A　付録:ALLC ライブラリの説明

A.1　PuLP の使い方

Python で混合整数計画問題を解くためのライブラリとして *PuLP* と呼ばれるライブラリがある。このライブラリにより、*Python* のコードとして問題の作成と求解を行うことができる。また、計画問題の共通形式である *lp* ファイルとして問題を出力することができる。

今回実装したライブラリ *ALLC* はこの *PuLP* を拡張したものである。したがって、*ALLC* の説明の前に *PuLP* の使い方について説明する。なお、*Python* は *Python3* であることを仮定する。

PuLP は *Python* パッケージを管理するためのコマンドである *pip* を用いてインストールできる。

A.1.1　使い方

ここでは、*PuLP* を使って問題を解く *Python* コードの書き方を説明する。

- まず初めに *pulp* モジュールをインポートする。

```
import pulp
```

- 問題を作る。(目的関数、制約式は後で追加しておく)

```
prob = pulp.LpProblem("problem", pulp.LpMaximize)
```

なお、最小化問題の場合は第二引数に *pulp.LpMinimize* を指定する。

- 変数を定義する。
LpVariable により、新しく変数を作ることができる。指定できるパラメータは順番に、変数名、下限、上限、変数の種類を指定できる。なお、変数の種類は 3 種類指定できる。*LpContinuous,LpInteger,LpBinary* の 3 つがあり、それぞれ、実数条件、整数条件、2 値条件を表す。

```
x1 = pulp.LpVariable("x1", 0, None, pulp.LpInteger)
x2 = pulp.LpVariable("x2", 0, None, pulp.LpInteger)
```

- 目的関数、制約式を設定する。
pulp では、問題に対して”+=”演算子を使うことで、目的関数と制約式を追加できる。なお、その追加された式が一次式ならば、目的関数として追加され、不等式か等式ならば、制約式として追加される。

```
prob += 5.5*x1 + 2.1*x2

prob += -x1 + x2 <= 2
prob += 8*x1 + 2*x2 <= 17
```

- 求解, 解の存在性
  問題に対し、*solve* メソッドを用いてその問題の解を求めることができる。また、このメソッドの返り値を *LpStatus* にキーとして与えると、問題の解の状況がわかる。

```
status = prob.solve()
print(pulp.LpStatus[status])
```

以下にソースコード例と、その出力を示す。

```
import pulp

prob = pulp.LpProblem("problem", pulp.LpMaximize)

x1 = pulp.LpVariable("x1", 0, None, pulp.LpInteger)
x2 = pulp.LpVariable("x2", 0, None, pulp.LpInteger)

prob += 5.5*x1 + 2.1*x2

prob += -x1 + x2 <= 2
prob += 8*x1 + 2*x2 <= 17

status = prob.solve()
print(pulp.LpStatus[status])

print("answer is {}".format(prob.objective.value()))
print("x1 is {}".format(x1.value()))
print("x2 is {}".format(x2.value()))
```

```
Optimal
answer is 11.8
x1 is 1.0
x2 is 3.0
```

図 11: 出力結果

A.2　導入方法

ALLC は *GitHub* 上でソースコードを公開している。この *URL* は
https://github.com/KyushuUniversityMathematics/ALLC
である。この *URL* でソースコードをダウンロードすれば（本体はこのフォルダ内の *allc* フォルダ）*ALLC* ライブラリは使用可能である（このフォルダ内の *src_and_sample* フォルダにソースとサンプルが含まれている。このフォルダ以外は旧 *Java* バージョンのものであるので削除しても構わない。また、事前に *PuLP* はインストールしておく必要がある。*)* ライブラリをどのディレクトリからもインポートできるようにするためには環境変数 *PYTHONPATH* を設定しておけばよい。例えば、*mac* ならばホームディレクトリにある *.bash_profile* ファイルの末尾に以下の文字列を書いておけばよい。
export PYTHONPATH=$PYTHONPATH:'(allc-path)'
なお、上で *(allc-path)* と書いてある部分は *allc*（*src_and_sample* フォルダの中の *allc* フォルダのこと）が置いてあるディレクトリのパスを入れる。
ALLC は計 *787* 行のソースコードからなり、*PuLP* で制約式に使える演算子、関数に加えて、*19* 種類の演算子や関数を制約式で扱うことができる。

