Gene Expression Programming

Wprowadzenie

Pan Ćwikowski
Pan Caputa

7.02.2020

Wprowadzenie

Podczas gdy algorytmy genetyczne i programowanie genetyczne są dobrze znane w literaturze, programowanie ekspresji genów (Ferreira [31], Hardy i Steeb [42]) nie jest jeszcze dobrze znane. Programowanie ekspresji genów jest algorytmem genomu / fenomu, który łączy w sobie prostotę algorytmów genetycznych i możliwości programowania genetycznego. W pewnym sensie programowanie ekspresji genów jest uogólnieniem algorytmów genetycznych i programowania genetycznego. Programowanie ekspresji genów różni się od programowania genetycznego, ponieważ wyeliminowano koszt zarządzania strukturą drzewa i zapewnienia poprawności programów. Poniżej przedstawiamy wprowadzenie

Coś o grafach

Gen jest symbolicznym sznurkiem z głową i ogonem. Każdy symbol reprezentuje operację. Operacja + pobiera dwa argumenty i dodaje je. Na przykład + x2 to operacja + z argumentami x i 2, dającymi x + 2. Operacja * bierze również dwa argumenty i mnoży je. Operacja x oszacowałaby do wartości zmiennej . Ogon składa się tylko z operacji, które nie przyjmują argumentów. Ciąg reprezentuje wyrażenia w notacji przedrostka, tj. 5-3 będzie przechowywany jako - 5 3. Powodem ogona jest zapewnienie, że wyrażenie jest zawsze pełne. Załóżmy, że łańcuch ma symbole h w głowie, które są określone jako dane wejściowe do algorytmu, oraz t symbole w ogonie, który jest określony od h. Zatem jeśli n jest maksymalną liczbą argumentów dla operacji, którą musimy mieć

$$h + t - 1 = hn\,.$$

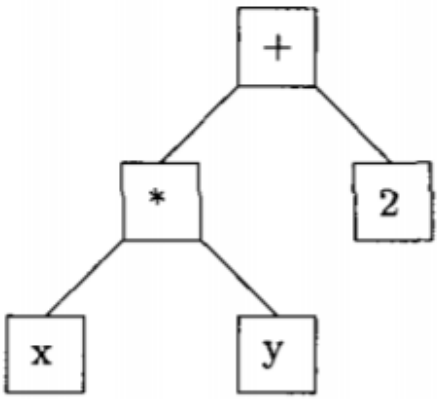
Lewa strona to łączna liczba symboli, z wyjątkiem pierwszego symbolu. Prawa strona to całkowita liczba argumentów wymaganych dla wszystkich operacji. Zakładamy oczywiście, że każda operacja wymaga maksymalnej liczby argumentów, aby każdy ciąg tej długości był prawidłowym ciągiem dla wyrażenia. Zatem równanie stwierdza, że musi istnieć wystarczająca liczba symboli, aby służyć jako argumenty dla wszystkich operacji. Teraz możemy określić wymaganą długość ogona t = hin - 1) + 1.

Przykład 1.

Rozważ symbole x, y i operacje *, +, które przyjmują dwa argumenty. Niech h = 5. Wtedy t = 6, ponieważ n = 2. Zatem gen

+*xy2|xyy32y

byłby Linia pionowa wskazuje początek ogona. Wyrażenie to (x * y) +2. Struktura drzewa byłaby



