4. Представление данных

Большая часть описываемых далее методов обычно применяется в популяционных алгоритмах. Поэтому далее будем, как правило, использовать варианты терминов для эволюционных вычислений: *особь* вместо *потенциальное решение*, *приспособленность* вместо *качество* и т.д.

Под **представлением данных** (***representation***) особи понимается подход к ее формированию, изменению и способу представления для вычисления приспособленности. Несмотря на то, что мы часто говорим о представлении данных, как о структуре данных, используемой для описания особи (вектор, дерево и т.д.), полезно считать, что представление это не тип данных, а просто две следующие функции:

- Функция **инициализация**, применяемая для генерации случайной особи.
- Функция **Tweak**, которая использует одну (или более) особей и незначительно ее модифицирует.

К этому можно также добавить:

- Функция **вычисление приспособленности**.
- Функция **Copy**.

Эти функции определяют все участки программы, в которых во многих оптимизационных алгоритмах осуществляется работа с внутренним содержимым особи. Во всех других случаях особи для алгоритмов представлены в виде черного ящика. Работая с этими функциями особым образом, можно полностью изолировать особенности представления данных от всей остальной системы.

Успех или неудача метаэвристического метода в значительной степени зависят от представления данных особей, т.к. это представление, в частности функция Tweak, имеет огромное влияние на траекторию процесса оптимизации, определяя маршрут по ландшафту приспособленности (т.е. функции качества). Большая часть шаманства используется для создания такого подходящего представления данных, которое улучшает (или по крайней мере не *ухудшает*) **гладкость** (***smoothness***) этого ландшафта. Как уже говорилось ранее, критерий глдакости можно приблизительно определить следующим образом: особи, похожие друг на друга, демонстрируют схожее поведение (и поэтому имеют близкие значения приспособленности), в то время как непохожие особи, ведут себя по-разному.

Чем более гладким является ландшафт, тем меньше он имеет холмов и тем больше напоминает одномодальный ландшафт, как показано на рис. 16. Напомним, что это гладкость не является *достаточным* критерием, т.к. **функции типа иголка-в-стогу-сена** (***needle-in-the-haystack***) или (еще хуже) **обманчивые** (***deceptive***) функции являются очень гладкими, однако могут представлять существенные трудности для оптимизационных алгоритмов.

Говоря, что особи *похожи*, имеется в виду, что у них схожие **генотипы** (***genotypes***), а когда говорится, что особи имеют схожее поведение, считается, что их **фенотипы** (***phenotypes***) похожи[1]. Что значит, что генотипы *похожи*? В общем случае генотип A похож на генотип B, если существует высокая вероятность преобразования от A к B (и наоборот) посредством функции Tweak. Таким образом, эти элементы схожи не потому, что их генотипы *выглядят* похоже, а потому что они в пространстве расположены близко друг к другу *по отношению к выбранной операции Tweak*.

Весьма заманчиво считать, что система стохастической оптимизации работает в пространстве генотипов, а потом преобразует генотипы в фенотипы для их оценки. Однако с точки зрения представления данных целесообразнее думать наоборот: естественная форма особи — это фенотип, а когда необходимо создать новую особь, фенотип преобразуется в генотип, выполняется Tweak, а затем

[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_1.pdf`

[1] Вспомним, что, по крайней мере для эволюционных вычислений, слово «генотип»относится к тому, как особь организована с точки зрения генетических операторов (вектор это или дерево), а слово «фенотип»обозначает *как* (но не *как хорошо*) особь *проявляет* себя при вычислении приспособленности.

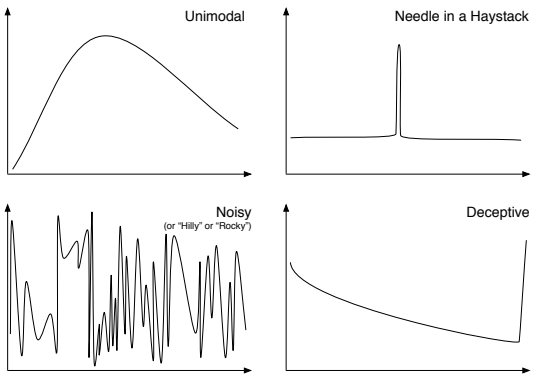

Рис. .16: Четыре ландшафта приспособленности. Повтор рис. 6

производится обратное преобразование к фенотипу. Преобразование фенотип → генотип принято называть **кодированием** (***encoding***), а обратное преобразование — **декодированием** (***decoding***). Поэтому весь процесс можно изобразить следующим образом:

