4.4 Списки, генетическое программирование на машинном языке и эволюция грамматик

Разреженные деревья являются не единственным способом для представления программ. Некоторые практики от ГП кодируют программы в виде **списков** (***lists***) произвольной длины (или **строк** (***strings***)), состоящих из инструкций на машинном языке. Особи оцениваются путем преобразования списков в функции с их последующим исполнением. Такой подход называют **Линейным генетическим программированием** (***Linear Genetic Programming***), а его наиболее известными приверженцами являются Вольфганг Банжаф (*Wolfgang Banzhaf*), Питер Нордин (*Peter Nordin*), Роберт Келлер (*Robert Keller*) и Франк Франконе (*Frank Francone*), которые продают супербыструю ГП систему, Discipulus, основанную на данном подходе, и которые написали известную книгу по генетическому программированию, наполовину посвященную ГП с деревьями, и наполовину — линейному ГП[1].

Выполнение произвольных строк машинного кода может быть опасным в том случае, если не поддерживается его прерывание. Но как его реализовать? Очевидно, что особь не может быть представлена битовой строкой, потому что в этом случае возможна запись любой машинной инструкции, включая нежелательные или не имеющие смысла[2]. И понятно, что набор инструкций должен составляться из некоторого тщательно определенного множества.

Если множество инструкций конечно, то можно присвоить каждой инструкции уникальный номер и представить генотип как список целых чисел. Обычно набор регистров также конечен, что позволяет исполнять списки машинного кода, представленные ориентированным ациклическим графом, когда более ранние инструкции оказывают влияние на более поздние в силу того, что используются одни и те же регистры. Также бывает желательно включить специальные инструкции для работы с константами (Прибавить 2 и т.д.).

Стековые языки тесно связаны с машинным кодом, поэтому не должно быть большим сюрпризом то, что, как упоминалось в Разделе 4.3.7, некоторые стековые языки непосредственно применяются к представлению программ с использованием списков, если, конечно, в языке нет препятствующих этому синтаксических ограничений.

Списки могут применяться и для генерации деревьев: рассмотрим финальную схему ГП, названную **Грамматической эволюцией** (ГЭ) (***Grammatical evolution, GE***), предложенную Конором Райаном (*Conor Ryan*), Дж. Дж. Коллинз (*J.J. Collins*) и Майклом О'Нилом (*MichaelO'Neill*)[3]. В грамматической эволюции применяется представление списком целых или булевских значений, и для построения ГП дерева используются заранее определенные решающие точки (*decision points*) в дереве грамматики. Полученное дереве затем оценивается в традиционной для ГП манере для вычисления приспособленности. Такой достаточно сложный подход является еще одним примером косвенного кодирования, и хотя здесь нет модульности, присущей многим косвенным кодировкам, он безумен по-своему: в нем можно задать *любое дерево* на *любом языке*.

В качестве примера рассмотрим грамматику, не имеющую особого смысла, и особь, представленную списком (рис. 28). Для интерпретации начинаем с узла *tree* и используем первый элемент из списка, чтобы определить как развернуть этот узел (предполагаем, что *false* используется для разворачивания с использованием первого элемента, а *true* — второго). После этого производится разворачивание в глубину оставшихся переменных. На рис. 29 показано разворачивание особи, изображеннной на рис. 28.

$tree$	$\rightarrow$	$n+n$	$\mid$	$n-n$	
n	$\rightarrow$	$n*m$	$\mid$	$\sin m$	false false true true false true false true true false false
m	$\rightarrow$	1	$\mid$	2	

Рис. 28. Грамматика для грамматической эволюции и особь, представленная списком

---

[0] Перевод раздела из книги Luke S. Essentials of Metaheuristics. A Set of Undergraduate Lecture Notes. Zeroth Edition. Online Version 0.8. March, 2010 (`http://cs.gmu.edu/~sean/book/metaheuristics/`). Перевел – Юрий Цой, 2010 г. Любые замечания, касающиеся перевода, просьба присылать по адресу yurytsoy@gmail.com
Данный текст доступен по адресу: `http://qai.narod.ru/GA/meta-heuristics_4_4.pdf`

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone, 1998, Genetic Programming: An Introduction, Morgan Kaufmann.

