

4.2 Прямое кодирование графов

Графы представляют *наибольшие сложности* для кодирования, но будет полезнее обсудить их, прежде чем идти дальше. Зачем нужно искать граф с оптимальной структурой? Графы используются для представления многих вещей: нейронные сети, конечные автоматы или сети Петри или другие простые вычислительные устройства, электрические цепи, отношения между людьми и т.д. Соответственно существуют различные типы структуры графов: ориентированные, неориентированные, графы с помеченными ребрами или вершинами, графы с весами (числами) ребер вместо меток, рекуррентные, прямого распространения (нерекуррентные), разреженные или плотные, планарные графы и др. Все зависит от задачи. Множество механизмов функции *Tweak* должны учитывать ограничения на структуру графа, определенные исследователем.

Для начала отметим, что если структура графа фиксирована и необходимо подобрать значения весов ребер или метки, то особого способа кодирования не нужно. Например, если разрабатывается нейронная сеть с заданным набором связей, то нет необходимости искать ее структуру (она уже определена!). Нужно просто найти значения весов связей. Если связей 100, то требуется оптимизировать вектор 100 вещественных переменных, по одной на каждый вес связи, и готово. Поэтому рассматриваемые далее «способы представления графов» используются для **кодирования графов с произвольной структурой**. Такие структуры используются уже очень давно. Ларри Фогель (*Larry Fogel*) разработал эволюционное программирование (*Evolutionary Programming*), вероятно, самый первый эволюционный алгоритм, специально для поиска структуры графа, соответствующей структуре конечного автомата¹.

Существует два основных способа кодирования структуры графа (а также ряда других сложных структур): **прямое кодирование** (*direct encoding*) и **косвенное** (*indirect, developmental*). При прямом кодировании информация для каждого ребра и вершины графа хранится в явном виде. В косвенном кодировании способ представления задает небольшую программу или набор правил некоторого вида, которые, будучи исполненными, «выращивают» структуру графа.

Зачем нужно косвенное кодирование? Иногда для того, чтобы иметь возможность скрещивания определенных свойств структуры графа, описанных подмножеством взаимосвязанных правил. Иногда потому, что если одни правила рекурсивно запускают на выполнение другие, то некоторые наборы правил можно рассматривать как *функции* или *модули*, которые всегда создают один и тот же подграф. Таким образом, если структура графа содержит много повторяющихся элементов, то можно воспользоваться этим преимуществом косвенного кодирования, получив в ходе эволюции одну единственную функцию для генерации одного такого элемента, вместо того, чтобы снова и снова переоткрывать кодирование этого подграфа в процессе поиска. Если граф содержит мало повторов (например, значения весов связей нейронной сети редко повторяются) и имеет плотную структуру, то более разумным выбором будет прямое кодирование. Поскольку косвенные способы кодирования представляют граф в виде структуры данных, отличной от графа, (как дерево, или набор правил, или список инструкций для построения графа и т.д.), то мы обсудим их позднее (в Разделах 4.3.6 и 4.5). А пока рассмотрим прямые способы кодирования.

Самым простым способом прямого кодирования является **полная матрица смежности** (*full adjacency matrix*). Предположим, что задано максимальное количество вершин графа. Например, необходимо задать рекуррентную структуру ориентированного графа с максимум 5 вершинами и не более 1 ребра между узлами, для каждого направления. Кроме этого, положим, что возможны петли, и задача заключается в поиске весов ребер. Тогда структуру графа можно представить просто в виде матрицы смежности 5×5 , описывающей все ребра:

$$\begin{bmatrix} 0.5 & 0.7 & -0.1 & 0.2 & Off \\ Off & -0.5 & -0.8 & 0.4 & Off \\ 0.6 & 0.7 & 0.8 & Off & -0.4 \\ -0.1 & Off & Off & 0.2 & Off \\ 0.2 & Off & -0.7 & Off & Off \end{bmatrix}$$

«Off» в ячейке $\langle i, j \rangle$ означает «вершины i и j не связаны». Если необходимо задать граф с меньше чем 5 вершинами, то необходимо установить значение «Off» для *всех* входных и выходных ребер