Родительский фенотип → Кодирование → Tweak → Декодирование → Фенотип потомка

Такой подход позволяет лучше видеть недостатки плохих вариантов кодирования. Предположим, что фенотип особей в силу ряда причин, представлен конфигурациями кубика Рубика. Необходимо, чтобы оператор Tweak производил небольшие изменения, например, поворот грани и т.д. Если использовать генотип в виде кубика Рубика, то получаем комплект под ключ: оператор Tweak уже делает, что от него требуется. Но представим, что операция кодирования выглядит так:

Родитель → 20 необычных изменений → Tweak → 20 обратных изменений → Потомок

Теперь после 20 изменений один поворот грани (Tweak) приведет к *гигантским* последствиям после двадцати обратных изменений. Это может привести к тому, что потомок практически не будет зависеть от родителя, т.е. будет случайным. Мораль: необходимо использовать такой механизм кодирования/декодирования, который не развалит пространство фенотипов, в результате применения тщательно настроенного и «гладкого»оператора Tweak.

И причина этого носит не только академический характер. В прошлом исследователи генетических алгоритмов *везде* применяли кодирование бинарными векторами фиксированной длины. Основным аргументом являлось следующее: если существует только один возможный генотип, то можно разработать канонический генетический алгоритм в виде самостоятельной библиотеки функций, и единственное значимое отличие при решении конкретной задачи будет заключаться в процедуре кодирования. Однако, как оказалось, это была не такая уж и хорошая идея. Предположим, что особь включает одно целое число от 0 до 15, представленное 4 битным вектором. Функция приспособленности показана справа. Заметим, что она возрастает до 8, а потом «падает с пика»при 9. Такая функция приспособленности обладает плохим свойством для генотипа, известным в ГА-сообществе как **Хеммингов пик** (***Hamming cliff***). Хеммингов пик возникает в случае, когда для *небольшого* изменения фенотипа, необходимо сделать очень *большое* изменение генотипа. Например, для мутации от 7 (0111) до 8 (1000) нужно перевернуть подряд 4 бита. Функцию, показанную справа сложно оптимизировать, потому что чтобы дойти до 8, необходимо сначала получить либо 7 (требующую 4 удачных мутации), либо 9 или 10 (которые не будут выбраны из-за низкой приспособленности).

Табл. 2. Функция приспособленности с Хеммингoвыми пиками

Фенотип	Генотип	Код Грея	Приспособленность
0	0000	0000	0
1	0001	0001	1
2	0010	0011	2
3	0011	0010	3
4	0100	0110	4
5	0101	0111	5
6	0110	0101	6
7	0111	0100	7
8	1000	1100	8
9	1001	1101	0
10	1010	1111	0
11	1011	1110	0
12	1100	1010	0
13	1101	1011	0
14	1110	1001	0
15	1111	1000	0

Рассмотрим, что будет, если представлять особь не бинарным кодированием, показанным в таблице, а с помощью **кода Грея**[2] (***Gray code***), показанного в соседнем столбце. У этого способа кодирования есть интересное свойство: каждое последующее число отличается от предыдущего на один бит. И 15 отличается от 0 также на 1 бит. Поэтому имея 7 (код Грея 0100), можно легко осуществить мутацию до 8 (код Грея 1100). Проблема Хеммингових пиков решена. Кстати, код Грея легко реализовывается:

Алгоритм 40 Код Грея

1: $\overrightarrow{v}$ ←булевский вектор, содержащий число в стандартном бинарном представлении $\langle v_1, v_2, \ldots v_l \rangle$ для преобразования в код Грея

2: $\overrightarrow{w}$ ←Copy($\overrightarrow{v}$)
3: **for** i от 2 до l **do**
4: **if** $v_{i-1} = true$ **then**
5: $w_i \leftarrow \neg(v_i)$
6: **end if**
7: **end for**
8: **return** $\overrightarrow{v}$

Основной задачей этого примера *не* является убеждение в использовании кодов Грея, т.к. можно сконструировать функцию приспособленности сложную и для кодов Грея, к тому же использование кода Грея постепенно становится в некоторой степени старомодным. Основной задачей является продемонстрировать понятие гладкости и его значимость. **Если кодировать особь так, что малые изменения генотипа (такие как переворачивание одного бита), скорее всего приведут к малым изменениям приспособленности, то это может оказаться полезным для алгоритма оптимизации.**

Одним из эвристических подходов к сглаживанию ландшафта приспособленности является создание генотипа, который как можно больше похож на фенотип: если фенотип является структурой графа, то пусть и генотип будет представлять структуру графа. В этом случае функция приспособленности все равно может быть холмистой, но она не будет *еще более холмистой*, чем при использовании неудачного способа кодирования. Но помните, что при таком подходе представление данных считается структурой данных, хотя в действительности это не так. Это в основном две функции: функция инициализации и функция Tweak.