[2] Вряд ли вам захотелось бы выполнить знаменитую инструкцию HCF («Halt and Catch Fire»). Поищите ее в Википедии.

[3] Conor Ryan, J.J. Collins, and Michael O'Neill, 1998, Grammatical evolution: Evolving programs for an arbitrary language, in EuroGP 1998, pages 83-96.

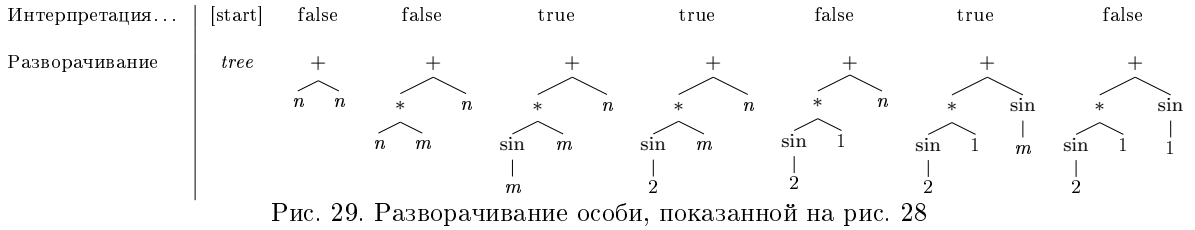

Рис. 29. Разворачивание особи, показанной на рис. 28

Итак, у нас есть дерево, которое можно оценивать! Отметим, что последние 4 бита (*true true false false*) особи не были использованы. А как быть если список слишком короткий и имеющихся решающих точек недостаточно? В этом случае обычно список снова читается с начала. Вряд ли это отличное решение, но оно работает. ГЭ хороша тем, что позволяет построить любое правильное дерево для данной грамматики, и поэтому является гораздо более гибкой по сравнению со стандартным ГП с кодированием деревьями: действительно, в ней не нужно даже волноваться о сильной типизации. Недостатком является то, что такое представление в определенных случаях лишено гладкости: малейшие изменения в начале списка могут привести к огромным изменениям дерева. От этого могут быть проблемы.

4.4.1 Инициализация

Способ создания новых списков зависит в основном от тех потребностей используемого метода, которые обусловлены самой проблемной областью. В общем случае имеется две сложности: определение длины списка и его заполнение. Одним из простых решений для первой является определение длины с помощью геометрического распределения (алгоритм 46, возможно нужно указать в качестве минимально возможной длины списка 1). Нужно учитывать, что при данном распределении будет большое количество коротких списков, поэтому более плоское распределение может оказать полезнее.

Для заполнения списка нужно просто последовательно пройтись по всем его элементам и для каджого назначить случайное, но корректное значение. Помните, что для некоторых задач этого недостаточно, т.к. существуют ограничения, при которых одни элементы в списке могут появиться только после других, и в этом случае нужно действовать разумнее.

4.4.2 Мутация

Как и инициализация, мутация списков состоит из двух этапов: изменение размера списка и изменение содержимого.Последнее может быть изменено точно также, как и при использовании векторов фиксированной длины: битовая мутация, случайные целые числа и т.д. Помните, что иногда нельзя произвести изменения одних элементов без предварительного изменения других из-за наличия соответствующих ограничений.

