図 5.2: 頑張った後のメモリ使用量

具体的にやったこと

実際にやったことをつらつらと書いていてもしょうがないのでやったことをいくつか軽く説明します。

再帰のメモリの展開

SA-IS では再帰処理が行われますが、再帰するたびにサイズが $\frac{1}{2}$ になり、そのおかげで計算時間が線形で保たれています。勿論これはメモリについても同様のことがいえるので、先に 2*SIZE 分のメモリを確保しておいて、再帰をするたびに適切にシフトをすることで動的にメモリを確保することなく効率的に処理を行えます。



一般に、動的にメモリを確保する処理は重いということでこれは初期のときから実装されているものです。これ自体は特にメモリを削減するものではないです。

しかし、今見ているサイズが N であるとすると、その先には N だけこれから使われる予定の配列が確保されているため、この部分はうまく使いまわすことができます (大切)。

1 つの変数に複数の情報を詰め込む

int 型は 2^{32} 分の情報量を持つわけですが、int 型に与える値域がそれよりも小さかった場合、残りの部分を使って別の情報を持たせることでメモリを節約できます。

今回の場合は、LMS であるかどうかなどの bool 判定を他の変数に埋め込みました。しかし、これを行うとコードが非常に読みにくくなるので普段はあまりやらないほうがいいように思います。

1 つの変数に色々な仕事をしてもらう

普通にプログラミングをするときは、各変数に意味を持たせてそれに従って使いますが、勿論そんなことを許すわけはなく、あちこちで使いまわします。

例えば、入力された配列は流石に破壊してはいけないのであまり有効に使うことはできませんが、出力用の配列はそこまでの過程であちこちで使いまわします。

一番最初の処理を別で行う

文字列が入力されるので、せいぜい文字種は 256 文字しかなく[1]、これは文字の長さに比べて非常に小さいです。

一回再帰に入ってしまうと文字種は文字列長とほぼ同じになってしまうのですが、一番最初だけでも工夫する価値はありそうです。なぜなら、最初の文字列長は N であり、その後は再帰するごとに文字列長が半分になるため最初の影響力が非常に大きいからです。

具体的には、文字種分の配列を新たに確保し[2]、うまく使うことでメモリを節約します。これによって、もともと 2*SIZE 分必要だったメモリの一部を SIZE 分だけで済むようにすることができます。

再帰の前後で余計なデータを保持しない

再帰関数内で再帰関数を呼び出す場合に、呼び出す前の状態を保持していると復帰したときにスムーズに計算を再開することができて計算速度的には嬉しいのですが、その分メモリを確保しなければなりません。

これは計算速度との兼ね合いですが、今回はメモリに重点を置いているので極力速度を落とさないように、できるだけデータを保持しないようにしています（状態を戻すための最低限のデータは保持する必要があります）。

おわりに

空間計算量を削ったので今度は計算速度をという話になりそうなものですが、正直 SA-IS に多少飽きたのでしばらくは放置しそうです。

[1] 256 というのは char 型のサイズで、これは今回の問題で勝手に設定されたものですが

[2] これは一般に文字列長に比べて十分小さいので無視をしていますが、厳密には文字列長が文字種数と同程度、またはそれ以下の時を考慮して空間計算量を書くべきです。

Ruby の正規表現で計算する

akouryy

はじめに　～続・正規表現超絶技巧～

文字列処理を行う際に重宝する正規表現は、本来は正規言語というかなり限られたルールに基づいたマッチングしかできませんでしたが、Ruby(やその他いくつかの言語) においては正規言語の枠を超え、より大きな力を持つツールとなりました。

この記事では、正規表現 (以降、指定のない限り Ruby2.4.2 における拡張された「正規表現」を指します) で解くことのできた問題を通して、正規表現の力を紹介します。正規表現の基本的・発展的な知識を前提とするので、昨年の部報の moratorium さんの記事「正規表現超絶技巧[1]」を先にご覧になることをお勧めします。

なお、この記事のテストコードは Gist[2]に公開されています。

補数

正規表現で「計算する」といっても、正規表現には任意の文字列を出力する機能はありません。そこでこの記事では全ての問題を「入力全体が○○のときマッチせよ」という形式で設定します。

問題 1: 同じ桁数 (n とする) の 2 進数がカンマ区切りで 2 つ与えられる[a]。右の数が左の数の 1 の補数 (bitwise NOT) になっているときマッチせよ。
例: `100,011` にマッチし、`100,010` にはマッチしない。

