IBM

Cultured Perl: Генетические алгоритмы, выполненные на Perl

Создайте ваши собственные Дарвинские питомники

Уровень сложности: простой

Теодор Златанов, программист, Gold Software Systems

12.07.2007

Генетическое программирование, основанное на дарвинском принципе выживания наиболее приспособленных, использует мутацию и репликацию при создании алгоритмов для создания постоянно улучшающихся компьютерных программ. В этой статье понятным языком изложено, что такое генетический алгоритм. Тед показывает, как реализовать на Perl некоторые специфические задачи, которые вы можете адаптировать для общего использования. Чтобы продемонстрировать генетический алгоритм, Тед "размножает" числа для того, чтобы они удовлетворяли формулам, и буквы, чтобы составлять английские слова.

Вы можете запускать примеры из этой статьи, если в вашей системе есть Perl версии 5.005 или более поздней. Желательно, чтобы у вас была современная (2000 или более поздняя) UNIX-система (Linux, Solaris, BSD), но также подойдут и другие типы операционных систем. Примеры могут быть выполнены и на более ранних версиях Perl и UNIX, а их сбои в функционировании можно рассматривать как упражнения для читателей.

История

Прогресс в генетике в 20 веке соперничает в скорости только с эволюцией электроники и вычислительной техники. Именно поэтому одним из наиболее захватывающих алгоритмов 20 века является генетический алгоритм.

Появившийся в начале 1960-х генетический алгоритм (и вообще, эволюционный алгоритм) занял в компьютерных науках место между детерминированными и недетерминированными алгоритмами. Генетический алгоритм является, по существу, настолько детерминированным, насколько вы хотите его таковым сделать, что означает, что пользователи могут сами определить количество итераций и критерии завершения. Он имитирует дарвиновский естественный отбор, делая "приспособленность" (определяемую формулой, применяемой к индивидууму) вместе с мутацией главным критерием отбора индивидуума на выживаемость и воспроизведение потомства.

Другие эволюционные алгоритмы пытаются имитировать эволюцию по Ламарку, в которой поведение как механизм выживания может передаваться между поколениями, и даже есть эволюционные программы, которые сами себя пишут для каких-то конкретных целей. Все это выходит за рамки данной статьи.

Основной недостаток Perl для генетических алгоритмов -- это скорость. Из-за необходимости вычислений в генетических алгоритмах они более эффективны на C или других компилируемых языках низкого уровня. Показанные ниже примеры на Perl не обладают таким быстродействием, как их эквиваленты на C, но эти примеры покажут вам, как работает генетический алгоритм, и они обладают достаточным быстродействием для некоторых задач.

Итак, что такое генетический алгоритм?

Генетический алгоритм достаточно прост для понимания при использовании биологических терминов, которым обучают в средней школе. Возьмем популяцию индивидуумов, причем каждый имеет свою собственную ДНК. Затем измеряем приспособленность каждого индивидуума (измеряется как функция, применяемая к ДНК индивидуума) и считаем, что индивидуум с большей вероятностью произведет потомство, если его приспособленность выше. Индивидуумы с очень малой приспособленностью уничтожаются. Каждый выживший получает возможность произвести потомство (и важно, что это никогда не запрещено выжившим, только оно

менее вероятно для тех, у кого приспособленность ниже). Слияние родительских ДНК и применение затем случайных мутаций к слившимся ДНК моделирует процесс производства потомства. Новый индивидуум будет теоретически таким же приспособленным, как родители, плюс-минус небольшие вариации, вызванные мутациями. Затем цикл повторяется.

Очевидно, что существует множество переменных, которые могут влиять на генетический алгоритм: это размер популяции, число поколений (итерации в алгоритме), методы слияния, функция приспособленности, то, как влияет приспособленность на шансы для производства потомства и сколько произойдет мутаций.

Часто задаваемые вопросы по генетическим алгоритмам
Смотрите раздел в Ресурсы по часто задаваемым вопросам о генетических алгоритмах. В FAQ есть алгоритм GA (генетический алгоритм), и обращено внимание на комплект программного обеспечения по генетическому алгоритму, как бесплатному, так и коммерческому.