В представлении данных больше искусства, чем науки Как определить функцию Tweak для графа, чтобы получить гладкость ландшафта приспособленности? Я серьезно. Некоторые способы представления (часто булевские или вещественные векторы фиксированной длины) очень понятны и для них получено немало теоретических результатов. Однако многие способы представления все еще

[2] В честь Фрэнка Грея (*Frank Gray*), разработавшего этот код в 1947 в компании Bell Labs, чтобы уменьшить количество ошибок в телефонном коммутационном узле.

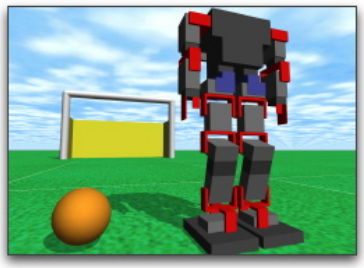

Рис. .17: Наш двуногий робот

являются эвристическими. Не нужно рассматривать многие алгоритмы и идеи в этом разделе как *руководство к действию*, или хотя бы *рекомендации*. Лучше считать их частными *предложениями* возможных способов представления, сохраняющих гладкость. Рассмотрение мы начнем с простых и хорошо понятных представлений, которые уже неоднократно встречались ранее: с векторов.

4.1. Векторы

Чтобы избежать недоразумений, под *векторами* (*vectors*) будем понимать одномерные массивы фиксированной длины. Списки переменной длины будут рассмотрены в разделе 4.4. Векторы обычно бывают трех типов: булевские, целочисленные и вещественные[3]. Первые два типа уже достаточно много обсуждались ранее. Разделы 3.2.1 и 3.2.2 содержат обсуждение методов мутации и кроссинговера булевских векторов, а в разделе 3.1.1 представлены инициализация и мутация вещественных векторов с использованием гауссовской свертки.

Для целочисленных векторов необходимо отдельно рассмотреть одну особенность, а именно, что представляют элементы таких векторов? Определяют ли они множество неупорядоченных объектов (1=Китай, 2=Англия, 3=Франция, ...) или же формируют метрическое пространство (коэффициенты IQ, номера домов, оценки за экзамены), в котором *расстояние* между, скажем, 4 и 5 больше, чем меньше 1 и 5? Является ли пространство метрическим часто важно для выполнения мутации.

4.1.1 Инициализация и смещение

Создание случайных начальных векторов обычно подразумевает выбор значения каждого элемента v_i вектора по равномерному закону в заданном диапазоне. Если имеется дополнительная информация о задаче, то можно **сместить** (***bias***) инициализацию системы, выбирая значения элементов векторов из некоторого более узкого интервала. Например, если считается, что более хорошие решения обычно удовлетворяют выражению $v_1 = v_2 \times v_3$, то можно принудительно генерировать новые элементы векторов из соответствующих областей диапазона значений.

Другим способом задать смещение для векторов является инициализация популяции созданными «вручную»особями. К примеру, мои студенты пытались оптимизировать векторы, определяющие как двуногий робот будет ходить, пинать мяч и т.д. Эти векторы задавали углы между сочленениями и законы движения приводов робота. Вместо того, чтобы начинать со случайных значений, большинство из которых не имело смысла, они предпочли использовать 3D систему для слежения за движениями одного из студентов, который выполнял необходимые действия. Затем полученные параметры кодировались в виде углов между сочленениями и использовать для инициализации популяции.

Несколько замечаний. Во-первых, смещение может быть опасно. Можно считать, что *знаешь* где находятся наилучшие решения, однако это может быть не так. Поэтому если начальная конфигурация будет смещенной, то это может осложнить поиск правильного ответа. Нужно оценивать последствия. Во-вторых, если даже выбирается смещенная система, то возможно будет более предусмотрительным начать со значений, которые *не все* или *не совсем* являются смещенными. Разнообразие полезно, особенно на начальном этапе.

[3] Однако ничто не мешает определять векторы деревьев, или правил, или вектор, в котором часть элементов вещественного типа, а часть — булевского и т.д. (на самом деле, мы встретимся с векторами деревьев в разделе 4.3.4). В таких случаях нужно просто быть внимательнее с механизмами инициализации и мутации.