Изменение длины списка также зависит от решаемой задачи: к примеру в некоторых лучше добавлять новые элементы в конец списка. В простейшем случае по некоторому распределению выбирается число, которое прибавляется (или отнимается, если необходимо) к длине списка. Например, можно начать случайные блуждания от 0 и подбрасывать на каждом шаге монетку, до тех пор пока не выпадет решка. Номер шага будет определять число элементов добавляемых (или удаляемых, если число отрицательное) к концу списка. Следующий алгоритм должен выглядеть знакомым:

Не спутайте этот алгоритм с алгоритмом 42 (мутация случайным блужданием), в котором случайное блуждание используется для определения шума для мутации. Его следует применять с осторожностью, т.к. списки не могут быть меньше 1, зато они могут быть сколь угодно большими, а случайное блуждание, подобное описанному, может привести к существенному росту длины списков, против которого потребуется введение правил предотвращающих большой рост из-за оператора мутации (см. также обсуждение **разрастания кода** (***bloat***) для описания других причин роста).

Предупреждение В некоторых подходах с использованием представления списками, таких как грамматическая эволюция, начальные элементы списка имеют *гораздо* большее значение, чем последние. В ГЭ это объясняется тем, что начальные элементы определяют ранние этапы построения дерева и их изменение существенно изменит само дерево, в то время как последние элементы относятся к малым поддеревьям или отдельным элементам (либо если список очень длинный, они

Алгоритм 58 Случайное блуждание

```
1: b ←вероятность переворачивания монетки {При большем b случайное блуждание будет дольше
   и результат будет более неопределенным}
2: m ←0
3: if p ≥ случайное число равномерно распределенное от 0.0 до 1.0, включая концы then
4:   repeat
5:     n ← либо +1 либо -1, случайно
6:     if m + n в допустимых границах then
7:       m ← m + n
8:     else
9:       m ← m − n
10:    end if
11:  until b ≥ случайное число равномерно распределенное от 0.0 до 1.0, включая концы
12: end if
13: return m
```

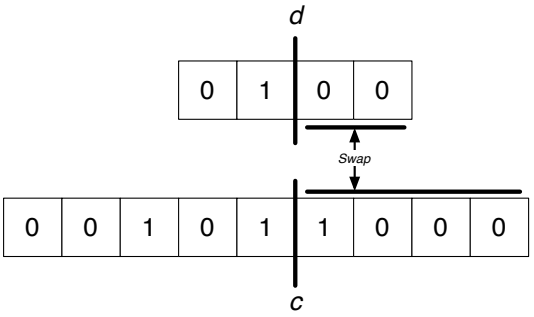

Рис. .28: Одноточечный списочный кроссинговер

вообще ни на что не влияют!). Это очень сильно сказывается на гладкости рельефа функции приспособленности, и будет лучше, если процедура мутации будет учитывать данную особенность. Например, можно только *изредка* изменять элементы в начале списка и гораздо чаще изменять элементы ближе к концу. Линейное ГП может обладать этим свойством в зависимости от рассматриваемой задачи и в действительности может возникнуть противоположная ситуация, если участки машинного кода в конце списка имеют наибольшую значимость.

4.4.3 Рекомбинация

Как и мутация, кроссинговер также может зависеть от различных ограничений, но если не обращать на это внимания, то существуют разнообразные способы выполнения кроссинговера между списками переменной длины. Простейшими являются **одноточечный** (***one-point***) и **двухточечный списочный кроссинговер** (***two-point list crossover***), представляющие модификации стандартного одно- и двухточечного кроссинговера. В одноточечном списочном кроссинговере, рис. 30, выбираются (возможно различные) точки в каждом списке, а затем производится обмен сегментами справа от этих точек. Сегменты должны быть ненулевой длины. Вот до жути знакомый алгоритм:

Двухточечный списочный кроссинговер, показанный на рис. 31, работает аналогично: выбираем *две* точки в каждом списке и обмениваем местами средние участки. Снова отметим, что точки могут быть в разных местах для каждого списка. Прежде чем применять одно или двухточечный списочный кроссинговер, тщательно проанализируйте используемый способ кодирования, чтобы определить, какой оператор лучши использовать. У них очень разные свойства. К примеру, зависит ли ваше кодирование от того, что записано в середине списка, и насколько оно чувствительно к разрушениям в этой части?