Также существуют и некоторые недостатки алгоритма. Он лучше работает, если функция приспособленности применяется к ДНК как набор битов. Иными словами, лучше, если ДНК представляет собой набор бинарных опций: да или нет. Голубые глаза? Карие глаза? Рыжие волосы? Черные волосы? Слияние родительских ДНК и последующие мутации не должны разрешать некоторые комбинации битов, так как получающаяся в результате ДНК не будет допустимым решением исходной задачи. Вспомним, что "ДНК" является ни чем иным, как решением для математической формулировки приспособленности. Некоторые величины, используемые в этой формулировке, могут быть несправедливыми – например, деление на ноль.

Кроме того, генетический алгоритм не связан со временем. Вы отбираете количество поколений. Вы можете определить некоторую цель – например, "найти индивидуума с приспособленностью 0.99999" и остановиться, но тогда алгоритм никогда не закончит работу, так как он не сможет найти такого индивидуума. Если вы ставите нереальные цели или задаете слишком мало поколений, то могут возникнуть проблемы. Метод проб и ошибок и хорошее чутье являются лучшими помощниками в этой области.

Формула приспособленности должна выдавать число с плавающей точкой от 0 до 1. Могут быть использованы и другие диапазоны, но, по моему опыту, числа с плавающей точкой работают лучше. Вы можете захотеть интервал от 0 до 32767, если вы хотите для оптимизации использовать 7-битовое число для приспособленности.

Конечно, отсрочка оптимизации, пока вы знаете, что она нужна, является хорошей идеей, так что вам следует, по крайней мере, начинать с простой формулы приспособленности. Формула приспособленности является наиболее используемой функцией в генетическом алгоритме (она вызывается (размер популяции)x(количество поколений) раз), так что вы должны сделать ее как можно более простой и быстродействующей.

Существуют три "хороших" способа выхода из генетического алгоритма. Во-первых, вы можете решить выйти из алгоритма, если больше нет разнообразия в фонде ДНК. Это, в действительности, хитрый тест, и модуль CPAN для нахождения протяженности строк может быть полезным в случае, если вы можете представлять ДНК как строку. Во-вторых, вы можете выйти, если вы достигли заданной приспособленности. Пока вы хорошо не поймете формулу приспособленности (для этого вам вообще может быть не нужен генетический алгоритм) постановка задач по приспособленности будет приводить к одному из двух результатов: или бесконечные циклы, или индивидуум, который является всего лишь "довольно хорошим". В третьих, вы можете выйти после заданного числа итераций или "поколений".

На практике все эти три способа (или, по крайней мере, второй и третий) используются для контроля генетического алгоритма. Небольшое число прогонов, возможно 10 или 20, даст вам правильное ощущение того, как быстро сходится алгоритм и какую степень приспособленности вы можете ожидать.

Простой пример

Код в Листинге 1 использует единичный байт как ДНК (величина между 0 и 255, 8 бит). Формула приспособленности применяется один раз к каждому новому индивидууму, для этого берется значение байта ДНК и делится на 256. Это означает, что формула приспособленности всегда будет выдавать число между 0 и 255/256, так что оно никогда не достигнет 1. Как вы думаете, какая ДНК будет наиболее приспособленной?