[a]これ自体を正規表現でチェックすることもできますが、非本質的な部分が増えるだけなので今回は無条件で信頼し、この前提に反する入力は動作未定義とします。

HITCON CTF 2016 moRE で出題されていた問題です。これは「正規表現超絶技巧」でも紹介されていますが、この記事で重要となるテクニックを含むので (別解を) 紹介します。

この問題で一番重要なのは、「左の数の i 桁目[3]」と「右の数の i 桁目」をどう対応させるかという点です。「正規表現超絶技巧」では再帰の深さを利用するという方針でしたが、ここでは左の i

[1]TSG 部報 2016 年駒場祭号 p.5 (http://tsg.ne.jp/buho/312/buho312.pdf)
[2]https://gist.github.com/akouryy/fa6a58129bd3c7d3e0ba9fae36ba40a3
[3]単に x 桁目と書いたとき**上から** x 桁目を表すものとします。

桁目を読むタイミングで右の i 桁目も読みに行きます。

具体的には以下のようになります[45]。

```
1  %r{ ^
2    (?:
3      (?<A> [01] )
4      (?= (?<B> [01] \g<B> [01] | , .*? (?! \k<A>) [01] ) $ )
5    )++
6    , [01]++ $
7  }x
```

左の i 桁目が (?<A>) にマッチしたあと、先読みで右の i 桁目を読みに行きます。(?<B>) の動作は、左右の数の $i+1$ から n 桁目をそれぞれ昇順、降順に読みながら再帰していき、$n-i$ 桁読み終わったところでカンマに到達する、その際残っている文字列の右端が右の数の i 桁目なので、この文字が \k<A>と異なることをチェックする、というものです。

つまり、それぞれの数の i 桁目の後にはどちらも $n-i$ 桁の数字が存在する、という事実を用いています。

例: 101,010

1 桁目 A:1 → B_0:0$B_1$0 → B_1:1$B_2$1 → B_2:,0 → B_2 の右端:0 != A
2 桁目 A:0 → B_0:1$B_1$0 → B_1:,01 → B_1 の右端:1 != A
3 桁目 A:1 → B_0:,010 → B_0 の右端:0 != A

2 進数と “x”

問題 2: 2 進数および “x” が 0 個以上並んだ文字列がカンマ区切りで与えられる。右の x の文字数が左の 2 進数に一致するときマッチせよ。
例: 100,xxxx にマッチし、100,xxxxx にはマッチしない。

dai さんに出題された問題です。

左の数を上の桁から順に読んでいきながら x の列を構成していきます。この方針に従うと下のような正規表現を書きたくなります (が、動きません)。

```
1  # Wrong Answer
```

[4] %r{〜}x が正規表現リテラルです。パターン内の空白やコメントが無視されます。

[5] ++、*+、?+は「絶対最大量指定子 (possessive quantifier)」と呼ばれるもので、この記事内では単なる+、*、?と考えても問題ありません。詳しくはググってください。

```
2  %r{ ^
3    (?<B>)
4    (?<A>
5      0 (?= [01]* , (?<B> \k<B-1>{2}  )) \g<A>
6    | 1 (?= [01]* , (?<B> \k<B-1>{2} x)) \g<A>
7    | , \k<B-1> $
8    )
9  }x
```

(?<B>) が “x” の並ぶ列 (初期値:空文字列) です。普通の文字列→数値変換と同じように、左の桁から読み込み、“0” なら (?<B>) を 2 倍した (2 回繰り返した) もの、“1” なら 2 倍した上でもう 1 つ “x” を付け加えたものを新たな (?<B>) とする……つもりでしたが、うまくいきませんでした。

“10” という並びがあったとき、“0” のときに \k<B-1>で参照したい “1” 内の (?<B>) は、“0” の \k<B-1>より後にあります。おそらくこれが原因で、\k<B-1>は何も参照することができていません[6]。悩んだ末に絞り出した解決策がこちらです。

```
1  %r{ ^
2    (?<B>)
3    (?<A>
4      (?: 0 (?= (?<D>)) | 1 (?= [^x]*+ (?<D> x)))
5      (?= [01]*+ , (?<B> \k<B-1>{2} \k<D+0>))
6      \g<A>
7    | , \k<B-1> $
8    )
9  }x
```

(?<D>) には 0 個または 1 個の “x” が入り、\k<D+0>で参照されます。2 つの (?<D>) はどちらも \k<D+0>より前にあるため、期待通り動作します。

例: 101,xxxxx

初期状態 B: ""
1 桁目 (1) D: "x" → B: ""*2+"x" = "x" (x が 1(2) 個並んでいる)
2 桁目 (0) D: "" → B: "x"*2+"" = "xx" (x が 10(2) 個並んでいる)
3 桁目 (1) D: "x" → B: "xx"*2+"x" = "xxxxx" (x が 101(2) 個並んでいる)