4.1.2 Мутация

Мутация вещественных векторов редко производится чем-то отличным от гауссовской свертки (или иной процедуры наложения шума, основанной на использовании некоторого распределения). Аналогично битовые векторы обычно мутируют с использованием битовой мутации. Для целочисленных векторов мутация зависит от задачи. Если представление рассматривает целые числа как элементы множества, то лучшее, что можно сделать, это случайно изменить каждый элемент с некоторой вероятностью:

Алгоритм 41 Случайная целочисленная мутация

1: $\overrightarrow{v}$ ←мутирующий целочисленный вектор $\langle v_1, v_2, \ldots v_l \rangle$
2: p ←вероятность случайного изменения элемента {Вероятно, стоит использовать p равное $1/l$ или меньше}

3: **for** i от 1 до l **do**
4: **if** $p \geq$ случайное число, равномерно распределенное на интервале [0;1] **then**
5: $v_i \leftarrow$ новое случайное допустимое целое значение
6: **end if**
7: **end for**
8: **return** $\overrightarrow{v}$

Если целые числа принадлежат метрическому пространству, то можно осуществлять их мутацию аналогично гауссовской свертке так, чтобы изменения значений были небольшими. Одним из многих отличных способов для этого является подбрасывание монетки до тех пор, пока не выпадет решка, а затем делаем случайный шаг, длина которого равна количеству подбрасываний[4]. Это создает шум, центрированный относительно начального значения.

Алгоритм 42 Мутация случайным блужданием

1: $\overrightarrow{v}$ ←мутирующий целочисленный вектор $\langle v_1, v_2, \ldots v_l \rangle$
2: $p \leftarrow$ вероятность случайного изменения элемента {Вероятно, стоит использовать p равное $1/l$ или меньше}
3: $b \leftarrow$ вероятность переворачивания монетки {Можно задать большое значение b, если диапазон разрешенных значений велик. Это увеличит длительность случайного блуждания}

4: **for** i от 1 до l **do**
5: **if** $p \geq$ случайное число, равномерно распределенное на интервале [0;1] **then**
6: **repeat**
7: $n \leftarrow$ либо +1, либо -1, выбранное случайно
8: **if** $v_i + n$ попадает в диапазон разрешенных значений **then**
9: $v_i \leftarrow v_i + n$
10: **else**
11: $v_i \leftarrow v_i - n$
12: **end if**
13: **until** $b \geq$ случайное число, равномерно распределенное на интервале [0;1]
14: **end if**
15: **end for**
16: **return** $\overrightarrow{v}$

Точечная мутация Все методы мутации, описанные ранее, обладают одним общим свойством: *каждый* ген в геноме мутирует независимо от других генов. Возможно, вы уже подумали о другом подходе: выбираем случайно один ген, осуществляем с ним мутацию и готово (или можно выбрать случайно n генов и провести мутацию с ними). Такие методы **точечной мутации** (***point mutation***) иногда могут оказаться полезными, но часто они опасны.

Для начала рассмотрим положительные стороны. Существуют задачи, в которых можно улучшить решение изменив один ген, однако если одновременно меняются несколько генов, даже на

[4] Заметьте, я только что выдумал этот способ мутации, хотя, наверное, он совсем не плох. И, возможно, кто-то уже изобрел его до меня.

небольшую величину, то повысить качество уже сложнее. Картинка Моны Лизы на обложке представляет пример такой задачи: геном состоит из m полигонов со случайными цветами. Изменяя в каждый момент времени только один полигон, совсем немного, можно по прошествии определенного времени получить изображение Моны Лизы. Если подвергать мутациям n полигонов (или все m полигонов), даже совсем немного, то создать более приспособленного потомка будет гораздо сложнее.

Но будьте бдительны: легко создать задачу, в которой точечные мутации будут вредить. Рассмотрим простые булевские особи вида $\langle x, y\rangle$, где x и y могут быть либо 1, либо 0, и будем использовать простой алгоритм локального поиска (или (1+1), если хотите). В задаче используется функция приспособленности, представленная таблицей 3, а наша отважная первая особь равна $\langle 0, 0\rangle$, что соответствует приспособленности равной 5. Функция мутации инвертирует один ген. Если изменить ген x, то окажемся в точке $\langle 1, 0\rangle$, приспособленность в которой равна -100, и скорее всего эта точка будет отвергнута. С другой стороны, при изменении только y, придем в точку $\langle 0, 1\rangle$, где приспособленность также равна -100. Если не инвертировать *оба бита* одновременно, то добраться до оптимума в точке $\langle 1, 1\rangle$ невозможно.