Листинг 1. numbers.pl

```
#!/usr/bin/perl -w

# Демонстрация генетического алгоритма (ГА) с численной ДНК (от 0 до 255)

use strict;
use Data::Dumper;

# индивидуумы в популяции – нет смысла делать больше, чем может обеспечить ДНК
my $popsize = 256;
my $mut_rate = 0.01;                    # скорость мутации
my $min_fitness = 0.1;                  # минимальная приспособленность для выживания
my $generation_count = 100000;          # выполнить для этого числа поколений
my $generation = 0;                     # счетчик поколений
my $pop_ref = [];                       # ссылка на массив популяции
init_population($pop_ref, $popsize);

do
{
 evaluate_fitness($pop_ref, \&fitness);

 # вывод сводки по поколению
 my @sorted_population = sort { $a->{fitness} <=> $b->{fitness} } @$pop_ref;
 printf "generation %d: size %d, least fit DNA [%d], most fit DNA [%d]\n",
  $generation,
   scalar @sorted_population,
    $sorted_population[0]->{dna},
     $sorted_population[-1]->{dna};

 survive($pop_ref, $min_fitness);       # выбрать выживших из популяции
 select_parents($pop_ref);
 $pop_ref = recombine($pop_ref);        #  возвращает ссылку на новый массив популяции

 # начиная с этого места, мы работаем с новым поколением в  $pop_ref
 mutate($pop_ref, $mut_rate);           # применить мутацию к индивидуумам
} while ($generation++ < $generation_count); # выполнять, пока мы не исчерпаем поколения

sub init_population
{
 my $population = shift @_;
 my $pop_size = shift @_;

 # для каждого индивидуума
 foreach my $id (1 .. $pop_size)
 {
  # ввести анонимную  hash ссылку в массив популяции с данными индивидуума
  # ДНК равна номеру индивидуума минус 1 (0-255)
  push @$population, { dna => $id-1, survived => 1, parent => 0, fitness => 0 };
 }
}

sub evaluate_fitness
{
 my $population = shift @_;
 my $fitness_function = shift @_;

 foreach my $individual (@$population)
 {
  # присвоить приспособленности значение результата вызова функции приспособленности
  # для ДНК индивидуума
  $individual->{fitness} = $fitness_function->($individual->{dna});
 }
}

sub survive
{
```

```
 my $population = shift @_;
 my $min_fitness = shift @_;

 foreach my $individual (@$population)
 {
  # присвоить приспособленности значение результата вызова функции приспособленности
  # для ДНК индивидуума
  $individual->{survived} = $individual->{fitness} >= $min_fitness;

  # присвоить значение приспособленности 0 для негодных
  # индивидуумов (так что они не будут производить потомство)
  $individual->{fitness} = 0 if $individual->{fitness} < $min_fitness;
 }
}

sub select_parents
{
 my $population = shift @_;
 my $pop_size = scalar @$population;    # размер популяции

 # создать весовой массив весов: выбирать для популяции только выживших
 # после этого использовать map, чтобы получить только приспособленных
 my @weights = map { $_->{fitness} } grep { $_->{survived} } @$population;

 # если у нас только 2 выживших у нас неприятности
 die "Population size $pop_size is too small" if $pop_size < 2;

 # нам нужно заполнить  $pop_size слотов для воспитания
 # детей, чтобы сохранить размер популяции
 foreach my $slot (1..$pop_size)
 {
  my $index = sample(\@weights); # здесь мы передаем ссылку на весовой массив

  # делаем проверку на готовность $index
  die "Undefined index returned by sample()"
   unless defined $index;
  die "Invalid index $index returned by sample()"
   unless $index >= 0 && $index < $pop_size;

  # увеличиваем слоты воспитания детей для этих членов популяции
  $population->[$index]->{parent}++;

 }

}

sub recombine
{
 my $population = shift @_;
 my $pop_size = scalar @$population;    # размер популяции
 my @parent_population;
 my @new_population;

 my $total_parent_slots = 1;

 while ($total_parent_slots)
 {
  # выяснить, сколько осталось родительских слотов
  $total_parent_slots = 0;
  $total_parent_slots += $_->{parent} foreach @$population;

  last unless $total_parent_slots;

  # Если мы здесь, то мы уверены, что есть по крайней мере один индивидуум с parent>0
  my $individual = undef;               # начать с неопределенного индивидуума
```

```
  do
  {
   # выбрать случайного индивидуума
   $individual = $population->[int(rand($pop_size))];
   # индивидуум будет приемлемым, только если он может быть родителем
   undef($individual) unless $individual->{parent};
  } while (not defined $individual);

  push @parent_population, $individual; # включить индивидуума в популяцию родителей
  $individual->{parent}--;              # уменьшить родительский слот на 1

 }

 foreach my $parent (@parent_population)
 {
  # выбрать случайного индивидуума из популяции родителей (родитель #2)
  my $parent2 = @parent_population[int(rand($pop_size))];

  my $child = { survived => 1, parent => 0, fitness => 0 };

  # это размножение!
  my $bitmask = int(rand(256));          # случайный байт между 0 и 255
  # обменяться случайными битами между родителями в соответствии с bitmask
  $child->{dna} = ($parent2->{dna} & $bitmask) | ($parent->{dna} & ~$bitmask);

  push @new_population, $child;         # ребенок является частью нового поколения
 }

 return \@new_population;
}

sub mutate
{
 my $population = shift @_;
 my $mut_rate   = shift @_;

 foreach my $individual (@$population)
 {
  #  мутировать индивидуумов если  rand() больше  mut_rate
  next if rand > $mut_rate;
  # промутировать ДНК  выполнением and и затем or с двумя случайными
  # целыми числами от 0 до 255
  my $old_dna = $individual->{dna};
  $individual->{dna} &= int(rand(256));
  $individual->{dna} |= int(rand(256));
#  print "Mutated $old_dna to ", $individual->{dna}, "\n";
 }
}

sub fitness
{
 my $dna = shift @_;
 return $dna/256;
}

# Функция для выбора из массива взвешенных элементов=weighted elements
# исходно написана Абигайлом (Abigail) <abigail@foad.org>
sub sample
{
 # дать ссылку на массив весов
 my $weights = shift @_ or return undef;
 #  внутренние счетчики
 my ($count, $sample);

 for (my $i  = 0; $i < scalar @$weights; $i ++)
```

```
 {
  $count += $weights->[$i];
  $sample = $i if rand $count < $weights->[$i];
 }

 # возвратить индекс в массив весов
 return $sample;
}
```