Еще одно предупреждение Точно также, как уже упоминалось для мутации, необходимо учитывать, что одни элементы списка могут иметь большую значимость, чем другие, и быть более чувствительными к перемешиванию посредством кроссинговера. Так, например, в грамматической

Алгоритм 59 Одноточечный списочный кроссинговер

1: $\overrightarrow{v} \leftarrow$ первый список $\langle v_1, v_2, \ldots, v_l \rangle$ для скрещивания
2: $\overrightarrow{w} \leftarrow$ второй список $\langle w_1, w_2, \ldots, w_l \rangle$ для скрещивания

3: $c \leftarrow$случайное целое, равномерно распределенное в диапазоне от 1 до l
4: $d \leftarrow$случайное целое, равномерно распределенное в диапазоне от 1 до k
5: $\overrightarrow{x} \leftarrow$вырезанный участок из $\overrightarrow{v}$ от v_c до v_l
6: $\overrightarrow{y} \leftarrow$вырезанный участок из $\overrightarrow{w}$ от w_d до w_k
7: Вставить $\overrightarrow{y}$ в $\overrightarrow{v}$ вместо вырезанного участка
8: Вставить $\overrightarrow{x}$ в $\overrightarrow{w}$ вместо вырезанного участка
9: **return** $\overrightarrow{v}$ и $\overrightarrow{w}$

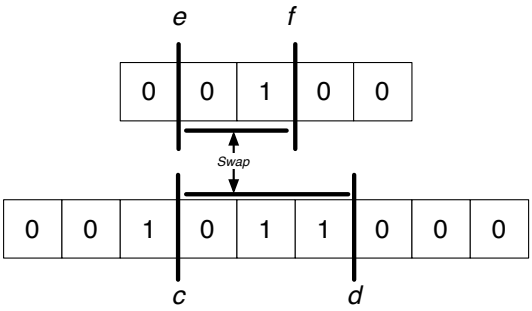

Рис. .29: Двухточечный списочный кроссинговер

эволюции имеет смысл генерировать точки разрыва для двухточечного кроссинговера чаще в конце списка, чем в начале. Или использовать только одноточечный кроссинговер.

Алгоритм для двухточечного списочного кроссинговера также должен выглядеть знакомым:

Алгоритм 60 Двухточечный списочный кроссинговер

1: $\overrightarrow{v} \leftarrow$ первый список $\langle v_1, v_2, \ldots, v_l \rangle$ для скрещивания
2: $\overrightarrow{w} \leftarrow$ второй список $\langle w_1, w_2, \ldots, w_l \rangle$ для скрещивания

3: $c \leftarrow$случайное целое, равномерно распределенное в диапазоне от 1 до l
4: $d \leftarrow$случайное целое, равномерно распределенное в диапазоне от 1 до l
5: $e \leftarrow$случайное целое, равномерно распределенное в диапазоне от 1 до k
6: $f \leftarrow$случайное целое, равномерно распределенное в диапазоне от 1 до k
7: **if** $c > d$ **then**
8: Поменять местами значения c и d
9: **end if**
10: **if** $e > f$ **then**
11: Поменять местами значения e и f
12: **end if**
13: $\overrightarrow{x} \leftarrow$вырезанный участок из $\overrightarrow{v}$ от v_c до v_d
14: $\overrightarrow{y} \leftarrow$вырезанный участок из $\overrightarrow{w}$ от w_e до w_f
15: Вставить $\overrightarrow{y}$ в $\overrightarrow{v}$ вместо вырезанного участка
16: Вставить $\overrightarrow{x}$ в $\overrightarrow{w}$ вместо вырезанного участка
17: **return** $\overrightarrow{v}$ и $\overrightarrow{w}$