パー研の部誌

AGC 展の展示母体である麻布学園パーソナルコンピュータ研究部の部誌です。

部員の製作や活動に関する技術的なコラム的なものがいくつか乗っています。この部誌の内容は、 https://apcc.github.io にアクセスするか、以下の QR コードから読むこともできます。

部誌
オセロの作り方・実装　　H1-3 笹倉颯人

いつの間にかもう高1になっていて、時間の流れの早さ恐ろしさを感じつつ、
中2と中3の文化祭ではろくでもないものしか出せなかったので、今年は
オセロを作って出すことにしました。なので部誌にどんな感じで動いているのかなどを
ざっと書きたいと思います。

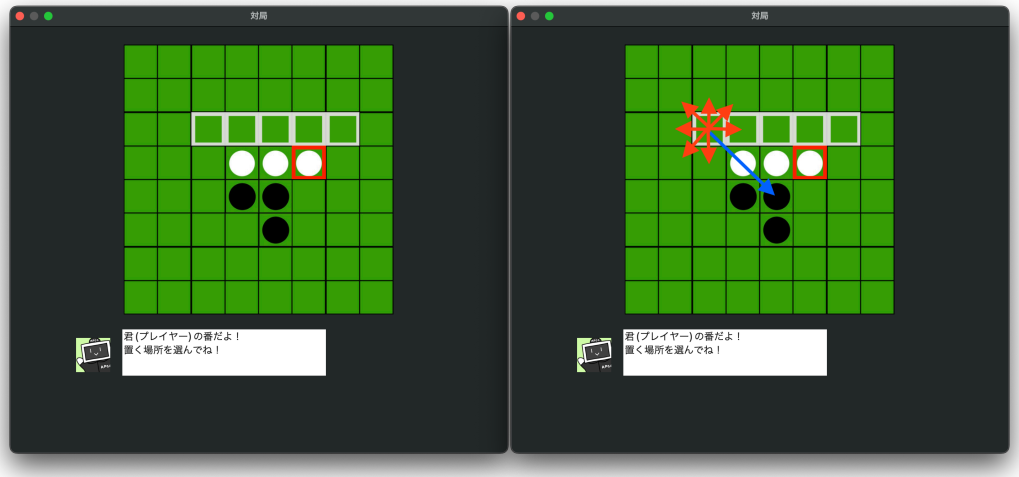
1. 基本的な実装

(1) 盤面

　まず、ボードゲームで絶対に必要な盤面の実装。
いくつか手法があり、有名どころではビットボード、配列などがあります。
ビットボードは聞いたことがある人もいるのではないでしょうか。
ここでは最も簡単な配列について話します。
配列というのは、「複数の変数をまとめられる箱」みたいなものです。8x8(64個)の要素を持てる配列を作って、何もない状態を0, 黒を1, 白を2のようなな感じで割り当てます。簡単ですね。
ちなみにビットボードでは、64桁の2進数2つで盤面を表現します。
詳しい説明はここでは割愛します。

(2) 石を置ける場所の判定

　置ける場所の判定は、なんとなくわかっている人もいると思いますが、
各空きマスから8方向に矢印を飛ばすイメージです。隣の石が相手の色の場合はさらに先に進んでいき、最後に自分の石があれば置くことができます。

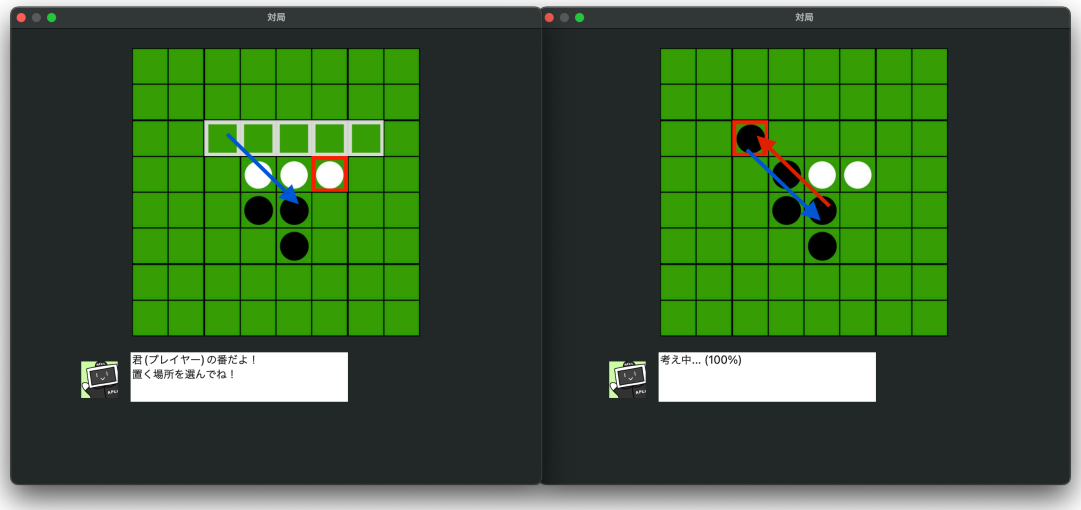


図のように空きマスから8方向に矢印を飛ばします。
これを全ての空きマスに対して行なって石が置けるかどうか判定します。

ただこの方法は何度も繰り返し矢印を飛ばしてその先に石があるかどうか判定するので
時間がかかります。1回実行するくらいなら問題ないのですが、botを実装するとなると
繰り返す回数が大幅に増えるため話が変わってきます。詳しいことは後ほど説明します。