⁰Перевод 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 присыпать по адресу yurytsoy@gmail.com

Данный текст доступен по адресу: http://qai.narod.ru/GA/meta-heuristics_4_2.pdf

¹См. ссылку 9 на стр. 7 Главы 3 с названием диссертации Ларри, в которой были развиты эти идеи

этой вершины. Такая матрица может быть представлена различными способами. Рассмотрим два из них. Во-первых, можно использовать вектор из 25 элементов, хранящий все веса, в котором «Off» обозначено как 0.0. Либо можно представить матрицу как *два* вектора, один из которых вещественный и хранит значения весов, а второй — булевский, задающий активно ли ребро («On») или нет («Off»). В любом случае можно использовать стандартные операторы кроссинговера и мутации, хотя необходимо быть осторожным при изменении значений «Off». При использовании сразу двух векторов такая необходимость отпадает. Для случая с одним вектором можно создать модифицированный алгоритм гауссовской свертки, который лишь *иногда* включает и отключает ребра:

Алгоритм 45 Гауссовская свертка с учетом нулевых значений

```

1:  $\vec{v} \leftarrow$  вектор  $\langle v_1, v_2, \dots, v_l \rangle$  для свертки
2:  $p \leftarrow$  вероятность изменения активности ребра с «On» на «Off» и обратно
3:  $\sigma^2 \leftarrow$  дисперсия гауссовского распределения для свертки
4:  $\min \leftarrow$  минимальное допустимое значение элемента вектора
5:  $\max \leftarrow$  максимальное допустимое значение элемента вектора

6: for  $i$  от 1 до  $l$  do
7:   if  $p >$  случайное число равномерно распределенное на интервале [0;1] then
8:     if  $v_i = 0.0$  then
9:        $v_i \leftarrow$  случайное число равномерно распределенное на интервале [0;1]
10:    else
11:       $v_i \leftarrow 0.0$ 
12:    end if
13:    else if  $v_i \neq 0.0$  then
14:      repeat
15:         $n \leftarrow$  нормально распределенное случайное число из  $N(0, \sigma^2)$  {См. алгоритм 12}
16:        until  $\min \leq v_i + n \leq \max$ 
17:         $v_i \leftarrow v_i + n$ 
18:    end if
19:  end for
20: return  $\vec{v}$ 

```

Недостатком этого подхода является то, что после дезактивации ребра («Off»), при его последующем включении («On») оптимизированный вес окажется потерянным. Возможно, подход с двумя векторами даст более хороший результат.

Если максимальный размер графа не задан, то, вероятно, необходимо использовать **произвольную структуру ориентированного графа**, которая использовалась и раньше (в ЭП), но была популяризирована алгоритмом **GNARL**² Питера Ангелина (*Peter Angeline*), Грега Сандерса (*Greg Saunders*) и Джордана Поллака (*Jordan Pollack*). Здесь способ кодирования представлен уже не вектором, а графом, записанным любым удобным образом. Для работы с ним требуется настроить оператор инициализации и операторы кроссинговера и мутации для добавления и удаления узлов и ребер, изменения меток ребер и узлов и т.д.

Схожий подход используется в методе NEAT³ Кеннета Стенли и Ристо Мииккулайнена для оптимизации нейронных сетей прямого распространения. В NEAT граф представлен двумя множествами, одно для узлов, а второе для ребер. Для каждого узла хранится его номер и объявление назначения (в нейросетевой терминологии: входной, скрытый или выходной узел). Данные о ребрах более интересны: помимо всего прочего, в них хранятся номера узлов, которые соединены данным ребром, вес ребра, а также *дата рождения*: уникальный номер, определяющий на каком этапе это ребро было создано. Дата рождения оказывается полезной при поиске ребер, которые могут быть слиты при кроссинговере, см. обсуждение в Разделе 4.2.3.