В Листинге 1 есть некоторые интересные моменты. Основной цикл находится в начале, и вы можете понять все его части и как они вместе работают над популяцией (и все же они отдельные, так что мы можем снова использовать их в следующем примере).

Мы строим весовой массив в функции select_parents() с помощью map() поверх grep(). Это могло бы быть написано как цикл, однострочное решение будет здесь более прозрачным и не будет существенно замедлять программу.

Листинг 2. Однострочное решение

```
my @weights = map { $_->{fitness} } grep { $_->{survived} } @$population;
```

Ссылка на массив $population разыменована. Тогда только элементы массива с «полем выживания» (установленного раньше с помощью функции survive()) пройдут сквозь grep. После этого выжившие *отбираются* по их номерам приспособленности, которые помещаются на свои места в весовом массиве.

Размер популяции был выбран равным 256, так как в этом случае легко инициализировать каждого индивидуума с номером, равным его индексу. Не стесняйтесь начинать с другим размером популяции.

Скорости мутации более 1% заставляют максимум и минимум выживаемости значительно флуктуировать. Популяция никогда не придет к высокой приспособленности. Малые скорости мутации приводят популяцию к достижению высокой приспособленности, как правило, много медленнее. Итого, 1% вполне подходит для нашего размера популяции.

Алгоритм отбора размножений выбирает одного из родителей, просматривая веса – в действительности каждый индивидуум может иметь шанс быть родителем, но число родительских слотов конечно. Второй родитель выбирается *случайно* из родительской популяции. Почему? Мы могли бы использовать веса и для выбора второго родителя, но таким способом мы даем возможность каждому индивидууму участвовать в процессе размножения.

Фактическое размножение выполняется случайной 8-битовой bitmask. Мы делаем AND bitmask к ДНК первого родителя (вспомните, что это просто байт) и AND инвертированную bitmask к ДНК второго родителя. Эффект состоит в том, что мы выбираем случайные биты у одного из родителей, а остальные биты приходят от второго родителя.

Мутация выполняется с помощью AND и OR для индивидуальной ДНК с помощью случайной 8-битовой bitmask.