Przykład 2

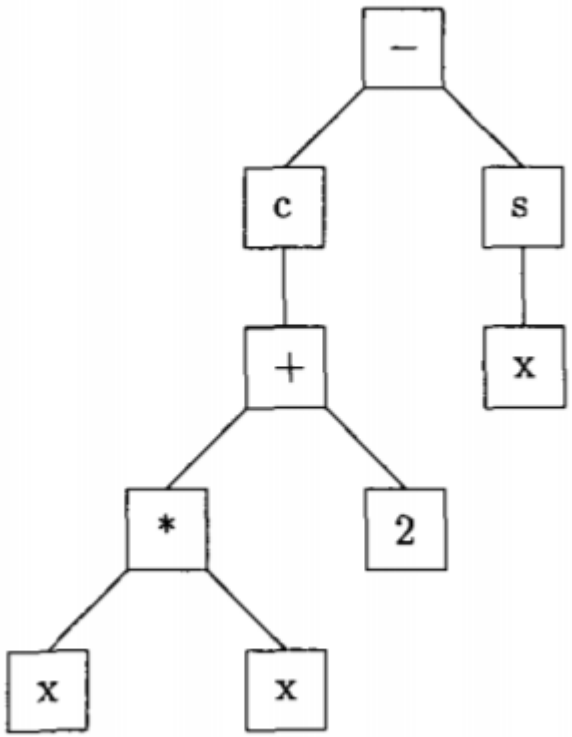
Załóżmy, że używamy h = 8, a n = 2 dla operacji arytmetycznych. Zatem długość ogona musi wynosić t = 9. Zatem całkowita długość genu wynosi 17. Możemy wtedy przedstawić ekspresję

$$\cos(x^2 + 2) - \sin(x)$$

with the string

`-c+*xx2s|x1x226x31`

Pionowy | służy do wskazania początku ogona. Tutaj c oznacza cos, a s oznacza grzech. Możemy reprezentować wyrażenia za pomocą drzew. W powyższym przykładzie rdzeniem drzewa byłoby „-” z gałęziami dla parametrów. W ten sposób możemy przedstawić wyrażenie w następujący sposób.



Pamiętaj, że nie wszystkie symbole z ogona są używane. W niniejszym przykładzie użyto tylko jednego symbolu z ogona.

Wprowadzenie

Chromosom to zbiór genów. Geny łączą się, tworząc ekspresję za pomocą operacji z taką samą liczbą argumentów, jak geny w chromosomie. Na przykład ekspresje genów chromosomu można dodawać razem. W przypadku operacji zastosowanych do chromosomów często łączymy geny w celu uzyskania pojedynczego ciągu symboli. Załóżmy na przykład, że mamy następujące geny

```
-c+*xx2s|x1x226x31
-c+*xx2s|x1x226x31
-c++x2*x|x1x226x31
```

tworzące chromosom tworzymy chromosom przez konkatenację `-c+*xx2s|x1x226x31|-c+*xx2s|x1x226x31|-c++x2*x|x1x226x31` gdzie | wskazuje początek części głowy i ogona genów. Stosujemy operacje do chromosomów w zbiorze zwanym populacją chromosomów.

Chromosomy mają wiele operacji.

• Replikacja. Chromosom pozostaje niezmieniony. Wybór koła ruletki można zastosować technikę do wyboru chromosomów do replikacji. Mutacja. Losowo zmieniaj symbole w chromosomie. Symbole w ogonie genu mogą nie działać na żadnych argumentach. Zazwyczaj stosuje się 2 punktowe mutacje na chromosom. Na przykład niech + * xy2 xyy32y. Wtedy mutacja może dać + * yy2 | xyy32y lub ++ xy2 | xyy32y lub + * xy- | xyy32y. Wprowadzenie. Część chromosomu jest wybrana do wstawienia do głowy genu. Ogon genu pozostaje nienaruszony. W ten sposób symbole są usuwane z końca głowy, aby zrobić miejsce dla wstawionego sznurka. Zazwyczaj stosuje się prawdopodobieństwo wprowadzenia 0,1. Jako przykład załóżmy, że należy wstawić + x2-C + * xx2s | x1x226x31 na czwartej pozycji w głowie. Otrzymujemy -C ++ x2 * x1x1x226x31 który reprezentuje wyrażenie cos ((x + 2) + x2) - 1.

• Transpozycja genów. Losowo wybiera się jeden gen w chromosomie pierwszy gen. Wszystkie inne geny w chromosomie są przesunięte w dół na chromosomie, aby zrobić miejsce dla pierwszego genu. • Transpozycja genów. Jeden gen w chromosomie jest losowo wybierany jako pierwszy gen. Wszystkie inne geny w chromosomie są przesunięte w dół na chromosomie, aby zrobić miejsce dla pierwszego genu.

