IBM

Cultured Perl: Генетические алгоритмы, следующее поколение

Более сложные примеры генетических алгоритмов на Perl

Уровень сложности: простой

Теодор Златанов, программист, Gold Software Systems

12.07.2007

> Тед делает следующий шаг в работе с генетическими алгоритмами на Perl. Он начинает там, где он остановился в первой статье по генетическим алгоритмам, работая с программой, которая ищет наборы словарных слов в ДНК индивидуума.

Одним из наиболее увлекательных алгоритмов является генетический алгоритм. Генетический алгоритм моделирует дарвиновский естественный отбор, где "приспособленность" отбирает индивидуумов для выживания. Основы этого я изложил в предыдущей статье, а также я показал там две реализации на Perl, одна из которых размножает байты, а другая слова.

В этой статье я изложу более сложный материал по генетическим алгоритмам на Perl. Вам может захотеться вернуться к первой статье и прочитать ее, прежде чем погрузиться в эту статью; генетические алгоритмы имеют точно определенные шаги, и некоторые коды в этой статье заимствованы из предыдущей статьи без объяснения деталей.

Прежде чем вы начнете, вы должны иметь в вашей системе инсталлированный Perl 5.6.0 или более позднюю версию. Примеры могут работать и на более ранних версиях Perl и на платформах, отличающихся от обычных UNIX-платформ (таких как Windows), но они не были протестированы на таких окружениях и могут потребовать дополнительной работы, чтобы заставить их запускаться.

Опять слова

В предыдущей статье был показан пример, в котором ищут словарные слова в ДНК индивидуума и располагают индивидуумов в зависимости от того, сколько словарных слов (чем длиннее, тем лучше) имеет ДНК индивидуума.

Здесь мы начнем с этого словарного примера и модифицируем его, как показано в Листинге 1 (загрузите полный исходник в commands.pl). Теперь каждое слово (слова в ДНК разделены пробелами) представляет собой команду для увеличения или уменьшения приспособленности индивидуума. Если вы сделаете правила такими, что будет слишком трудно улучшать приспособленность, то ваша исходная популяция не имеет шансов. Если вы слишком ослабите правила, индивидуальная приспособленность не будет неуклонно повышаться со временем, что сделает генетический алгоритм бесполезным.

Неудивительно, что индивидуумы так быстро приспосабливаются к новым правилам и используют их для достижения больших успехов, начиная с чисто случайного шума. Что удивительно, так это то, что правила используются неожиданным образом. Например, когда я сделал такое правило, что численная команда будет устанавливать приспособленность равной этому числу, ДНК отбросила все другие команды в пользу численной команды. Когда я уменьшал приспособленность для неправильных команд, чтобы не дать ДНК превратиться в полностью численную, индивидуумы просто сдвинули численную команду в конец ДНК, где она была защищена от неправильных команд.

Листинг 1. Функция commands.pl fitness()

```
# вычислить приспособленность ДНК
sub fitness
{
 my $dna = shift @_;
 my @words = split ' ', dna_to_words($dna);
```

```
 my $fitness = 0;                        # начать с нулевой приспособленности
 my $max_entry_length = 20;              # самое длинное слово, которое допустимо

 # заметьте что 'слова' здесь означают командные слова или числа

 foreach my $word (@words)               # выполнять все слова как  "команды"
 {
 if ($word eq 'M')      #  команда 'умножить (multiply)'
  {
   $fitness *= 2;
  }
  elsif ($word =~ /^A\D*(\d+)/) # команда 'усилить (amplify)'
  {
   $fitness *= $1
  }
  elsif ($word =~ /(\d+)/)            # если команда число
  {
   $fitness += length($1); # увеличить приспособленность  в зависимости количества цифр
  }
  else
  {
   $fitness *= 0.80;          # наказание за 'плохую' команду
  }
 }

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

Я говорю о ДНК и индивидуумах так, как будто бы они живые. До известной степени это так. Я никогда не забуду их апатию, когда я задал им жесткие правила, и как они размножались, когда правила позволили им сильно поднять приспособленность путем, которого я не предвидел. Попробуйте ввести новые правила в функцию `fitness()` и посмотрите сами, как индивидуумы эволюционируют, чтобы выжить.

Более сложные головоломки

После завершения программы в предыдущем разделе я подумал, что делать дальше. Я мог бы улучшить алгоритм, добавив дополнительных причиндалов, таких как фенотипы или более гибкие правила, или я мог бы еще позабавляться с функцией `fitness()`.

В разделе Ресурсы есть ссылка на MyBeasties, улучшенную версию модуля Perl для применения в генетических алгоритмах. Я вряд ли бы смог улучшить в этом модуле набор техник для реализации генетического алгоритма, но, в рамках того, что мы уже сделали, я мог сделать несколько забавных примеров, которые не требуют более сложных генетических алгоритмов.

В качестве следующего теста на приспособленность я реализовал последовательность команд для движения от точки A к точке B. Каждый индивидуум стартует с приспособленностью, равной 1, из точки A и может увеличить свою приспособленность в зависимости от того, как близко он подойдет к точке B. Каждая из команд "U", "D", "L" или "R" перемещает вверх, вниз, влево или вправо, соответственно. Команда "B" перемещает назад. Число после команды указывает, сколько раз ее нужно выполнить.

Двигаясь, индивидуумы должны следовать по маршруту. Движение вдоль траектории дает индивидууму приспособленность, пропорциональную тому, как далеко он ушел вдоль траектории. Смотрите Листинг 2; полный исходный текст находится в motion.pl. Индивидуумы обычно делают девять шагов вдоль траектории с 110-байтной ДНК; по-видимому, более длинная ДНК позволяет им проходить дальше по траектории. Без числовых параметров индивидуумы действительно чувствуют себя лучшее, возможно потому, что мутации с меньшей вероятностью портят ДНК при более простых командах. Человеческая ДНК имеет только 4 основных строительных блока (которые обычно записывают как G, A, T, и C), так что это не через чур надуманная теория.

Обратите внимание на то, как реализованы команды: с помощью гибкого стека движения и команды "back (назад)", которая вызывается способом, совместимым с остальными командами (как член хэша %instructions). К

тому же использование модуля `Math::Complex` делает моделирование двумерного движения тривиальным. (Можете ли вы реализовать функцию "повторить последнюю команду"?) Обратите также внимание на то, что приспособленность сильно возрастает для команды, которая продолжает движение по маршруту.

Листинг 2. Функция fitness() программы motion.pl