(3) 石をひっくり返す処理

　石を置ける場所の判定の処理とほとんど同じです。



違いは飛ばした矢印を逆向きに飛ばすところです。
(図の赤色の矢印)
置いた場所まで飛ばして通った場所にある石を自分の色にしていきます。

ひっくり返す処理は置いた場所だけでいいので、置ける場所の判定に比べれば軽いです。
ただ、2回同じことをやっているので少し無駄があります。

2. botの実装

　とまあこんな感じで基本的な実装を紹介したところで、じゃあbotはどうなってるの？
という説明をしていきます。
Botを作るのには、まず「アルゴリズム」というものが必要です。
いくつか種類がありますが、基本的には「MiniMax法」と「モンテカルロ法」の
2つがあり、その派生の「AlphaBeta法」や「NegaAlpha法」などがあります。
僕のオセロでは、NegaAlpha法を使っています。

(1) 盤面の評価

　まず一番大事なものとして、「盤面の評価」があります。
先ほど言ったモンテカルロ法では必要ないのですが、MiniMax法の派生のアルゴリズムでは
必要です。作成方法はいくつかあるみたいですが、今回は簡単で強いbotが作れる
手法を説明します。

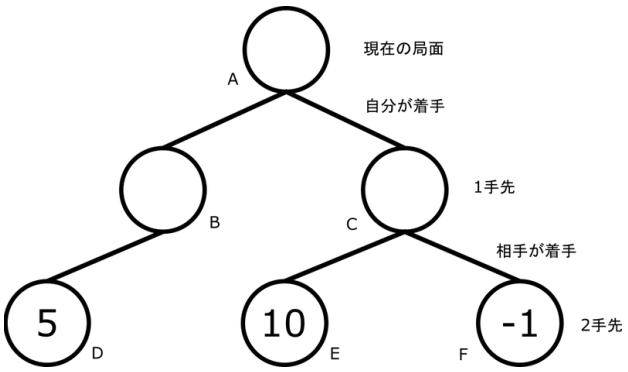
30	-12	0	-1	-1	0	-12	30
-12	-15	-3	-3	-3	-3	-15	-12
0	-3	0	-1	-1	0	-3	0
-1	-3	-1	-1	-1	-1	-3	-1
-1	-3	-1	-1	-1	-1	-3	-1
0	-3	0	-1	-1	0	-3	0
-12	-15	-3	-3	-3	-3	-15	-12
30	-12	0	-1	-1	0	-12	30

こんな感じでマス一つ一つに重みをつけて合計値で判定します。
角は数値が大きく、角の隣は低いなど、基本的なことです。
ただし、これだけだと弱いので、いくつか別のものを
足し合わせるのですが、ここでは説明は省きます。

(2) NegaAlpha法

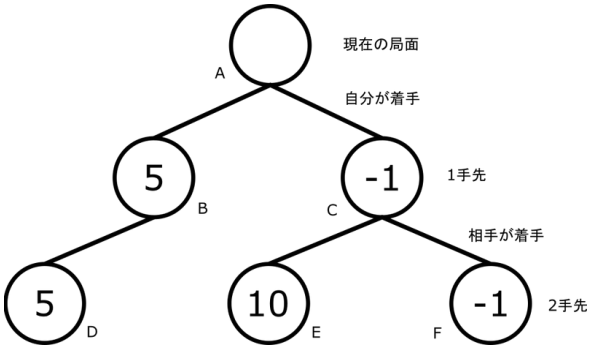
　では盤面の評価方法の説明を終えたところで、botの動きの「コア」となる
アルゴリズムの説明をしてきます。
いきなりNegaAlpha法の説明をしても理解しにくいと思うので、まずはベースとなっている
Minimax法の説明から始めます。

[1] Minimax法
　1手先を見るだけであれば、石を置いてひっくり返し、その時の評価値をとり、
それを全ての空きますに対して行って最も評価値の高いものを選べばいいだけです。
しかし、2手、3手先を読むとなると話は変わってきます。その際によく使われるアルゴリズムが
MiniMax法です。



この図(※1)のようなものを「ゲーム木」
というのですが、
これを使って説明していきます。

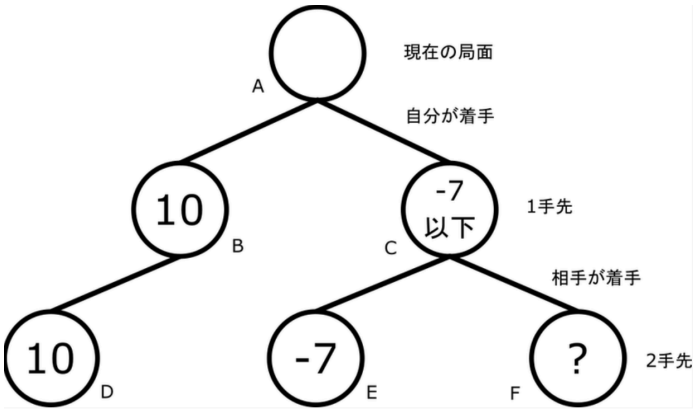
上の図に2手先の評価値が書いてあります。1手先であるB,Cのどちらを選べば
最大の数値となるでしょうか？「Eが10だから..Cだ！」と思うかもしれませんが、
2手先は相手の番です。自分にとって最も良い手を選ぶことなんてそうそうないでしょう。



実際にはこの図(※2)のようになります。相手は最小値(min)、自分は最大値(max)を選びます。
これがminimax法の名前の由来にもなっています。
また、値に-1をかけて常に最大値を選ぶようにしたNegaMax法というものもあります。

[2] NegaAlpha法

　[1]でMIniMax法を紹介しましたが、このアルゴリズム、全部探索するために
無駄が多く、すごく遅いです。それを改善するために、その手を選ばないことが確定した時点で
計算をカットして無駄な計算を省くAlphaBeta法があります。NegaAlpha法はNegaMax法と
同じように-1をかけて常に最大値を選ぶようにしたものです。



左の図(※3)のように、2手先(相手の番)で
-7評価値としてある時、相手は評価値が
-7以下の手を選ぶことが確定します。
結果的に1手先(自分の番)の評価値は-
7以下となり、別の手(評価値が10の手)
の方が評価値が高いため、選ぶことは
なくなります。このようにして余分な
計算をカットしたのがAlphaBeta法です。