Кстати, наиболее подходящая ДНК, конечно, 255. Вам не нужно ждать 100 000 поколений. Просто нажмите Ctrl-C, когда вы любуетесь строкой состояния.

Размножающиеся слова

В этом примере мы сделаем ДНК 32 бита (5 байт). Каждый байт будет представлять собой букву или пробел. Мы могли бы запаковать больше информации в каждый байт, но это сделало бы пример менее понятным. Каждое значение байта (численно 0-255) будет от A до Z (если величина находится между 65 и 90, условно выбранных, чтобы соответствовать набору ASCII), или пробелу (если величина находится от 0 до 64, или от 91 до 255).

Обратите внимание, насколько этот пример похож на пример в Листинге 1. Словарь сопровождает программу.

Листинг 3. words.pl

```
#!/usr/bin/perl -w
```

```
#!/usr/bin/perl -w

# Демонстрация ГА со словарной ДНК (512 бит)

use strict;
use Data::Dumper;

# индивидуумы в популяции
my $popsize = 1024;                     # хорошее начало
my $dna_length = 512;                   # 4 "буквы" в ДНК
my
 $dna_byte_length = $dna_length / 8;    # длина ДНК в байтах
my $mut_rate = 0.01;                    # скорость мутации
my $min_fitness = 0.1;                  # минимальная приспособленность для выживания

my $generation_count = 100000;          # выполнить для этого числа поколений
my $generation = 0;                     # счетчик поколений

my $pop_ref = [];                       # ссылка на массив популяции

init_population($pop_ref, $popsize);

do
{
 evaluate_fitness($pop_ref, \&fitness);

 # вывод сводки по поколению
 my @sorted_population = sort { $a->{fitness} <=> $b->{fitness} } @$pop_ref;
 printf "generation %d: size %d\nleast fit DNA [%s]/%d\nmost fit DNA  [%s]/%d\n",
  $generation,
   scalar @sorted_population,
    dna_to_words($sorted_population[0]->{dna}),
     $sorted_population[0]->{fitness},
      dna_to_words($sorted_population[-1]->{dna}),
       $sorted_population[-1]->{fitness};

 survive($pop_ref, $min_fitness);       # выбрать выживших из популяции
 select_parents($pop_ref);
 $pop_ref = recombine($pop_ref);        # возвращает ссылку на новый массив популяции

 # начиная с этого места мы работаем с новым поколением в $pop_ref
 mutate($pop_ref, $mut_rate);           # применить мутацию к индивидуумам
} while ($generation++ < $generation_count); # выполнять, пока мы не исчерпаем поколения

sub init_population
{
 my $population = shift @_;
 my $pop_size = shift @_;

 # для каждого индивидуума
 foreach my $id (1 .. $pop_size)
 {
  # ввести анонимную hash ссылку в массив популяции с данными индивидуума
  # ДНК представляет собой случайное число
  my $random_dna = 0;
  foreach my $byte (1 .. $dna_byte_length)
  {
   vec($random_dna, $byte-1, 8) = int(rand(256));
   #   printf "Byte $byte; Random DNA is now [%64s]\n", dna_to_words($random_dna);
  }
  push @$population, { dna => $random_dna, survived => 1, parent => 0, fitness => 0 };
 }
}

sub evaluate_fitness
{
```

```
 my $population = shift @_;
 my $fitness_function = shift @_;

 foreach my $individual (@$population)
 {
  # установить в качестве приспособленности результат вызова функции приспособленности
  # на ДНК индивидуума
  $individual->{fitness} = $fitness_function->($individual->{dna});
 }
}

sub survive
{
 my $population = shift @_;
 my $min_fitness = shift @_;
 my $survived = 0;

 foreach my $individual (@$population)
 {