• Rekombinacja. Operacja crossover. Może to być jeden punkt (chromosomy są podzielone na dwie części i odpowiednie sekcje są zamienione), dwa punkty (chromosomy są podzielone na trzy części środkowe są zamienione) lub gen (jeden cały gen jest zamieniony między chromosomami) rekombinacja. Zazwyczaj suma prawdopodobieństw rekombinacji wynosi 0,7. Kolejną operacją może być zamiana. Na przykład rozważmy ponownie + * xy2 xyy32y. Następnie zamieniając drugą i siódmą pozycję (licząc od lewej i zaczynając od zera) otrzymujemy + * yy2 xyx32y. W poniższym przykładzie implementujemy te operacje. Dla uproszczenia używamy tylko jednego genu w chromosomie i tylko jednej rekombinacji punktowej.

Przykład

W naszym przykładzie rozważamy gładkie jednowymiarowe mapy f: R -> R posiadające właściwości

$$f(0) = 0, \quad f(1) = 0, \quad f\left(\frac{1}{2}\right) = 1$$

$$\frac{1}{2} < f\left(\frac{1}{4}\right) < 1, \qquad \frac{1}{2} < f\left(\frac{3}{4}\right) < 1\,.$$

Na przykład mapa logistyczna g (x): = 4x (1 - x) spełnia ten zestaw właściwości. Używamy programowania wyrażeń genowych do przeprowadzania regresji symbolicznej w celu uzyskania map spełniających powyższe właściwości. Oczekujemy, że mapa logistyczna powinna zostać znaleziona, ponieważ ograniczamy funkcje w implementacji do wielomianów. Generalizujemy zestaw właściwości w następujący sposób. Punkty oceny są określone przez podzbiór XY, gdzie X i Y są podane przez f: X + Y. Oznacz przez F-CXXY podzbiór wszystkich (x, y) EXXY tak, że wymagamy od tego f (x) = y, przez FCXXY podzbiór wszystkich (x, y) EXXY taki, że wymagamy f, że f (x)> y, przez Fc CX * Y podzbiór wszystkich (2x, y) EX * Y taki, że wymagamy f, że f (x) <y. W ten sposób możemy zdefiniować funkcję sprawności funkcji g (gdzie mniejsza wartość oznacza wyższą sprawność)

$$\text{fitness}(g) := \sum_{(x,y)\in F_=} |g(x) - y| + \sum_{(x,y)\in F_>} |g(x) - y| H(y - g(x)) + \sum_{(x,y)\in F_<} |g(x) - y| H(g(x) - y)$$

where

$$H(x) := \begin{cases} 1 \, x > 0 \\ 0 \, \text{otherwise} \end{cases}$$

is the step function.

We apply gene expression programming until we find a function g such that

$$\text{fitness}(g) < \epsilon$$

for given $\epsilon > 0$. A typical value of ϵ is 0.001.