「そんなに変わらないんじゃないの？」と思うかもしれませんが、先読みする手数が増えれば
増えるほど影響は大きくなります。6手先まで読むのであれば、盤面の状況にもよりますが
計算量は1/10ほどまで減らせることがあります。先ほど盤面の実装の(2)で、
「繰り返す回数が大幅に増えるため話が変わる」と書きましたが、これは
このアルゴリズムを実行する際に 『おける場所を列挙=>石を置いてひっくり返す
=>指定した手読み数まで繰り返す=>評価値を出す=>1手戻す』という処理を
繰り返すためで、手読み数が増えるとものすごい勢いで繰り返す回数が増えます。
なので少しの無駄がbotを非常に遅くしてしまいます。そうならないために、
無駄な処理は減らした方がいいです。

3. おまけ

　どうでしょうか？なんとなく理解できましたか？理解してもらえたなら嬉しいです。
では最後に、少しだけビットボードの話をしたいと思います。少し難しいので、
簡単に説明します。

(1) ビットボードとは？

　ビットボードというのは、盤面を2進数で表し、全てのマスを0か1かで定義するものです。
オセロはマスが合計で64マスあり、現代のコンピューターはほとんどが64bit CPUを
使っているのでピッタリ1つの変数で盤面を表すことができてちょうどいいです。
実際には黒、白の2色分あるので2つの変数ですが。では、そんな直感的ではない実装をして
何がいいのかという話ですが、直感的な実装に比べて圧倒的に高速に処理できます。
配列の実装では何度も繰り返し処理が必要な「おける場所の判定」「石をひっくり返す処理」
などの処理を繰り返しを使わず、なおかつ少ない処理回数で実装できてしまいます。
強豪オセロAIではビットボードで実装されているものが多いです。
気になる人はこのサイトを見てみるといいかもしれません。
https://qiita.com/sensuikan1973/items/459b3e11d91f3cb37e43

4. 終わりに

　こんな感じでオセロの基本的な実装について書きましたが、どうだったでしょうか？
実は基礎的なプログラミング技術を持っていれば誰でも作れます。
もしよければ、皆さんも作ってみてください。

作る際に参考にしたサイト：

- オセロAIの教科書
  https://note.com/nyanyan_cubetech/m/m54104c8d2f12
- オセロをビットボードで実装する
  https://qiita.com/sensuikan1973/items/459b3e11d91f3cb37e43
- minimax法とαβ法
  https://kowaragan.com/gameinformatics4minmaxalphabeta/

※1, 2 オセロAIの教科書 4. minimax法
　　https://note.com/nyanyan_cubetech/n/n98c9a37a54fc　より引用
※3　オセロAIの教科書 5. alphabeta法
　　https://note.com/nyanyan_cubetech/n/n210cf134b8b1　より引用

CDCL ソルバの解説と実装

Tamego　　津久井 里央斗

2024 年 4 月 20 日

概要

制約充足問題を解くための SAT ソルバによく使われる CDCL アルゴリズムがある。まず考えたい問題である制約充足問題を紹介し、その後 SAT ソルバーを解説しながら実装していく。実行したときにどのように処理が行われているかを可視化するために、manim という python のライブラリを使ってアルゴリズムを動画で出力するようにもする。

1　制約充足問題

1.1　用語の定義

定義 1（割り当て） 領域 D から論理式 ϕ への**割り当て**（assignment）とは、ϕ の変数から D の要素への写像のことである（簡単に言えば、ϕ の各変数に値を対応させること）。今考えている問題において、$D = \{0, 1\}$ である。

定義 2（充足可能性） 論理式が**充足可能**（satisfiable）であるとは、その論理式が真となるような割り当てが存在するときのことである。

定義 3（命題変数） **命題変数**（boolean variable、propositional variable）とは、値が真か偽である変数のことである。

定義 4（原子論理式） **原子論理式**（atom）とは、一般的には述語のことであるが、今考えている問題においては命題変数のことである。

定義 5（リテラル） **リテラル**（literal）とは、原子論理式またはその否定である。**正リテラル**（positive literal）とは、原子論理式であるようなリテラルのことで、**負リテラル**（negative literal）とは、原子論理式の否定であるようなリテラルのことである。

定義 6（リテラルの評価） 正リテラルが**充足**（satisfied）しているとは、その原子論理式が真と割り当てられているときのことである。逆に、負リテラルが充足しているとは、その原子論理式が偽と割り当てられているときのことである。

定義 7（CNF） 論理式が **CNF**（conjunctive normal form、連言標準形）であるとは、その論理式が 1 つ以上のリテラルを**または**（or、論理和）でつないだもの（**節**（clause））を 1 つ以上**かつ**（and、論理積）でつないだものであるときのことである。すなわち

$$(l_{11} \vee \ldots \vee l_{1m}) \wedge \ldots \wedge (l_{n1} \vee \ldots \vee l_{nm})$$

のような論理式を CNF と呼ぶ。

定義 8（節の評価） 節が**充足**（satisfied）しているとは、その節の 1 つ以上のリテラルが充足しているときのことである。節が**矛盾**（conflict）しているとは、その節の全てのリテラルが割り当てられているがそれらが充足していないときのことである。節が**単位節**（unit clause）であるとは、その節の 1 つ以外のリテラル全てが割り当てられていてその節が充足していないときのことである。節が**未解決**（unresolved）であるとは、節が充足・矛盾・単位節のいずれでもないときのことである。

1.2 命題論理

論理式が**命題論理式**（propositional formula）であるとは、原子論理式を命題変数として、

- 原子論理式
- 論理式 A が命題論理式のとき、(A)
- 論理式 A が命題論理式のとき、$\neg A$
- 論理式 A, B が命題論理式のとき、$A \wedge B$