```
# вычислить приспособленность ДНК
sub fitness
{
 my $dna = shift @_;
 my @words = split ' ', dna_to_words($dna);

 my $fitness  = 2;                      # начать с малой приспособленности
 my $location = Math::Complex->make(0,0); # начальная координата
 # "идеальные" координаты
 my $goal     = Math::Complex->make(10,10);

 # маршрут к цели, которому должны следовать команды ДНК
 my @path = split ' ', 'U U U R R R U U U R R R U U U R R R U R';
 my $path_followed = 1;

 # массив для сохранения стека движения.
 my @motion_stack;

 # команды движения
 my $instructions = {
                     U  => Math::Complex->make( 0, 1),
                     D  => Math::Complex->make( 0,-1),
                     L  => Math::Complex->make(-1, 0),
                     R  => Math::Complex->make( 1, 0),
                     B =>               # move back
                     sub {
                      my $location = shift @_;
                      my $motion_stack = shift @_;
                      my $instructions = shift @_;
                      my $old_motion = pop @$motion_stack;
                      $location += $instructions->{$old_motion} if defined $old_motion;
                      return $location;
                     },
                    };

 # заметьте, что 'слова' здесь это команды слова или числа

 foreach my $word (@words)              # выполнить все слова как "команды"
 {
  # обрабатывать только легитимные команды (они могут быть вложены в более длинные слова)
  my ($motion, $mag) = $word =~ m/([A-Z])(\d*)/;
  if ($motion && exists $instructions->{$motion})
  {
   $mag = 1 unless $mag;                # выполнить по крайней мере один раз
   my $instruction = $instructions->{$motion};
   foreach (1..$mag)
   {
    if (ref $instruction eq 'Math::Complex')
    {
     $location += $instruction;         # использовать вектор движения
     push @motion_stack, $motion;
    }
    elsif (ref $instruction eq 'CODE')
    {
     $location = $instruction->($location, \@motion_stack, $instructions);
       # использовать подпрограмму
    }
```

```
   }
  }
 }

 # теперь проверить, следует ли индивидуум необходимой траектории
   # (если нет, то он  "упал с утеса")
 foreach my $path_instruction (@path)
 {
  my $instruction = shift @motion_stack;
    # извлечь фактически выполняемую команду
  if (defined $instruction && $instruction eq $path_instruction)
  {
   # увеличить приспособленность, этот индивидуум идет по правильному пути
   $fitness *= 2;
   $path_followed = 1;                  # индивидуум не заблудился
  }
  else
  {
   # чуть-чуть увеличить приспособленность, чтобы индивидуум получил вознаграждение за то
     # что он сделал шаг
   $fitness++;
   $path_followed = 0;  # индивидуум заблудился
   last;

  }
 }

 if ($location == $goal && $path_followed)
 {
  # индивидуум нашел цель, нет смысла продолжать.
  die "Individual [@words] reached the target!"; }

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

Назад к словам

В исходной статье мы использовали ДНК как источник слов и увеличивали приспособленность индивидуума за те слова, которые были в нашем словаре, учитывая длину слов. Листинг 3 (полный исходный текст находится в words2.pl) использует эту же идею, но применяя приблизительное сравнение с помощью модуля `String::Approx` (который вы можете установить). Идея состоит в том, чтобы награждать не только за точные совпадения, но и за приблизительные совпадения (в процентах), было бы лучше награждать по непрерывной шкале, но это существенно усложнит алгоритм.