A.3　ALLC の使い方

今回作成した *ALLC* も基本的には *PuLP* と同じような使い方である。しかしながら、いくつか異なる部分がある。まずは *ALLC* での問題を解く手順を説明する。そのあと *ALLC* で新しく追加された関数や演算子について説明する。

A.3.1　問題を解く手順

ここでは次の問題を解くプログラムの作成を例に問題を解く手順を説明する。
（問題）
A E の五人が *50m* 競争をし、その結果について次のように話した。ただし、本当のことを言っているのは *5* 位の人だけで、残りの *4* 人は嘘を言っている。同じ順位の人はいなかったとして、各メンバーの順位は何位か？

- *A*「私は *E* より順位が下だった。」
- *B*「*C* が一位だった。」
- *C*「私は一位ではない。」
- *D*「*C* が五位だった。」
- *E*「私は *D* より順位が下だった。」

(手順)

- まず初めに *allc* モジュールをインポートする。

```
from allc import *
```

- 問題を作る。(目的関数、制約式は後で追加する)

```
prob = LpProblem('Race␣Problem', LpMinimize)
```

- 変数を定義する。
  Variable により (*LpVariable* でないことに注意)、新しく変数を作ることができる。指定できるパラメータは順番に、変数名、下限、上限、変数の種類を指定できる。なお、変数の種類は 3 種類指定できる。*LpContinuous,LpInteger,LpBinary* の 3 つがあり、それぞれ、実数条件、整数条件、2 値条件を表す (*pulp* と同様に *Variable* クラスも *static* メソッドとして *dicts* メソッドを実装している。このメソッドの使い方を知りたい場合は *pulp* での *LpVariable* の *dicts* メソッドについて調べて頂きたい)。なお、ここでは各変数は各メンバーの順位を表すものとしている。

```
A = Variable('A', 1, 5, LpInteger)
B = Variable('B', 1, 5, LpInteger)
C = Variable('C', 1, 5, LpInteger)
D = Variable('D', 1, 5, LpInteger)
E = Variable('E', 1, 5, LpInteger)
```

- *LogicTranslateor* クラスのオブジェクトを作る。

```
lg = LogicTranslator()
```

- 変換オブジェクトに制約式を渡す。
  allc では、*LogicTranslateor* クラスのオブジェクトに対して"+="演算子を使うことで、制約式を追加できる (目的関数は追加できないことに注意)。なお、このコードの前半では各メンバーの順位が異なるという条件を加えている。後半では、各メンバーに対してその人が 5 位であることとその人の発言が真であることが同値であるという条件を加えている。

```
member = [A, B, C, D, E]
lg += forall(range(5), lambda s:
        forall(range(s+1,5), lambda t:
            member[s] != member[t] ))

```

```
lg += (A == 5) == (A > E)
lg += (B == 5) == (C == 1)
lg += (C == 5) == (C != 1)
lg += (D == 5) == (C == 5)
lg += (E == 5) == (E > D)
```

- 目的関数、制約式の設定問題に目的関数と、制約式を設定する。*LogicTranslater* クラスのオブジェクトに対して *for* ループを用いることで変換後の制約式を順次取得することができる。これを用いて *prob* に制約式を追加してゆく。目的関数に関しては、*pulp* のときと同様に式が一次式の場合に限り目的関数として解釈される。ここでは目的関数はなんでも良いので *0* を設定している。

```
prob += 0

for e in lg:
    prob += e
```

- 求解, 解の存在性
  pulp と同様に問題に対し、*solve* メソッドを用いてその問題の解を求めることができる。また、このメソッドの返り値を *LpStatus* にキーとして与えると、問題の解の状況がわかる。

```
status = prob.solve()
print(pulp.LpStatus[status])
```

よってこれらを合わせると *python* コード全体は次のようになる。

```
from allc import *

prob = LpProblem('Race␣Problem', LpMinimize)

A = Variable('A', 1, 5, LpInteger)
B = Variable('B', 1, 5, LpInteger)
C = Variable('C', 1, 5, LpInteger)
D = Variable('D', 1, 5, LpInteger)
E = Variable('E', 1, 5, LpInteger)

lg = LogicTranslator()

member = [A, B, C, D, E]

lg += forall(range(5), lambda s:
```

```
        forall(range(s+1,5), lambda t:
            member[s] != member[t] ))

lg += (A == 5) == (A > E)
lg += (B == 5) == (C == 1)
lg += (C == 5) == (C != 1)
lg += (D == 5) == (C == 5)
lg += (E == 5) == (E > D)

prob += 0

for e in lg:
    prob += e

prob.solve()

print("Status:", LpStatus[prob.status])

for m in member:
    print('The order of {} is {}'.format(m.getName() ,int(m.
      value())))
```

```
Status: Optimal
The order of A is 2
The order of B is 5
The order of C is 1
The order of D is 4
The order of E is 3
```