Przykładowy kod

```
// gepchaos.cpp

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>    // for sin, cos
#include <cstring>
using namespace std;

const double pi = 3.1415927;
const int nsymbols = 5;
// 2 terminal symbols (no arguments) x and 1
const int terminals = 2;
// terminal symbols first
```

```
const char symbols[nsymbols] = {'1','x','+','-','*'};
const int n = 2;   // for +,- and * which take 2 arguments
int h = 10;

double evalr(char *&e,double x)
{
  switch(*(e++))
  {
  case '1': return 1.0;
  case 'x': return x;
  case 'y': return pi*x;
  case 'c': return cos(evalr(e,x));
  case 's': return sin(evalr(e,x));
  case '+': return evalr(e,x)+evalr(e,x);
  case '-': return evalr(e,x)-evalr(e,x);
  case '*': return evalr(e,x)*evalr(e,x);
  default : return 0.0;
 }
}

double eval(char *e,double x)
{
  char *c = e;
  return evalr(c,x);
}
```

```
void printr(char *&e)
{
  switch(*(e++))
  {
  case '1': cout << '1'; break;
  case 'x': cout << 'x'; break;
  case 'y': cout << "pi*x"; break;
  case 'c': cout << "cos("; printr(e); cout << ")"; break;
  case 's': cout << "sin("; printr(e); cout << ")"; break;
  case '+': cout << '('; printr(e); cout << '+'; printr(e);
            cout << ')'; break;
  case '-': cout << '('; printr(e); cout << '-'; printr(e);
            cout << ')'; break;
  case '*': cout << '('; printr(e); cout << '*'; printr(e);
            cout<<')'; break;
  }
}

void print(char *e)
```

```
{
  char *c = e;
  printr(c);
}

double fitness(char *c,double *data,int N)
{
  double sum = 0.0;
  double d;
  for(int j=0;j<N;j++)
  {
  d = eval(c,data[3*j])-data[3*j+1];
  if(data[3*j+2] == 0) sum += fabs(d);
  else if(data[3*j+2] > 0) sum -= (d > 0.0)?0.0:d;
  else if(data[3*j+2] < 0) sum += (d < 0.0)?0.0:d;
  }
  return sum;
}

// N number of data points
// population of size P
// eps = accuracy required
void gep(double *data,int N,int P,double eps)
{
```

```
int i,j,k,replace,replace2,rlen,rp;
int t = h*(n-1)+1;
int gene_len = h+t;
int pop_len = P*gene_len;
int iterations = 0;
char *population = new char[pop_len];
char *elim = new char[P];
int toelim = P/2;
double bestf,f;        // best fitness, fitness value
double sumf = 0.0;     // sum of fitness values
double pmutate = 0.1; // probability of mutation
double pinsert = 0.4; // probability of insertion
double precomb = 0.7; // probability of recombination
double r,lastf;        // random numbers and roulette wheel selection
char* best = (char*)NULL; // best gene
char* iter;             // iteration variable

// initialize the population
for(i=0;i<pop_len;i++)
if(i%gene_len < h) population[i] = symbols[rand()%nsymbols];
else population[i] = symbols[rand()%terminals];
```

```
// initial calculations
bestf = fitness(population,data,N);
best = population;
for(i=0,sumf=0.0,iter=population;i<P;i++,iter+=gene_len)
{
f = fitness(iter,data,N);
sumf += f;
if(f<bestf)
{
bestf = f;
best = population+i*gene_len;
}
}

while(bestf >= eps)
{
// reproduction
// roulette wheel selection
for(i=0;i<P;i++) elim[i] = 0;
for(i=0;i<toelim;i++)
{
r = sumf*(double(rand())/RAND_MAX);
lastf = 0.0;
for(j=0;j<P;j++)
{
f = fitness(population+j*gene_len,data,N);
if((lastf<=r) && (r<f+lastf))
```

```
{
elim[j] = 1;
j = P;
}
lastf += f;
}
}

for(i=0;i<pop_len;)
{
if(population+i == best)
 i += gene_len; // never modify/replace best gene
 else for(j=0;j<gene_len;j++,i++)
 {
 // mutation or elimination due to failure in selection
 // for reproduction
 if((double(rand())/RAND_MAX < pmutate) || elim[i/gene_len])
```

```
if(i%gene_len < h)
population[i] = symbols[rand()%nsymbols];
else population[i] = symbols[rand()%terminals];
}

// insertion
if(double(rand())/RAND_MAX < pinsert)
{
// find a position in the head of this gene for insertion
// -gene_len for the gene since we have already moved
// to the next gene
replace = i-gene_len;
rp = rand()%h;
// a random position for insertion source
replace2 = rand()%pop_len;
// a random length for insertion from the gene
rlen = rand()%(h-rp);
// create the new gene
char *c = new char[gene_len];
// copy the shifted portion of the head
strncpy(c+rp+rlen,population+replace+rp,h-rp-rlen);
// copy the tail
strncpy(c+h,population+replace+h,t);
// copy the segment to be inserted
strncpy(c+rp,population+replace2,rlen);
// if the gene is fitter use it
if(fitness(c,data,N) < fitness(population+replace,data,N))
strncpy(population+replace,c,h);
delete[] c;
}
```

```
// recombination
if(double(rand())/RAND_MAX < precomb)
{
// find a random position in the gene for one point recombination
replace = i-gene_len;
rlen = rand()%gene_len;
// a random gene for recombination
replace2 = (rand()%P)*gene_len;
// create the new genes
char *c[5];
c[0] = population+replace;
c[1] = population+replace2;
c[2] = new char[gene_len];
c[3] = new char[gene_len];
```

```
 c[4] = new char[gene_len];
 strncpy(c[2],c[0],rlen);
 strncpy(c[2]+rlen,c[1]+rlen,gene_len-rlen);
 strncpy(c[3],c[1],rlen);
 strncpy(c[3]+rlen,c[0]+rlen,gene_len-rlen);
 // take the fittest genes
for(j=0;j<4;j++)
for(k=j+1;j<4;j++)
 if(fitness(c[k],data,N) < fitness(c[j],data,N))
 {
 strncpy(c[4],c[j],gene_len);
 strncpy(c[j],c[k],gene_len);
 strncpy(c[k],c[4],gene_len);
 }
delete[] c[2]; delete[] c[3]; delete[] c[4];
}
}
```

```
  // fitness
  for(i=0,sumf=0.0,iter=population;i<P;i++,iter+=gene_len)
  {
  f = fitness(iter,data,N);
  sumf += f;
  if(f < bestf)
  {
  bestf = f;
  best = population+i*gene_len;
  }
  }
  iterations++;
  }

  print(best);
  cout << endl;
  cout << "Fitness of " << bestf << " after "
       << iterations << " iterations." << endl;

  delete[] population; delete[] elim;
}

int main(void)
{
  srand(time(NULL)); // seed for the random number generator
  double data[] = { 0,0,0,1,0,0,0.5,1,0,0.25,0.5,1,0.75,0.5,1,
                    0.25,1, -1,0.75,1, -1 };
```

```
  gep(data,7,30,0.001);
  cout << endl;
  return 0;
}
```