Такая модификация оригинального алгоритма оказалась очень полезной, обычно дающей трехбуквенные слова на 40 поколениях. Приблизительные совпадения обеспечивают, что потенциально успешные образцы ДНК получают награду. То что награда основана не только на совпадении, но и на длине слова поощряет появление более длинных слов в ДНК.

Функция `fitness()` похожа на функцию из исходного примера, но структура наград отличается и цикл упрощен. Заметьте, что точные совпадения являются значительно более ценными, чем приблизительные совпадения.

Листинг 3. Функция words2.pl fitness()

```
# заключительный блок!
{
```

```
 # индивидуальная статическая переменная @dictionary только
 # в заключительной части fitness()
 my @dictionary;
 # вычислить приспособленность ДНК
 sub fitness
 {
  my $dna = shift @_;
  my @words = split ' ', dna_to_words($dna);
  my $fitness = 0;                      # начать с нулевой приспособленности
  my $max_entry_length = 20;            # наиболее длинное слово, которое мы допускаем
  my $precision = 90;                   # точность совпадения в процентах
  my $imprecision = 100 - 90;

  die "Can't use a negative imprecision (your precision must
                       be less than 100 but it's $precision)"
   if $imprecision < 0;

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

   # исключить слова длинной больше чем $max_entry_length букв и перевести их
   # в верхний регистр
   @dictionary = grep { length($_) > 1  && length($_) < $max_entry_length }
    map { uc } @dictionary;
  }

  # нет простого способа избежать этой полной проверки словаря
  # без слишком сильного усложнения примера
  foreach my $word (@words)
  {
   next unless length $ word > 1;       # не использовать отдельные буквы
   # заметьте, что мы используем "S#%", мы не хотим
   # вводить/присоединять сходные элементы
     # только замена сходных элементов
   my @matches = amatch($word, "$imprecision%", @dictionary);
   my @precise_matches = grep { $word eq $_ } @dictionary;
   $fitness += scalar @matches;
   $fitness += (10 ** length $_) foreach @precise_matches;
     # наградить более длинные слова существенно больше
  }
  return $fitness;
 } # конец fitness()
}
```

Выводы

Я надеюсь, что прогонка примеров, приведенных в этой статье, будет так же занимательна для вас, как и для меня. Не бойтесь поиграть с параметрами и с функцией `fitness()`. Обязательно изучите MyBeasties (смотрите раздел Ресурсы), если вы заинтересованы в использовании генетических алгоритмов в ваших приложениях. Так как скорость -- это основная проблема при реализации генетического алгоритма, и поэтому, в большинстве случаев, они должны быть тщательно оптимизированы, в любом языке существует мало жизнеспособных наборов инструментальных средств для генетического алгоритма общего назначения. MyBeasties является хорошим примером полезного и достаточно быстрого инструментального средства.

Область применений генетических алгоритмов безгранична. Если квантовые суперкомпьютеры будут когда-нибудь разработаны, генетические алгоритмы сразу станут не только осуществимыми, но и предпочтительными в качестве подхода к решению многих задач. Замысел одновременной оценки квантовыми компьютерами многих возможных решений задачи кажется предназначенным для удовлетворения замысла эволюционных вычислений с большими популяциями в генетических алгоритмах.

Ресурсы

- Примите участие в обсуждении материала на форуме.
- Оригинал этой статьи на developerWorks.
- Прочитайте другие статьи Теда о Perl в серии "Cultured Perl" на *developerWorks*.
- Загрузите листинги программ, упомянутых в этой статье:
  - commands.pl
  - motion.pl
  - words2.pl
- Эта статья построена на базе моей первой статьи по генетическим алгоритмам.
- Зайдите на CPAN за любыми модулями Perl.
- Зайдите на Perl.com для получения дополнительной информации о Perl и связанных с ним ресурсах.
- MyBeasties представляет собой полный набор модулей Perl для программирования генетических алгоритмов. Они значительно более усовершенствованные, чем те примеры, которые приведены в этой статье, в том, что касается построения структуры генетического алгоритма, но для начинающих программировать на Perl они будут, вероятно, слишком сложными.
- Детальный обзор некоторых интересных вычислительных трудностей, встречающихся при упорядочивании генома, читайте "Computational challenges in structural and functional genomics" (*IBM Systems Journal* 40:2, 2001)

Об авторе

sity в 1999. Он работает программистом с 1992, используя Perl, Java, C, и C++. Он интересуется работами с открытым