  # установить функцию приспособленности на 0 для неподходящих
  # индивидуумов (так что они не будут размножаться)
  $individual->{survived} = $individual->{fitness} >= $min_fitness;
  if ($individual->{survived})
  {
   $survived++;
  }
  else
  {
   $individual->{fitness} = 0
  }
 }
 if (0 == $survived)
 {
  die "No individuals survived, dying peacefully";
 }
}

sub
 select_parents
{
 my $population = shift @_;
 my $pop_size = scalar @$population;    # размер популяции

 # создать весовой массив: выбрать только выживших из популяции
 # затем использовать map, чтобы попали только приспособленные
 my @weights = map { $_->{fitness} } grep { $_->{survived} } @$population;

 # если у нас менее 2 выживших у нас неприятности
 die "Population size $pop_size is too small" if $pop_size < 2;

 # нам нужно пополнить $pop_size слотов для воспитания детей, чтобы сохранить
 # размер популяции
 foreach my $slot (1..$pop_size)
 {
  my $index = sample(\@weights); # здесь мы передаем ссылку на весовой массив

  # сделать проверку готовности к работе по  $index
  die "Undefined index returned by sample(), probably all individuals have died"
   unless defined $index;
  die "Invalid index $index returned by sample()"
   unless $index >= 0 && $index < $pop_size;

  # увеличить воспитательные слоты для этих членов популяции
  $population->[$index]->{parent}++;
```

```
 }
}

sub recombine
{

 my $population = shift @_;
 my $pop_size = scalar @$population;    # размер популяции
 my @parent_population;
 my @new_population;

 my $total_parent_slots = 1;

 while ($total_parent_slots)
 {
  #  выяснить, сколько осталось родительских слотов
  $total_parent_slots = 0;
  $total_parent_slots += $_->{parent} foreach @$population;

  last unless $total_parent_slots;

  # если мы здесь, то мы уверены, что есть, по крайней мере, один индивидуум с parent > 0
  my $individual = undef;               # начать с неопределенного индивидуума
  do
  {
   # выбрать случайного индивидуума
   $individual = $population->[int(rand($pop_size))];
   # индивидуум приемлем только, если он может быть родителем
   undef($individual) unless $individual->{parent};
  } while (not defined $individual);

  push @parent_population, $individual; # внести индивидуума в популяцию родителей
  $individual->{parent}--;              # уменьшить родительский слот на 1

 }

 foreach my $parent (@parent_population)
 {

  # выбрать случайного индивидуума из популяции родителей (родитель #2)
  my $parent2 = @parent_population[int(rand($pop_size))];

  my $child = { survived => 1, parent => 0, fitness => 0, dna => 0 };

  # это размножение!
  my $dna1 = $parent->{dna};
  my $dna2 = $parent2->{dna};

  # заметьте, что мы делаем операции на БАЙТАХ а не на БИТАХ.
  # единицей информации является байт (и сохранение их является быстрым
  # методом размножения)
  foreach my $byte (1 .. $dna_byte_length)
  {
   # взять один случайный байт от родителей и добавить его ребенку
   vec($child->{dna}, $byte-1, 8) = vec(((rand() < 0.5) ? $dna1 : $dna2), $byte-1, 8);
  }

  push @new_population, $child;         # ребенок теперь является частью нового поколения
 }

 return \@new_population;
}

sub mutate
{
```

```
 my $population = shift @_;
 my $mut_rate   = shift @_;

 foreach my $individual (@$population)
 {

  # мутировать индивидуумов если  rand() больше  mut_rate
  next if rand > $mut_rate;