[6]本当にそうなのかは知らないので時間があるときに Onigmo のソースコードを読んでみたいです。

インクリメント

問題 3: 先頭に余分な “0” のない[a]2 進数がカンマ区切りで 2 つ与えられる。(右の数)=(左の数)+1 のときマッチせよ。
例: `100,101` にマッチし、`100,110` や `100,100` にはマッチしない。

[a]非零ならば “1” で始まる。

まずはコーナーケース (特殊な入力) を処理します。「0,1」や「$11\cdots\overset{n\text{個}}{\cdots}1$,$100\cdots\overset{n\text{個}}{\cdots}0$」にはマッチする必要があります。

それ以外の場合、左の数に 1 を足したとき何が起きるのか考えます。まず、「$11\cdots\cdots1$」を除外したので、左の数と右の数の桁数は同じになります。その上で、左の数の**末尾**の「$01\overset{0\text{個以上}}{\cdots\cdots}1$」を「$10\cdots\cdots0$」に置き換えたものが右の数になることが必要十分条件だとわかります。

```
1  %r{
2    ^ 0,1 $
3  | ^ (?<A> 1 \g<A> 0 | , 1 ) $
4  | ^ (?= 1 (?<B> [01] \g<B> [01] | , 1 ) $ )
5    (?<C>
6      (?<D> [01]) (?= (?<E> [01] \g<E> [01] | , [01]* \k<D> ) $ )
7    | 0 (?<F> 1 \g<F> 0 | , [01]* 1 ) $
8    )++ $
9  }x
```

`(?<A>)` はコーナーケース「$11\cdots\overset{n\text{個}}{\cdots}1$,$100\cdots\overset{n\text{個}}{\cdots}0$」を調べるものです。

`(?<B>)` では左右の桁数が等しいことを確認します。次に `(?<E>)` では「補数」の解答の `(?<B>)` と同様のテクニックを用いて、左右の対応する桁の文字 (`(?<D>)`) が同じということを調べています。最後に `(?<F>)` で「$01\overset{0\text{個以上}}{\cdots\cdots}1$」と「$10\cdots\cdots0$」が残っているか調べて、問題 3 の機能が実現できました。

加算

問題 4: 同じ桁数の 2 進数がカンマ区切りで 3 つ与えられる。左の 2 数の和が右の数と等しいときマッチせよ。
例: `101,001,110` にマッチし、`100,001,101` や `100,100,000` にはマッチしない。

今回の記事の目玉になるはずだったのですが、原稿締切直前に方針を変えたためぎりぎり解き終わりませんでした。方針は主に 2 つ考えられて、

- 「2 進数と “x”」を参考に、下の桁から順に見ていき、繰り上げをキャプチャしながら右の数の対応する桁と比較していく
- 「インクリメント」を参考に、それぞれの桁について、繰り上げが存在するかどうかをその都度チェックしてから右の数の桁と比較する

となります。

最初は前者の方針を試みましたが、まさに「2 進数と “x”」と同じ問題が発生し、今度は問題 2 の解答における `(?<D>)` が繰り上げを保持するのですが、その中で `\k<D-1>`が必要になりました。この問題の場合 `\k<D-1>`の参照を共通化してくくり出すことができなかったので、「2 進数と “x”」とは違い、動かすことができませんでした。

というわけで後者の方針をとります。残念ながら締切までに完成させることはできませんでしたが、「インクリメント」を発展させれば解けるはずです。解けたら私のブログ[7]に掲載します。

まとめ

最後の問題は解き終わらず残念でした。個人的な感覚としては、後者の方針は前者の方針と比べて「**美しくない**」(かつ計算量的に遅そうに見える) ので、前者の方針で書けたという方がいれば教えてください。後者の方針の報告はいりません。

今後やりたいこととして、加算の正規表現の完成など個別の問題を解くのはもちろんですが、理論的に今の Ruby の正規表現の限界がどこにあるのか考えてみたいと思っています。

皆さんもぜひ色々な決定問題を Ruby の正規表現で解いて、正規表現と戯れてみてください。

[7] http://akouryy.hatenablog.jp/

編集後記

★ TSG へお越しくださってありがとうございます
★ 締切ギリギリでの作業だったので何かと粗くなってしまいました
★ 1 年間よろしくお願いします.

理論科学グループ　部報 第 313 号

2017 年 11 月 22 日　発行
発行者　加藤善夫
編集者　佐藤英一郎
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 **313** 号

── 駒場祭パンフレット号 ──

2017 年 **11** 月 **22** 日

THEORETICAL SCIENCE GROUP