4.2.1 Инициализация

При создании структуры графа требуется лишь знать, какой тип графа необходим с точки зрения исследователя. Сначала требуется определить количество узлов и ребер. Можно задать эти парамет-

² Peter J. Angeline, Gregory M. Saunders, and Jordan P. Pollack, 1994, An evolutionary algorithm that constructs recurrent neural networks, IEEE Transactions on Neural Networks, 5(1), 54–65.

³ Kenneth O. Stanley and Risto Miikkulainen, 2002, Evolving neural network through augmenting topologies, Evolutionary Computation, 10(2), 99–127.

ры с использованием некоторого распределения — например, равномерного от 1 до какого-нибудь большого значения. Или можно рассмотреть распределение, предлагающее небольшие значения, такое как геометрическое распределение. Данное распределение задается подбрасыванием монетки до тех пор, пока с вероятностью p не выпадет решка:

Алгоритм 46 Генерация значения из геометрического распределения

```

1:  $p \leftarrow$  вероятность генерации большого значения
2:  $m \leftarrow$  минимальное возможное значение

3:  $n \leftarrow m-1$ 
4: repeat
5:    $n \leftarrow n + 1$ 
6: until  $p <$  случайное число из диапазона  $[0; 1]$ 
7: return  $n$ 
```

Чем больше p , тем в среднем больше n , учитывая уравнение $E(n) = m + p/(1-p)$. К примеру, если $m = 0$ и $p = 2/3$, то n в среднем равно 3, в то время как при $p = 19/20$ в среднем $n = 19$. Учтите, что это при использовании этого распределения получается большое количество малых значений. Его легко вычислить, но в ряде случаев желательно использовать менее склоненное распределение.

После определения количества узлов и ребер, можно приступить к созданию самого графа, сначала создав узлы, а затем проложив ребра:

Алгоритм 47 Построение графа, начиная с узлов

```

1:  $n \leftarrow$  количество узлов
2:  $e \leftarrow$  количество ребер
3:  $f(j, k, Nodes, Edges) \leftarrow$  функция, возвращающая true если ребро от узла  $j$  к  $k$  разрешено

4: Набор узлов  $N \leftarrow \{N_1, \dots, N_n\}$  {Новые узлы}
5: Набор ребер  $E \leftarrow \{\}$ 
6: for каждый узел  $N_i \in N$  do
7:   ProcessNode( $N_i$ ) {Определение метки узла и т.д.}
8: end for
9: for  $i$  от 1 до  $e$  do
10:   repeat
11:      $j \leftarrow$  случайное число из диапазона  $[0; n]$ 
12:      $k \leftarrow$  случайное число из диапазона  $[0; n]$ 
13:     until  $f(j, k, Nodes, Edges)$  вернет 'true'
14:      $g \leftarrow$  новое ребро из  $N_j$  до  $N_k$ 
15:     ProcessEdge( $g$ ) {Определение метки, веса, направления узла и т.д.}
16:      $E \leftarrow E \cup \{g\}$ 
17: end for
18: return  $N, E$ 
```

Заметьте функции `ProcessNode` и `ProcessEdge`, которые раздают метки и веса узлам и ребрам. Сложность работы с данным алгоритмом заключается в том, что в результате может получиться несвязный граф. Альтернативой является «выращивание» графа добавлением к данному новых узлов, при этом каждый раз создаются ребра, соединяющие новые узлы с существующими:

Распределение вероятности для количества связей необходимо определить для каждого узла. Заметьте, что если не соблюсти осторожность, то узлы, сгенерированные ранее, будут иметь больше ребер, чем более поздние узлы, поэтому нужно либо сначала задавать меньшее количество ребер для каждого узла, либо ограничивать количество ребер константой.