のいずれかで表されるときのことである。例えば、$\neg A \wedge B$ という論理式は、

- A は命題論理式
- $\neg A$ は命題論理式
- B は命題論理式
- $\neg A \wedge B$ は命題論理式

のようにして命題論理式であることが分かる。ここでは $\neg$ と $\wedge$ しか命題論理式の演算子としていないが、$\vee$ のような他の演算子はその 2 つを組合わせることで作ることができるので、この 2 つだけで十分である。したがって $\neg A \vee B$ も命題論理式と言える。

制約充足問題（satisfiability problem、SAT 問題）とは、命題論理式の充足可能性を決定する問題のことである。ここでは厳密な説明はしないが、全ての命題論理式は CNF に変換することができる（Tseitin 変換）。なので制約充足問題を解くソルバを作るためには、CNF だけ扱えれば十分である。

2　CDCL ソルバ

2.1　方針

CDCL ソルバの説明を始める前に、制約充足問題を解くアルゴリズムがどのようなものになるかを考えてみる。

簡単のためにここでは CNF のみが与えられるものとする。全ての割り当てを調べて充足するようなものがあるかどうかを調べる方法が思い付きやすいと思う。ただ変数の数を n 個とすれば、2^n 通りの割り当てが存在することになる。n が大きくなると現実的な時間で処理が終わらなくなってしまうため、どうにかして改良したい。

割り当ての途中で変数の真偽値が分かることがあるということに注目してみる。例えば、次のような CNF$(x_1 \vee x_2 \vee \neg x_3)$ があって、部分的な割り当て $\{x_2 \mapsto 0, x_3 \mapsto 1\}$ を考える。すると、この節は x_1 以外のリテラルは全て割り当てられていて充足していないことが分かる（これを単位節というのであった)。この単位節を充足させるためにはリテラル x_1 が充足するように割り当てる必要がある。

ある部分的な割り当てが与えられて、単位節が発生するような状況では、まだ割り当てられていないリテラルが充足するようにしなければならない。これは**単位節規則**（unit clause rule）と呼ばれているものである。この規則を使うことで組み合わせをいくつか減らすことができる。さらに CDCL では**学習**（learn）をすることで探索の量を減らしている。詳しくは 2.5 節を参照すること。

このようにして制約充足問題を解くアルゴリズムが作れそうである。次は具体的な CDCL の仕組みについて見ていく。

2.2　CDCL

CDCL は前節の方針を参考にして次のように実装できる：

1. 単位節規則が適応できなくなるまで適応させる（**BCP**（boolean constraint propagation、ブール制約伝播)。
2. その結果が矛盾していたらそれを解析・学習していくつかの決定をやりなおす（**矛盾の解析**（analyze conflict))。このとき何も決定されていない状況で矛盾したなら充足不能と判定する。
3. もしそうでなければ新しく変数を選んで**決定**（decide）し、最初に戻る。

ここで出てきた決定とは、まだ割り当てられていない変数を 1 つ選んで真に割り当てることをいう。下に CDCL の具体的なソースコードを示しておく。

```
1   def CDCL(self) -> bool:
2     while True:
3       while not self.BCP():
4         backtrack_level: int = self.analyze_conflict()
5         if backtrack_level < 0:
6           self.change_title("unsatisfiable")
7           return False
8         self.backtrack(backtrack_level)
9       if not self.decide():
10        self.change_title("satisfiable")
```

```
11          return True
```

2.3 入力のパーサー

このソルバは CNF のみを扱うが、CNF を直接入力することはできない。そこで DIMACS CNF という形式を使って CNF を表現することにする。

DIMACS CNF は先頭に命題変数の数と節の数を書き、その後に続いて節の中身を書く。数字が命題変数を表わし、それが負の数であればそれは否定を表わす。例えば次のような DIMACS CNF 形式のファイル

```
p cnf 9 7
-1 -4 5 0
-4 6 0
-5 -6 7 0
-7 8 0
-2 -7 9 0
-8 -9 0
-8 9 0
```

は、

$$(\neg x_1 \vee \neg x_4 \vee x_5) \wedge (\neg x_4 \vee x_6) \wedge (\neg x_5 \vee \neg x_6 \vee x_7) \wedge (\neg x_7 \vee x_8) \wedge (\neg x_2 \vee \neg x_7 \vee x_9) \wedge (\neg x_8 \vee \neg x_9) \wedge (\neg x_8 \vee x_9)$$

というような CNF を表わす。さて、CNF を表現する方法が分かったが、それを python 内で扱わなければならない。ここでは、節を `List[int]` で扱うことにし、そうすると CNF 全体は `List[List[int]]` で扱うことになる。前の例を使うと、その CNF は python で

```
cnf: List[List[int]] = [
  [-1, -4, 5],
  [-4, 6],
  [-5, -6, 7],
  [-7, 8],
  [-2, -7, 9],
  [-8, -9],
  [-8, 9]
]
```

と表記できる。以下に DIMACS CNF 形式から python で扱える形に変換するプログラムを示す：

```
  def dimaccnf_parser(self, inputs: List[str]) -> None:
    lines = [
      line.split() for line in inputs
      if line[0] != 'c'
    ]
    if len(lines[0]) >= 4 and lines[0][:2] == ['p', 'cnf']:
      self.var_count = int(lines[0][2])
      self.clause_count = int(lines[0][3])
    else:
```

```
      raise SyntaxError("節と命題変数の数が最初に指定されていません")
    self.cnf: list = list()
    vars_set: set = set()
    literals: list = list()
    for line in lines[1:]:
      literals += map(int, line)
    clause: list = list()
    for lit in literals:
      if lit == 0:
        self.cnf.append(clause)
        clause = list()
      else:
        clause.append(lit)
        vars_set.add(abs(lit))
    self.create_clause_list()
    self.vars = list(vars_set)
    self.nodes = dict()
    for v in self.vars:
      self.nodes[v] = Node(v)
    if len(self.vars) != self.var_count:
      warnings.warn("命題変数の数が違います")
      self.var_count = len(self.vars)
    if len(self.cnf) != self.clause_count:
      warnings.warn("節の数が違います")
      self.clause_count = len(self.cnf)
```