We list some typical results below, all with fitness 0.

$((1-x)*((x+(((x-x)+x)+x))+x)) = 4x(1-x)$
$((x+(((((((x-x)-x)-x)*x)-x)*x)+x))+x) = x(1-x)(2x+3)$
$(x-(((x+x)*((x-1)+x))-x)) = 4x(1-x)$
$((((1-x)*1)*(((1+1)+1)+1))*x) = 4x(1-x)$
$((((1+1)-(((x+x)*x)+x))*x)+x) = x(1-x)(2x+3)$
$((1*((x-((((x+x)*x)+x)*x))+x))+x) = x(1-x)(2x+3)$
$((1-x)*((((((x+x)*x)+x)*1)+x)+x)) = x(1-x)(2x+3)$
$((((1-(1*(((x+x)*x)-1)))-x)+1)*x) = x(1-x)(2x+3)$

Okazuje się, że przez większość czasu 4x (1 - x) lub X (1 - 2) (2x +3) jest najsilniejszą mapą. Oczywiście znajdziemy pożądane funkcje z mniejszą liczbą iteracji, jeśli zwiększy się rozmiar nalewania. Mapa g (x) = x (1 - x) (2x + 3) spełnia powyższe warunki, ale g (x) ma wartości większe niż 1 w przedziale

$$\left(\frac{1}{2}, \frac{\sqrt{5}}{2} - \frac{1}{2}\right).$$

Stwierdzamy zatem, że prawie wszystkie wartości początkowe do (0, 1) unikają przedziału [0, 1] pod iteracjami mapy. Zbiór wszystkich punktów, których iteracja pozostaje w [0, 1], ma miarę zero i tworzy zbiór Cantora. Możemy także zmienić tablicę symboli, tak aby zawierała symbole c dla cosinusa, s dla sinusa i y dla 7X. W tym przypadku funkcja sin (7x) znajduje się prawie za każdym razem jako najlepsza mapa. Innym zastosowaniem programowania ekspresji genów jest znalezienie ekspresji boolowskich. W wyrażeniach boolowskich mamy operację AND, OR, XOR i NOT. Bramka NAND jest bramką AND, po której następuje bramka NOT. Brama NAND jest bramą uniwersalną, z której można zbudować wszystkie inne bramy.

Jako przykład rozważmy tabelę prawdy

gdzie a i b są wejściami, a O jest wyjściem

a	b	O
0	0	1
0	1	1
1	0	0
1	1	1

Koniec

Prezentacje Przygotowali:
Pan Ćwikowski
Pan Caputa