  # промутировать ДНК  выполнением and и затем or с двумя случайными
  # целыми числами между 0 и 2^$dna_length
  my $old_dna = $individual->{dna};
  my $new_dna = 0;

  foreach my $byte (1 .. $dna_byte_length)
  {
   vec($new_dna, $byte-1, 8) &= int(rand(256));
   vec($new_dna, $byte-1, 8) |= int(rand(256));
  }

  $individual->{dna} = $new_dna;
#  print "Mutated $old_dna to ", $individual->{dna}, "\n";
 }
}

# это замыкающий блок!
{
 # статическая переменная @dictionary только для fitness()
 my @dictionary;
 my %freqs;
 # вычислить приспособленность ДНК
 sub fitness
 {
  my $dna = shift @_;
  my $words = dna_to_words($dna);
  my $fitness = 0;                      # начать с приспособленности равной 0
  my $max_entry_length = 20;            # наиболее длинное слова которое мы допускаем

  # вы можете использовать любой список слов в конце программы
  # выполнить инициализацию @dictionary только один раз
  unless (@dictionary)
  {
   @dictionary = <DATA>;
   foreach (@dictionary)
   {
    chomp;
   }

   # выделить слова большие $max_entry_length букв, и перевести их в верхний регистр
   @dictionary = grep { length($_) > 1  && length($_) < $max_entry_length }
    map { uc } @dictionary;
   # составить hash частотности букв (помните, что все буквы в верхнем регистре)
   $freqs{$_}++ foreach split '', join '', @dictionary;
  }

  # нет простого способа избежать этой полной проверки словаря
  # без очень существенного усложнения
  foreach my $entry (@dictionary, 'A'..'Z')
  {
   # нечего не делать, если элемент не совпадает с ДНК, или наоборот
   next unless $words =~ /$entry/;

   # есть совпадение!  (это может быть часть строки, что нормально )
   # увеличьте приспособленность в зависимости от того, насколько длинным было совпадение;
   $fitness += 2**length($entry);
   $fitness+= $freqs{$entry} if exists $freqs{$entry};
```

```
  }
  return $fitness;
 } # конец fitness()

}

# Функция для выборки из массива взвешенных элементов
# исходно написана Абигайлом (Abigail) <abigail@foad.org>
# Документация по алгоритму находится на
# http://theoryx5.uwinnipeg.ca/CPAN/data/Sample/Sample.html
# (the CPAN Sample module)
sub sample
{
 # найти ссылку на массив весов
 my $weights = shift @_ or return undef;
 # внутренние счетчики
 my ($count, $sample);

 for (my $i  = 0; $i < scalar @$weights; $i ++)
 {
  $count += $weights->[$i];
  $sample = $i if rand $count < $weights->[$i];

 }

 # возвратить индекс в массив весов
 return $sample;
}

# перевод центрованных ASCII байтов в буквы
sub byte_to_letter
{
 my $dna = shift @_;
 my $byte = shift @_;
# print "Got byte $byte\n";
 my $letter = vec $dna, $byte, 8;
 # находится байт в диапазонах букв? если да, вернуть букву
 return chr($letter) if ($letter >= 65 && $letter <= 90);
 # если нет, вернуть пробел. Использование ord() может быть каждый раз кэшировано.
 return ' ';
}

# распечатать ДНК как скаляр
sub dna_to_words
{
 my $dna = shift @_;
 my @words;

 foreach my $byte (1.. $dna_byte_length)
 {
  # напечатать буквенные эквиваленты текущего байта
  push @words, byte_to_letter($dna, $byte-1);
 }

 # возврат пригодных для напечатания слов
 return join '', @words;
}

__DATA__
about
algorithm
and
biology
by
century
```

```
come
computer
electronics
evolution
field
fitting
genetic
in
intriguing
is
it
most
of
one
only
progress
reach
rivaled
sciences
speed
that
the
to
was
```