それぞれの変数の割り当てなどの状態は Node クラスを用いて管理している。Node クラスは次のような定義がされている：

```
class Node:
  def __init__(
    self,
    var
  ) -> None:
    self.var = var
    self.value: int = Literal.UNASSIGN
    self.level: int = -1
    self.clause: list[int] = []

  def reset(self) -> None:
    self.value: int = Literal.UNASSIGN
    self.level: int = -1
    self.clause: list[int] = []

<<manim用の定義（Nodeクラス）>>
```

2.4 決定

どの変数を決定するかはソルバの速度にも関わるため重要で、様々な戦略が考えられてきた。ただここでは簡単のために、添字が小さい順に変数を決定することにする。

また、今までに決定された変数の数を**決定レベル**（decision level）という。決定レベルを保存しておくことで、ある時点まで決定を取り消すのが簡単になる。

```
  def decide(self) -> bool:
    for v in self.vars:
      if self.nodes[v].value == Literal.UNASSIGN:
        self.level += 1
        self.nodes[v].value = Literal.TRUE
        self.nodes[v].level = self.level
        self.add_node(v)
        self.assign_history[self.level] = deque()
        self.assign_history[self.level].append(v)
        return True
    return False
```

2.5 BCP

単位節規則が適応できなくなるまで単位節規則を適応させていく。より詳細には、

1. 全ての節を 1 つずつ評価する。もしそれが単位節なら単位節規則を適応させ、もしそれが矛盾しているなら `False` を返す。
2. 単位節規則が 1 度でも適応できたのなら最初に戻る。
3. そうでなければ `True` を返す。

といったようになる。

その節が単位節であるかどうかを判断したいので、その節を評価する関数を作りたい。定義 8 に従ってそのまま実装すればよい。

```
  def evaluate_clause(self, clause) -> int:
    unassign_count: int = 0
    for literal in clause:
      match self.nodes[abs(literal)].value:
        case Literal.TRUE if literal > 0:
          return Clause.SATISFIED
        case Literal.FALSE if literal < 0:
          return Clause.SATISFIED
        case Literal.UNASSIGN:
          unassign_count += 1
    match unassign_count:
      case 0:
        return Clause.CONFLICT
```

```
      case 1:
        return Clause.UNIT
      case _:
        return Clause.UNRESOLVED
```

この BCP の動作を分かりやすく表現するために**含意グラフ**（implication graph）がしばしば用いられる。頂点は変数の割り当てを表し、単位節規則によってある変数からまた別の変数が導かれるとき、そのある変数からその別の変数へと結ぶ。つまり辺が変数の割り当ての理由となる。含意グラフによって BCP がどのように動いているかを表現できるように、manim も用いながら実装する。

```
  def BCP(self) -> bool:
    self.change_title("BCP")
    for i in range(len(self.cnf)):
      clause: list[int] = self.cnf[i]
      match self.evaluate_clause(clause):
        case Clause.SATISFIED:
          continue
        case Clause.CONFLICT:
          self.nodes["conflict"].value = Literal.TRUE
          self.nodes["conflict"].clause = clause
          self.add_implicated_node("conflict", i)
          return False
        case Clause.UNIT:
          implicated_variable: int = -1
          reason: list[int] = list()
          for literal in clause:
            if self.nodes[abs(literal)].value != Literal.UNASSIGN:
              continue
            if literal > 0:
              self.nodes[abs(literal)].value = Literal.TRUE
            else:
              self.nodes[abs(literal)].value = Literal.FALSE
            implicated_variable = abs(literal)
            self.assign_history[self.level].append(literal)
          self.nodes[implicated_variable].level = self.level
          self.nodes[implicated_variable].clause = clause
          self.add_implicated_node(implicated_variable, i)
    return True
```

2.6 矛盾の解析

矛盾を解析するとあるが、これは矛盾の**原因**（reason）を探ることにある。その原因を否定した節を**学習節**（learnt clause）といい、これを追加していく過程を**学習**（learn）という。

矛盾の原因を考えるとき含意グラフを用いると分かりやすくなる。含意グラフを**原因側**（reason side）と**矛盾側**（conflict side）に分ける。矛盾側に含まれる変数は、どれもその変数から延びる辺を削除すると矛盾ノードにたどりつけなくなるような変数となる。このような設定により、矛盾側の変数に直接辺をもつ理由側の変数の集合が矛

盾の理由となる。そしてこの理由の否定である学習節を追加することで探索の量を減らすことができる。

原因側と矛盾側の分け方は 1 通りではない。どこで分けるかによってどのような学習節が得られるかも変わってくる。最初は矛盾ノードのみが矛盾側に属する状態から始める。そこから、理由の中で最近に割り当てられたリテラル（現在の決定レベルで、単位節規則から導かれたもののみ）から順番に矛盾側に加えていく。これをやめるタイミングは、現在の決定レベルの変数が 1 つのみになったときが良いとされている。これにはバックトラックが関係している。詳しい説明については 2.7 節を参照すること。

実装の方針を軽く触れておく。

1. 現在の決定レベルが 0 なら充足不能と判断する（決定レベル-1 にバックトラックしようとする）
2. 学習節を最初は矛盾を導いた節として設定する
3. 直近に割り当てたリテラル・そのリテラルを導いた単位節規則の根拠である節・学習節を組みあわせて新たに学習節を作る
4. 学習節の中で現在の決定レベルの変数が 1 つになるまで 1 つ前のステップを繰り返して、それが終わったら学習節を追加して、どこまでバックトラックするかを計算する