図 12: レース問題プログラムの実行結果

A.3.2 ALLC で使用可能な関数

ALLC では *PuLP* で扱えた関数や演算子に加え、以下の関数も新たに使えるようになっている。一部の演算には演算子のオーバーロードを用いて演算子を使うことができる。また、一部の関数名は *Python* の組み込み関数名との衝突を避けるためにアンダーバーをつけている。

名称	記号	関数名	説明、補足
不等号	<		等号を含まない不等号 (逆向きも使用可)
等号否定	!=		等しくないことを表す
論理否定	~	not_	論理否定
論理積	&	and_	論理積 （関数の方は可変長引数対応）
複数個の論理積		all_	要素全ての論理積をとる (引数は有限リスト)
論理和	|	or_	論理和 （関数の方は可変長引数対応）
複数個の論理和		any_	要素 全ての論理和をとる (引数は有限リスト)
含意演算	>>	imply_	含意
全称記号		forall	有限個の範囲しか扱えない。
存在記号		exists	有限個の範囲しか扱えない。個数を指定可能
切り捨て除算	//		除数は定数でなければならない
剰余	%		除数は定数でなければならない
床関数		floor_	床関数
天井関数		ceil_	天井関数
絶対値		abs_	絶対値
最大値		max_	最大値（可変長引数対応）
最小値		min_	最小値（可変長引数対応）
総和		sum_	総和 （引数は有限リスト）
条件演算子		cond	ブール値による値の切り替え

なお、注意点として各演算子は *python* での演算子のオーバーロードにより実装されていることに注意。したがって、演算子の優先順位には注意が必要。例えば以下の式のカッコは省略不可能であることに注意（なぜなら、論理和の演算子は元はビット演算の記号として使っていたので比較演算より優先順位が高くなっている)。また、含意記号が右結合でないことにも注意。

```
1 (a==5) | (b<=3)
```

また、このことにより引数が全て定数の時には予想とは異なる挙動になってしまう。例えば、$False \Rightarrow False$ を意味する以下の式はビット演算の右シフト演算として解釈され、$False$ を意味する *0* となってしまう。このことは論理否定記号を使用する時にも注意が必要である（ビット反転になってしまう)。

```
1 0 >> 0
```

(関数の説明)
いくつかの関数の説明を述べる。

- *forall(l, f)*
  第一引数はリスト（正確にはイテラブルなオブジェクトなら良い)。
  第二引数は第一引数の要素を引数に取り論理式を返す関数。

lの各要素にfを適用してできる論理式たちを論理積で結んだ論理式を返す。

```
forall([1,2,3], f) # f(1) & f(2) & f(3)
```

- *exists(l, f)*
  exists は引数の個数により動作が異なる。引数が二つの場合は *forall* と同様であり、論理積が論理和に変わるだけである。

```
exists([1,2,3], f) # f(1) | f(2) | f(3)
```

- *exists(n)(l,f)*
  n は整数を表す式
  exists の引数が一つの場合は関数を返し、その関数は l の各要素に f を適用してできる論理式たちの中で真になるものが n 個であることを表す論理式を返す。

```
exists(2)([1,2,3,4], f) # |{ f(i) : i in [1,2,3,4] }| == 2
```

- *cond(c, t, f)*
  式 c が真の時は式 t になり、c が偽になる時は f になる式を表す。

```
cond(a==0, b, c) # b if a==0 else c
```

- *LogicTranslator* クラスの *writeTable(filename)* メソッド
  ファイル名 *filename* のファイルに各一時変数が表現する論理式のリストを書き込む。各変数がどのように生成されたか確認する時に便利である。

```
lg.writeTable('table.txt')
```

問題の引用元
SPIノートの会. Webテスト突破法. 洋泉社. p79 改　2015年