4.2.2 Мутация

Одним из способов осуществления мутации произвольного графа является определение количества n мутаций, а затем n раз выполнение одной из следующих операций:

- Удалить случайное ребро с вероятностью α_1

Алгоритм 48 Построение графа с пошаговым добавлением узлов

```
1:  $n \leftarrow$  количество узлов
2:  $f(j, k, Nodes, Edges) \leftarrow$  функция, возвращающая true если ребро от узла  $j$  к  $k$  разрешено
3:  $D \leftarrow$  распределение вероятности для определения количества ребер для узла

4: Набор узлов  $N \leftarrow \{\}$ 
5: Набор ребер  $E \leftarrow \{\}$ 
6: for  $i$  от 1 до  $n$  do
7:    $h \leftarrow$  новый узел
8:   ProcessNode( $h$ )
9:    $N \leftarrow N \cup \{h\}$ 
10:   $p \leftarrow$  случайное целое, выбранное в соответствии с  $D$ 
11:  for  $j$  от 1 до  $p$  do
12:    repeat
13:       $j \leftarrow$  случайное число из диапазона  $[0; \|N\|]$ 
14:      until  $f(j, k, Nodes, Edges)$  вернет 'true'
15:       $g \leftarrow$  новое ребро из  $h$  до  $N_j$ 
16:      ProcessEdge( $g$ )
17:       $E \leftarrow E \cup \{g\}$ 
18:    end for
19:  end for
20: return  $N, E$ 
```

- Добавить случайное ребро с вероятностью α_2 (при использовании NEAT это ребро получит новое число — дату рождения, см. Раздел 4.2.3 далее)
- Удалить узел и все инцидентные ему ребра с вероятностью α_3 (да-а-а!)
- Добавить узел с вероятностью α_4
- Изменить метку узла с вероятностью α_5
- Изменить метку ребра с вероятностью α_6 ... и т.д. ...

... где $\sum_i \alpha_i = 1$. Очевидно, что некоторые из этих операций несут очень большие изменения, и поэтому, возможно, должны иметь меньшую вероятность. Помните, что небольшие изменения генотипа должны повлечь малые изменения значения целевой функции, т.е. более сильные мутации должны происходить реже. И последнее, как выбрать значение n ? Например, можно выбирать его по равномерному закону в диапазоне $1 \dots M$. Или можно снова воспользоваться геометрическим распределением.

4.2.3 Рекомбинация

Кроссинговер на графах привносит такую неразбериху, что многие люди вовсе его не используют. Как вообще можно скрестить графы, чтобы получить осмысленный результат? Другими словами, обмен важными и полезными элементами между особями без использования специализированного кроссинговера будет иметь большой элемент случайности.

Для скрещивания узлов и ребер часто необходимо уметь выбрать подмножество этих элементов. Чтобы это сделать:

Этот алгоритм работает по такому же принципу, как и равномерный кроссинговер или битовая мутация. Однако это может привести к неудовлетворительному распределению подмножеств (относительно исходного множества). Альтернативой является генерация случайного числа по некоторому распределению и выбор подмножества этого размера:

Отметим, что в отличии от многих встречающихся здесь примеров, выбор элемента производится с возвратом — т.е. один и тот же элемент может быть выбран больше одного раза.

Итак, вернемся к кроссинговеру. Один из наивных подходов рассматривает выбор некоторого подмножества узлов и ребер из каждого графа и обмен этими подмножествами. Но что если от графа A к графу B переходит ребро $i \rightarrow j$, а у B нет узла i или j ? Возвращаемся к началу. Еще одним вариантом может быть обмен узлами, и затем обмен инцидентными ребрами с ограничением,

Алгоритм 49 Выбор подмножества

```
1:  $S \leftarrow$  исходное множество  
2:  $p \leftarrow$  вероятность выбора в подмножество  
  
3: Подмножество  $S' \leftarrow \{\}$   
4: for каждый элемент  $S_i \in S$  do  
5:   if  $p \geq$  случайное число, равномерно распределенное на интервале  $[0.0, 1.0]$  then  
6:      $S' \leftarrow S' \cup \{S_i\}$   
7:   end if  
8: end for  
9: return  $S'$ 
```

Алгоритм 50 Выбор подмножества (второй метод)

```
1:  $S \leftarrow$  исходное множество  
2:  $n \leftarrow$  количество элементов в подмножестве  
  
3: Подмножество  $S' \leftarrow \{\}$   
4: for  $i = 1$  до  $n$  do  
5:   if  $p \geq$  случайное число, равномерно распределенное на интервале  $[0.0, 1.0]$  then  
6:      $S' \leftarrow S' \cup \{\text{Случайный элемент из } S, \text{ взятый по схеме с возвратом}\}$   
7:   end if  
8: end for  
9: return  $S'$ 
```

что обмен ребра проводится только если у другого графа есть соответствующий этому ребру узел. Но здесь возникает сложность, которая заключается в том, что можно произвести обмен подграфом, который является несвязанным с остальными узлами графа, а при обмене важные для связности ребра были пропущены.