3 の操作は**二項導出**（binary resolution）と呼ばれている。これは、

$$(a_1 \vee \dots a_n),\ (b_1 \vee \dots b_n)$$

を前提としたとき、

$$(a_1 \vee \dots a_n \vee b_1 \vee \dots b_n)$$

が結論として導かれるというものである。3 つの要素を観察すれば、単位節規則の根拠になる節にはそのリテラルが含まれていて、学習節にはそのリテラルの否定が含まれていることが分かる。よってこの二項導出を使って新たに学習節を作ることができる。

```
  def analyze_conflict(self) -> int:
    self.change_title("analyze conflict")
    if self.level == 0:
      return -1
    learnt_clause: set[int] = set(self.nodes["conflict"].clause)
    self.conflict_side.add("conflict")
    self.scene.play(
      *self.format_graph(),
      self.learnt_clause_animation(list(learnt_clause)))
    while True:
      literal: int = 0
      for lit in reversed(self.assign_history[self.level]):
        if lit in learnt_clause or -lit in learnt_clause:
          literal = lit
          break
      learnt_clause.remove(-literal)
      for lit in self.nodes[abs(literal)].clause:
        if lit != literal:
```

```
          learnt_clause.add(lit)
      self.conflict_side.add(abs(lit))
      self.scene.play(
        *self.format_graph(),
        self.learnt_clause_animation(list(learnt_clause)))

      current_level_count: int = 0
      second_highest_level: int = 0
      print("--")
      for lit in learnt_clause:
        level: int = self.nodes[abs(lit)].level
        if level == self.level:
          current_level_count += 1
        elif level < self.level:
          print("variable:", abs(lit), "level:", level)
          second_highest_level = max(second_highest_level, level)
      if current_level_count == 1:
        print("learnt_clause:", learnt_clause)
        self.cnf.append(list(learnt_clause))
        self.add_clause_list(len(self.cnf) - 1, True)
        print("level:", second_highest_level)
        return second_highest_level
    return -1
```

2.7 バックトラック

指定された決定レベルより大きな決定レベルの変数の割り当てを削除する。どの決定レベルにどの割り当てが行われたかを保存しておくと、簡単に実装できる。

```
  def backtrack(self, backtrack_level: int) -> None:
    self.change_title("backtrack")
    self.remove_node_and_edge("conflict")
    for level in range(self.level, backtrack_level, -1):
      for variable in reversed(self.assign_history[level]):
        self.remove_node_and_edge(abs(variable))
      self.assign_history[level] = deque()
    self.level = backtrack_level
    self.conflict_side.clear()
```

3 おわりに

CDCL ソルバの解説を実装をしてきた。ソースコードは https://github.com/Tamego/SimpleCDCLSolver にて公開している。また生成される動画の例を https://youtu.be/8tLWwvICbgI で限定公開しているので、よければ参照してみてほしい。

タイピングゲームを実装する

H2-1-42 安田龍之介

タイピングゲームをやったことがある人は多いかと思います。寿司打など、タイピングの練習として単語をキーボードで打ち込むという形式のゲームのことです。AGC 展でも今年展示しています。

さて、あのようなタイピングゲームが、技術的にどのように実装されているかを考えたことはありますか？

この文章ではタイピングゲームをどのように実装すれば良いかについて考えていきます。なお、あくまで僕がこのやり方でうまく行ったというだけで、世の中のタイピングゲームが一般的にどう実装されているのかはよくわからないのでご了承ください。

さて、シンプルに考えれば、単語のひらがな列を特定のルールに基づいてローマ字列に変換し、入力がそれに沿っているかを確認していけば良いように思えます。

しかし、これではダメです。なぜなら、ローマ字には複数のパターンがあり、有効なローマ字列ならどう打っても正解となるようにしたいからです。

例えば、「じねんじょ」は「jinenjyo」と打つのが一般的でしょうが、「jinennjilyo」などと打つこともできます。

それを踏まえると、十分な機能を持つタイピングゲームは以下のようなことができる必要があります。

- まず、入力する単語と、その単語をローマ字で書く時の最も一般的と考えられるローマ字列(これを仮に予測列と呼びます)を画面に表示する
- プレイヤーのキー入力が発生するたびに、その入力が正しいかを判定し、正しければ受理、間違っていればミスとする
    - このとき、キー入力に応じて予測列を更新する必要がある。例えば、「じこ」の予測列が「jiko」だったとして、プレイヤーの 1 文字目の入力が「z」だったら、予測列を「ziko」に変更するなど
- ローマ字列が単語を全て表現するところまで行ったらその単語はクリアとなる

:

さて、これらの機能を実現する方法を考えるために、さらに厳密に機能を定義していきます。

- 入力したい単語のひらがな列を H とする
- 現在入力済みのローマ字列を S とする
- プレイヤーがキーを入力するたびに、入力された文字 C が受理されるかを判定する
    - 判定方法: S の末尾に C を連結した文字列を S’ とする。H を表現するローマ字列であって、S’ を接頭辞として持つ(すなわち、S’ から始まる)ローマ字列が存在すれば受理、無ければだめ
    - 受理された場合 S を S’で置き換える
- 予測列は、H を表現するローマ字列で、S を接頭辞として持つもののうち、最も「一般的」と考えられるものとする
    - 何が一般的であるかは難しいが、何かしらの方式に基づいた一貫性を持った規則を実装すれば割となんでも良いだろう
- S が H を表現するローマ字列となったらその単語はクリア

上記の機能を実装する上で本質的な部分は、入力を受理するかの判定における「H を表現するローマ字列であって、S’ を接頭辞として持つものの存在判定」と、予測列の表示における「H を表現するローマ字列で、S を接頭辞として持つもののうち、最も「一般的」なものの取得」です。