Основная проблема в этом примере состоит в том, что ДНК длиной больше 32 бит трудно управлять. Я начинал делать свои собственные битовые операции, которые получились не только громоздкими, но и чрезвычайно медленными. Затем я попробовал пакет Math::BigInt, который оказался очень медленным для этой цели.

В конце концов я остановился на функции vec() -- она достаточно быстрая, и была правильным выбором для управления ДНК (по существу ДНК -- это *битовый вектор*, встроенная структура данных Perl). Используйте "perldoc -f vec" для того, чтобы больше узнать о функции vec().

Можно закончить с 1024 индивидуумами с нулевой приспособленностью. Поэтому этот пример лучше защищен против такой "ситуации", чем первый пример.

Функции init_population(), recombine() и mutate() были изменены для того, чтобы обрабатывать битовые вектора вместо байтов.

Функция dna_to_words() несколько неэффективна, но она вызывается не слишком часто, чтобы это имело значение. Наибольшее замедление происходит от функции fitness(), которая пытается найти совпадения со всеми словами в словаре, плюс со всеми буквами алфавита.

Приспособленность вычислялась следующим образом: 2 за каждую букву в ДНК, плюс частотность буквы в словаре, плюс 2^N за каждое словарное слово длиной N в ДНК. Массив словаря и hash частотности букв получаются только однажды (используя *closure*). Пожалуйста, модифицируйте функцию приспособленности и словарь, для того чтобы создавать свои английские слова. Показанная Формула приспособленности сильно смещена в сторону букв и ленится объединяться в английские слова (хотя "on" и "in" появляются достаточно часто).

Выводы

Эволюционный генетический алгоритм -- очаровательная тема, которая вряд ли может быть исчерпана одной статьей. Я надеюсь, что вы поэкспериментируете с примерами и создадите собственные основы дарвинского размножения. Особенно увлекательно поиграть с функцией приспособленности во втором примере и наблюдать, как английские слова появляются из шума.

Технологии, показанные в примерах, находятся в диапазоне от начального до самого передового уровня, так что постарайтесь как следует понять их. Часто можно сделать улучшения. Функция vec() особенно интересна. Она прекрасно подходит для длиннобитовых векторов, таких как ДНК и других численных данных.

Напишите собственную реализацию генетического алгоритма. Сравните ее с моей и выясните недостатки (не обязательно ваши, любые). Реализация алгоритма -- это хитрая задача. Есть многое, что вы можете сделать неправильно, и существует всего несколько путей правильной реализации.

Ресурсы

- Примите участие в обсуждении материала на форуме.
- Оригинал этой статьи на developerWorks.
- Прочитайте другие статьи по Perl, написанные Тедом в серии "Cultured Perl" на *developerWorks*.
- Посетите CPAN, там есть все модули Perl, какие можно пожелать.
- Проверьте Perl.com для получения информации о Perl и связанных с ним ресурсах.
- "Программирование на Perl, третье издание (Programming Perl Third Edition)", написанная Ларри Волом (Larry Wall), Томом Кристиансеном (Tom Christiansen) и Джоном Орвантом (Jon Orwant) (O'Reilly & Associates 2000) является сегодня лучшим справочником по Perl, от 5.005 до 5.6.0.
- "Создание алгоритмов на Perl (Mastering Algorithms with Perl)", написанная Джоном Орвантом (Jon Orwant), Яаркко Хитаниеми (Jarkko Hietaniemi) и Джоном Макдональдом (John Macdonald) (O'Reilly & Associates 1999) является большим сборником алгоритмов на Perl. Часть 14, "Probability," показывает, как делать взвешенные и невзвешенные распределения с помощью Perl.
- Genetic Algorithms FAQ достаточно устарела, но она указывает на полезные наборы программного обеспечения по генетическим алгоритмам, как бесплатные, так и платные.
- Статьи Теодора Златанова на *developerWorks* :
  - One-liners, 101
  - Маленькие наблюдения большой картины (Perl; Small observations about the big picture)
- Пролистайте дополнительные ресурсы по Linux на *developerWorks*.
- Пролистайте дополнительные ресурсы по Open source на *developerWorks*.
- Участвуйте в русскоязычных форумах ОС Linux.

Об авторе

Теодор Златанов (Teodor Zlatanov) получил диплом магистра по вычислительной технике в Boston University в 1999. Он работает программистом с 1992, используя Perl, Java, C, и C++. Он интересуется работами с открытым исходным кодом по синтаксическому анализу текста, трехуровневыми архитектурами клиент-серверных баз данных, системным администрированием UNIX, CORBA и управлением проектами.

IBM обладает всеми авторскими правами касательно информации, расположенной на developerWorks. Использование информации приведенной на этом ресурсе без явного письменного разрешения от IBM или первоначального автора запрещены. Если Вы желаете использовать информацию с developerWorks, пожалуйста воспользуйтесь регистрационной формой для того, чтобы связаться с нами запрос на использование материалов developerWorks Россия.