Третьим вариантом является обмен целыми подграфами. Алгоритм выбора подграфа:

Алгоритм 51 Выбор подграфа

```
1:  $N \leftarrow$  узлы исходного графа  
2:  $E \leftarrow$  ребра исходного графа  
  
3:  $N' \subseteq N \leftarrow$  узлы подграфа (выбранные с помощью операции выбора подмножества)  
4: Подмножество  $E' \leftarrow \{\}$   
5: for каждое ребро  $E' \in E$  do  
6:    $j, k \leftarrow$  узлы, соединенные  $E_i$   
7:   if  $j \in N'$  и  $k \in N'$  then  
8:      $E' \leftarrow E' \cup \{E_i\}$   
9:   end if  
10: end for  
11: return  $N', E'$ 
```

Здесь снова возникает проблема, что добавляемый в результате обмена подграф не будет связан с существующим графом. Возможно, понадобится объединение некоторых узлов исходного графа и подграфа. При слиянии узлов потребуется переименовать ребра, т.к. некоторые инцидентные им вершины перестанут существовать. В данном случае вероятность, что граф и подграф не будут связаны, все равно остается, но она мала. Можно принудительно осуществить слияние для хотя бы одной вершины, чтобы убедиться, что графы связаны. Алгоритм может выглядеть так:

Последний метод, использованный в алгоритме NEAT, это слияние *всех* родительских ребер при формировании потомка. Однако если два ребра имеют одинаковую дату рождения (т.е. изначально соединяли одинаковые вершины), то в NEAT одно из них отбрасывается. Таким образом, подграфы сливаются не произвольно, а в соответствии с тем, как они были созданы. Идея заключается в сохранении структур подграфов и уменьшении случайности кроссинговера.

Здесь мы даже не подошли к тому, как удостовериться, что все ограничения для вашего графа (без петель, без множественных ребер между вершинами и т.д.) соблюdenы в результате кроссинговера.

Алгоритм 52 Случайное слияние двух графов

```
1:  $N \leftarrow$  случайно перемешанные узлы первого графа {Для случайного перемешивания массива  
см. алгоритм 26}  
2:  $N' \leftarrow$  узлы второго графа  
3:  $E \leftarrow$  ребра первого графа  
4:  $E' \leftarrow$  ребра второго графа  
5:  $p \leftarrow$  вероятность слияния узла из  $N$  с узлом из  $N'$   
  
6: for  $l$  от 1 до  $\|N\|$  do  
7:   if  $l = 1$  или  $p \geq$  случайное число из интервала  $[0.0; 1.0]$  then  
8:      $n' \leftarrow$  случайный узел из  $N'$  {Будем сливать узел  $N_l$  с  $n'$ }  
9:     for  $i$  от 1 до  $\|E\|$  do  
10:       $j, k \leftarrow$  узлы, соединенные  $E_i$   
11:      if  $j = N_l$  then  
12:        Изменить  $j$  в  $E_i$  на  $n'$   
13:      end if  
14:      if  $k = N_l$  then  
15:        Изменить  $k$  в  $E_i$  на  $n'$   
16:      end if  
17:    end for  
18:  else  
19:     $N' \leftarrow N' \cup \{N_l\}$  {Не сливаем, а просто добавляем  $N_l$  к новому графу}  
20:  end if  
21: end for  
22:  $E' \leftarrow E' \cup E$   
23: return  $N', E'$ 
```

и мутации. Ну и морока.