この 2 つの機能は、後者の機能があればどちらも実現できます(ある文字列を接頭辞として持つローマ字列の存在判定は、一度取得するアルゴリズムを実行してみて、それが失敗したら存在しないということ、というふうにできる)。

なので、ここからは「あるひらがな列 H とローマ字列 S について、H を表現するローマ字列で、S を接頭辞として持つもののうち、最も「一般的」なものを求める」という問題について考えていきます。

はじめに、ローマ字に存在する変換規則を洗い出してみましょう。一般的には、以下のような規則があるでしょう:

1. a→あ、ku→く　など、標準的な規則
2. cu→く　など、同じひらがなに対し複数通り存在する規則
3. hyo→ひょ　など、拗音を含む 2 文字を一度に打つ規則
4. kka→っか　など、あるローマ字規則の 1 文字目を重ねることで促音を付ける規則
5. nta→んた　など、n などから始まらないローマ字規則の前に 1 つの n を付けて、それだけで「ん」とする規則
6. や→lya、っ→xtu　など、拗音・促音を単体で打つ規則

このように日本語のローマ字にはたくさんの規則が存在し、1 つのひらがな列に対して非常に多くのローマ字表現が存在します。

しかし、規則 4,5 を除くと、全てローマ字とひらがなの対応が単体で完全に定まっており、かつ「4,5 を除いた規則の一覧について、その規則一覧に存在するローマ字列の集合を R として、任意の文字列 X について、R の相異なる要素 i,j がどちらも X の接頭辞となることがない」という性質があり、非常に扱いやすくなります。(逆に、規則 4,5 は前後の文字に深く依存するので難しいです)

さらに、あるローマ字列をひらがなに解釈する方法は 1 通りしかないという性質もあります(これは当たり前です、普通にローマ字入力をした時に入力されるひらがなが 1 通りに定まらなかったら意味不明なので)。この 1 通りの解釈は、ローマ字列を前から見ていって変換規則に基づいて変換していくことで容易に求めることができます。

これにより、S を接頭辞として持つ、という条件は簡単に扱うことができます。S をひらがなに解釈したときのひらがな列は既に入力済みであるというマークをすればよくて、残りのひらがな列について最も一般的な表現を探す、という問題が残りました。

ここで、ローマ字列が完全にひらがなとして解釈できるとは限らなくて、あるローマ字を途中まで入力している、という状態の場合、そのローマ字から始まるという条件付きで残りのローマ字表現を探す必要があります。

なお、ひらがなに解釈する部分について厳密に手続きを書くとこんな感じになります。背景が少し灰色の部分は細かい部分なので飛ばしてもよくて、細部が気になったら読んでください:

ローマ字列 S を先頭からひらがなに変換するのは、規則のうち 4,5 を一旦無視するととても簡単です。(規則 4,5 以外は「ローマ字→ひらがな」の対応が単体で完全に定まっているのに対して、4,5 は前後に依存するので難しくなります)

4,5 を除いた規則一覧を用意しておくと、その規則一覧に存在するローマ字列の集合を R として、任意の文字列 X について、R の相異なる要素 i,j がどちらも X の接頭辞となることがないので、規則一覧を適当に見ていって、マッチする規則を適用して進めていけば良いです。

そして規則 4,5 については、規則一覧がどれも当てはまらないときに、規則 4 の判定(子音が 2 つ続いているか)と規則 5 の判定(1 つの n の次に子音が続いているか)をして、当てはまるなら 1 文字だけ消費して次に進む、とすればうまく処理できます。

このローマ字→ひらがなの変換を行っていき、生成されたひらがな列が H の接頭辞でなくなった場合(すなわち、入力したいひらがな列と食い違ってしまった場合)、失敗として結果を返します。

そして食い違わずに最後まで進んだ場合、余ったひらがな列とローマ字列がそれぞれ存在する可能性があります。それぞれが存在する/しないの 4 パターンについて、以下のように処理します。

- ひらがな→なし・ローマ字→なし
    - 単語の入力が終わったということなので、クリアという結果を返して終了する
- ひらがな→あり・ローマ字→なし
    - 入力中で、ある地点までの入力は完了しているという状態。余ったひらがな列を一般的な方法でローマ字列に変換したものを予測列とする
- ひらがな→なし・ローマ字→あり
    - 無駄に入力しているという状態。ふつう、タイピングゲームの実装でこの状態になることはないだろうが、とりあえず失敗とする
- ひらがな→あり・ローマ字→あり
    - 入力中で、かつ、ある変換規則を途中まで入力しているという状態
    - まず、「途中まで入力している変換規則」というものが何かを判定する: 規則 4,5 を除いた変換規則を適当に見ていき、「余ったローマ字列」が「変換規則のローマ字列」の接頭辞となっていて、かつ「変換規則のひらがな列」が「余ったひらがな列」の接頭辞となっているものを探す
        - 規則 4 について: 余ったひらがな列の 1 文字目が「っ」で、余ったローマ字列の長さが 1 文字で、その 1 文字が余ったひらがな列の 2 文字目(厳密には、2 文字目より後ろの文字の中で最も前にある促音でない文字、それが存在しないなら促音)の一般的なローマ字表現の 1 文字目だったら、規則 4 による変換規則を入力していると判定する
        - 規則 5 について:「ん」を打つのに規則 5 を使用するかは次の入力に完全に依存する(n を 1 文字打つか 2 文字打つかの違いなので、2 文字目になってみないとわからない)ので、この段階では普通に「nn」で入力しているという判定にしておく
        - 該当する規則が存在しなければ、間違ったアルファベットを入力したということなので、失敗として結果を返す
    - 「途中まで入力している変換規則」が判明したら、既に変換した部分とその変換規則のひらがな烈に該当する部分を入力済みとして、残りの部分について最も一般的なローマ字列を生成し、変換規則のローマ字列と合わせて予測列とする