Табл. 3. Простейшая булевская функция приспособленности, «враждебная»к точечной мутации

		x	
		0	1
y	0	5	-100
	1	-100	10

Однако наш оператор мутации этого не допустит. Проблема в том, оператор точечной мутации *не является глобальным*, а способен осуществлять горизонтальные движение в пространстве, поэтому он не может достичь областей пространства, в которые нельзя добраться в один присест. Итог: точечная мутация может оказаться полезной, но нужно знать ее последствия.

4.1.3 Рекомбинация

До этого момента было описано три вида векторной рекомбинации общего назначения: **одно-** и **двухточечный** и **однородный кроссинговер**. Кроме этого были представлены два вида рекомбинации для вещественных чисел: **линейная** и **промежуточная**. Естественно, что их можно применить два этих алгоритма для целых чисел в метрическом пространстве.

Алгоритм 43 Целочисленная линейная рекомбинация

$\overrightarrow{v} \leftarrow$ первый вектор $\langle v_1, v_2, \ldots v_l\rangle$ для скрещивания
$\overrightarrow{w} \leftarrow$ второй вектор $\langle w_1, w_2, \ldots w_l\rangle$ для скрещивания
$p \leftarrow$ положительное число, определяющее разброс значений генов потомков вдоль числовой оси

$\alpha \leftarrow$ случайное число из интервала $[-p; 1+p]$
$\beta \leftarrow$ случайное число из интервала $[-p; 1+p]$
for i от 1 до l **do**
  repeat
    $t \leftarrow \alpha v_i + (1-\alpha) w_i$
    $t \leftarrow \beta w_i + (1-\beta) v_i$
  until значения $\lfloor t + 1/2\rfloor$ и $\lfloor s + 1/2\rfloor$ находятся в заданных пределах
  $v_i \leftarrow \lfloor t + 1/2\rfloor$
  $w_i \leftarrow \lfloor s + 1/2\rfloor$
end for
return $\overrightarrow{v}$ и $\overrightarrow{w}$

Вместо применения описанных алгоритмов можно осуществить мутацию или кроссинговер, учитывая особенности *фенотипа*. Например, что если фенотипом является **матрица**, и для ее представления используются векторы? Тогда, возможно, оператор скрещивания должен принимать во внимание двумерный вид фенотипа. Можно предложить оператор рекомбинации, который производит обмен прямоугольных областей:

$$\left[\begin{array}{cc|c} 1 & 4 & 7 \\ 9 & 2 & 3 \\ \hline 8 & 5 & 6 \end{array}\right] \text{ скрещивается с } \left[\begin{array}{cc|c} 21 & 99 & 46 \\ 31 & 42 & 84 \\ \hline 23 & 67 & 98 \end{array}\right] \rightarrow \left[\begin{array}{ccc} 1 & 4 & 46 \\ 9 & 2 & 84 \\ 23 & 67 & 98 \end{array}\right]$$

Алгоритм 44 Промежуточная линейная рекомбинация

$\overrightarrow{v} \leftarrow$ первый вектор $\langle v_1, v_2, \ldots v_l \rangle$ для скрещивания
$\overrightarrow{w} \leftarrow$ второй вектор $\langle w_1, w_2, \ldots w_l \rangle$ для скрещивания
$p \leftarrow$ положительное число, определяющее разброс значений генов потомков вдоль числовой оси

for i от 1 до l **do**
  repeat
    $\alpha \leftarrow$ случайное число из интервала $[-p; 1+p]$
    $\beta \leftarrow$ случайное число из интервала $[-p; 1+p]$
    $t \leftarrow \alpha v_i + (1-\alpha) w_i$
    $t \leftarrow \beta w_i + (1-\beta) v_i$
  until значения $\lfloor t + 1/2 \rfloor$ и $\lfloor s + 1/2 \rfloor$ находятся в заданных пределах
  $v_i \leftarrow \lfloor t + 1/2 \rfloor$
  $w_i \leftarrow \lfloor s + 1/2 \rfloor$
end for
return $\overrightarrow{v}$ и $\overrightarrow{w}$

Отлично: больше не будем рассматривать всякие сложные способы кодирования. Помните рассуждения о ценности гладкости целевой функции? Всегда имейте в виду этот фактор, т.к. при работе с более капризными способами представления гарантировать гладкость становится очень сложно.