さて、残ったのは、あるひらがな列に対して最も「一般的」なローマ字表現を取得する部分だけになりました。これはそこまで難しくありません。

「一般的」な表現というのは普通の入力方式のみを使うことがほとんどで、実際に変換規則の 1,3,4,5 のみを使用するだけで十分でしょう。これは以下のような単純なルールで構成できます。

1. 見ている文字が「ん」で、次の文字が「ん・あ・い・う・え・お」のいずれかでないなら、規則 5 に基づいて「ん」を「n」とする
2. 「ん」でなく、規則 1 で変換できる普通のひらがななならそれを採用
3. 規則 3 で変換できる拗音の 2 文字ならそれを採用
4. 促音なら、次の 1 文字(厳密には、見ていてる文字より後ろの文字の中で最も前にある促音でない文字、それが存在しないなら促音)の一般的なローマ字表現の 1 文字目を重ねて、規則 4 を使う
5. ひらがなを全て表現し終わるまで、1 から繰り返して構築していく

なお、このルールにより構成されるローマ字列は、アルゴリズムの実装者がどのような変換方式をより一般的にするかによって変わりますが、概ね一般的であるものが取得できればいいので、そこの調整は主観でやります(たぶんヘボン式とかそういうちゃんと定まったものを参考にすべきですが、めんどくさいので僕の実装では適当にしました)

これで晴れてタイピングゲームのアルゴリズム部分の実装方法がわかりました。実際にタイピングゲームを作ろうと思うと、画面、キー入力の受け付け、単語帳の作成など考えることが多いですが、割と面白いのでおすすめです。

ちなみに、僕が作ったタイピングゲームは AGC 展で展示予定なのでぜひ遊んでみてください。

また、今回紹介した方針で実装したタイピングゲーム用のローマ字パースライブラリを Rust で実装したものは github で公開しています: https://github.com/nahco314/successive_romaji

部誌

kaichou243

2024/5/2

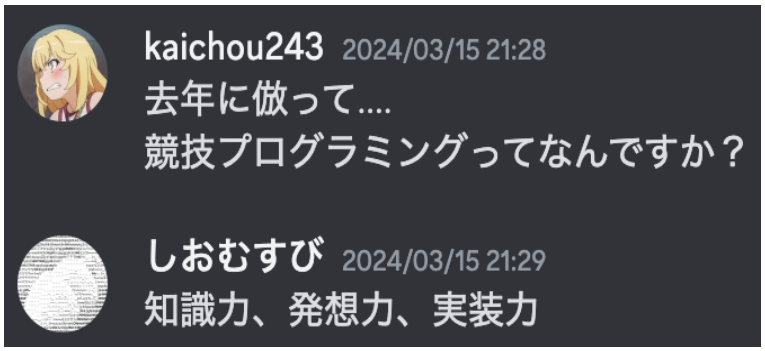
0　はじめに

ちゃんと説明する気は全くありません。ご了承ください。

1　競技プログラミング

競技プログラミングってなんですか？

第 36 回国際情報オリンピック (IOI 2024) 日本代表選手さんによると、「知識力、発想力、実装力」だそうです。



2　Aliens DP

Aliens DP とは以下のような手法です

> $[l, r]$ において下に凸な $f\colon \mathbb{Z} \to \mathbb{R} \cup \{+\infty\}$ がある。
>
> ある $k \in \mathbb{Z}$ に対して $f(k)$ を求めるために、$\min_x(f(x) - px)$ と $\arg\min_x(f(x) - px)$ を計算して p が f の点 $(k, f(k))$ における接線の傾きか判定し、二分探索によって f の点 $(k, f(k))$ における接線の傾き $\tilde{p}$ を特定する。
>
> $f(k) = \tilde{p}k + \min_x(f(x) - \tilde{p}x)$ として $f(k)$ を求める。

$f(k)$ より $\min_x(f(x)-px)$ が高速に求まる時嬉しい手法ですが、そのような場面はぱっとは思いつかないかもしれません。例を出しましょう。

AtCoder Beginner Contest 218 Red and Blue Lamps

1 から N の番号がついた N 個のランプが一列に並べられています。あなたはこのうち R 個を赤く、$N-R$ 個を青く光らせようとしています。
各 $i=1,\ldots,N-1$ について、ランプ i とランプ $i+1$ が異なる色で光っているとき、あなたは A_i の報酬を得ます。
ランプの色を適切に定めたときに得られる報酬の合計の最大値を求めてください。

$R>N/2$ なら赤と青を入れ替えることにより $R\leq N/2$ とすることができます。

このようにした時、明らかに最適解において赤が隣り合うことはありません。

したがって、この問題は「数列 $(B_1,B_2,\ldots,B_N)$ がある。この数列から隣り合う要素を選ばないように R 個要素を選ぶ方法について選んだ要素の総和の最大値を求めよ」という形の問題に帰着できます。

$f(x):=B$ から隣り合う要素を選ばないように x 個要素を選ぶ方法について選んだ要素の総和の最大値

とします f は上に凸であることが示せます。(ここでは証明は省略します。)

$f(R)$ を Aliens DP の手法以外で求める方法として、

$g(n,k):=(B_1,\ldots,B_n)$ から隣り合う要素を選ばないように k 個要素を選ぶ方法について選んだ要素の総和の最大値

としてこの $g(n,k)$ の漸化式を立てると $O(N^2)$ 時間で求めることができます。

一方、$\max_x(f(x)-px)$ は

$h(n):=(B_1-p,B_2-p,\ldots,B_n-p)$ から隣り合う要素を選ばないようにいくつか要素を選ぶ方法について選んだ要素の総和の最大値

という $h(n)$ について漸化式を立てることにより $O(N)$ 時間で求めることができます。

したがって、f は凸なので、上記にて説明した Aliens DP の手法を用いれば (上記で説明したのは f が下に凸である時のパターンですが、同じ要領でできます)$f(R)$ を $O(N\log\max A)$ 時間で求めることができます。