3 ПОПУЛЯЦИОННЫЕ МЕТОДЫ[1]

Методы, основанные на использовании популяции, отличаются от методов, описанных в предыдущей главе, тем, что работают с *набором* потенциальных решений, а не с единственным возможным решением. Каждое решение постепенно улучшается и оценивается, и основным отличием от простого параллельного метода локального поиска заключается в том, что это потенциальное решение влияет на то, как будут улучшены другие решения. Это может произойти из-за того, что хорошие решения либо повлияют на то, какие плохие решения будут отброшены, либо на направление улучшений плохих решений.

Нет ничего удивительного в том, что большинство популяционных методов «украло» эту концепцию из биологии. Один из особенно популярных наборов таких методов известен как **Эволюционные вычисления** (**ЭВ**) (**Evolutionary Computation**, **EC**), заимствующий подходы популяционной биологии, генетики и эволюции. Алгоритм, относящийся к этому набору называют **Эволюционным алгоритмом** (**ЭА**) (**Evolutionary Algorithm**, **EA**). Большинство ЭА можно классифицировать на **поколенческие** (**generational**) алгоритмы, в которых производится полное обновление решений на каждой итерации, и **устойчивые** (**steady-state**) алгоритмы, в которых на каждой итерации обновляются только несколько решений. Традиционно ЭА включают **Генетический алгоритм** (**Genetic algorithm**, **GA**) и **Эволюционные стратегии** (**Evolution strategy**, **ES**), причем для каждого вида существуют как поколенческие, так и устойчивые варианты. Кроме этого есть еще несколько других алгоритмов, тоже описываемых нагромождением букв[2].

Поскольку методы ЭВ основываются на биологии, то они активно (и часто неправильно) используют термины из генетики и теории эволюции. Поскольку подобное безобразие стало общепринятым, мы тоже будем использовать эти термины.

Определение 1. Общие термины, используемые в эволюционных вычислениях

особь (individual)	потенциальное решение
потомок и родитель (child, parent)	*потомок* – это улучшенная копия потенциального решения (*родителя*)
популяция (population)	набор потенциальных решений
приспособленность (fitness)	качество
ландшафт приспособленности (fitness landscape)	функция качества
оценка приспособленности (fitness assessment and evaluation)	вычисление приспособленности особи
селекция (selection)	отбор особей на основе их приспособленностей
мутация (mutation)	обычное улучшение. Часто рассматривается как «бесполое» размножение.
рекомбинация или кроссинговер	особое улучшение, которое использует двух родителей,

[1] Перевод раздела из книги Luke S. Essentials of Metaheuristics. A Set of Undergraduate Lecture Notes. Zeroth Edition. Online Version 0.5. October, 2009 (http://cs.gmu.edu/~sean/book/metaheuristics/). Перевел – Юрий Цой, 2009 г.
Любые замечания, касающиеся перевода, просьба присылать по адресу yurytsoy@gmail.com
Данный текст доступен по адресу: http://qai.narod.ru/GA/meta-heuristics_3.pdf

[2] В оригинале эта шутка выглядела изящнее: «a few more alphabet soup subalgorithms». – Прим. перев.

(recombination, crossover)	производит обмен их частями и (как правило) создает двух потомков. Часто рассматривается как «половое» размножение.
размножение (breeding)	создание одного или несколько потомков из популяции родительских особей путем использования итерационного процесса селекции и улучшения (обычно кроссинговер или мутация)
генотип или геном (genotype, genom)	структура данных особи, используемая в процессе размножения
хромосома (chromosome)	генотип в виде массива фиксированной длины
ген (gene)	определенная позиция в хромосоме
аллель (allele)	частное значение гена
фенотип (phenotype)	характеризует особь во время вычисления приспособленности
поколение (generation)	один цикл оценивания, размножения и обновления популяции; либо популяция, создаваемая в каждом таком цикле

В общем случае методы эволюционных вычислений являются **resampling-методами**: новые наборы решений (популяции) создаются на основании свойств предыдущих. Оптимизация с использованием роящихся частиц (**Particle Swarm Optimization**), описанная в Разделе 3.6, наоборот, является примером метода с направленной мутацией, когда потенциальные решения изменяются, но никакого **resampling** по сути не производится.

Простейший поколенческий алгоритм ЭВ начинает работу с создания популяции, к которой итерационно применяются три процедуры. Первая **вычисляет приспособленность** всех особей в популяции. Вторая использует полученную информацию о приспособленностях для **размножения** – получения популяции потомков. Третья некоторым образом **объединяет** популяции родителей и потомков для формирования популяции нового поколения и после этого цикл начинается сначала.

Algorithm 17 *An Abstract Generational Evolutionary Algorithm (EA)*

```
1: P ← Build Initial Population
2: Best ← □                                          ▷ □ means "nobody yet"
3: repeat
4:     AssessFitness(P)
5:     for each individual P_i ∈ P do
6:         if Best = □ or Fitness(P_i) > Fitness(Best) then   ▷ Remember, Fitness is just Quality
7:             Best ← P_i
8:     P ← Join(P, Breed(P))
9: until Best is the ideal solution or we have run out of time
10: return Best
```

Отметим, что в отличие от методов с одним состоянием появилась отдельная функция `AssessFitness` для вычисления приспособленности, поскольку обычно перед размножением необходимо иметь информацию о приспособленности всех особей. Поэтому в алгоритме есть отдельное место для вычисления приспособленности.

Эволюционные алгоритмы в основном отличаются друг от друга тем, как осуществляют операции `Join` и `Breed`. Операция `Breed`, как правило, состоит из двух частей: **селекция** особей из популяции и их улучшение (обычно с использованием **мутации** и **рекомбинации**) для получения потомства. Типичная операция `Join` либо полностью заменяет родителей потомками, либо включает в следующее поколение наиболее приспособленных родительских особей и потомков[3].

Инициализация популяции. Все описанные здесь алгоритмы используют одни и те же процедуры инициализации, поэтому имеет смысл дать общие рекомендации. Как правило, в результате инициализации просто создается n случайных особей. Однако если имеется информация о «хороших» начальных областях пространства поиска, то можно **адаптировать**[4] случайное создание особей в сторону генерации особей из этих областей. На самом деле, можно даже **сгенерировать**[5] начальную популяцию, по крайней мере, частично, с использованием особей с заранее заданными параметрами. С этими методами следует быть осторожным: часто вы можете считать, что знаете, где находятся эти хорошие области, но есть немалая вероятность, что вы ошибаетесь. Не кладите все яйца в одну корзину: пусть инициализация использует существенную долю равномерной случайности. Более подробны мы будем обсуждать этот вопрос, когда будем рассматривать способы представления данных (в Разделе 4.1.1).

Также имеет смысл усилить разнообразие путем проверки, что каждая особь в начальной популяции является уникальной. Однако это не значит, что при создании каждой особи необходимо пробегать по всей популяции и производить сравнение со всеми уже созданными особями: это имеет значительную вычислительную сложность $O(n^2)$ и вообще глупо. Вместо этого создавайте хеш-таблицы, в которых особи представляют собой ключи, а в качестве значения хранится что-нибудь другое. При создании особи нужно проверять содержится ли соответствующий ключ в хеш-таблице. Если да, то особь отбрасывается и генерируется новая. В противном случае особь добавляется в популяцию, а в хеш-таблицу заносится новый ключ. Этот метод имеет сложность порядка $O(n^2)$.

3.1 ЭВОЛЮЦИОННЫЕ СТРАТЕГИИ

Эволюционные стратегии (**ЭС**) (**Evolution Strategies**, **ES**) были разработаны Инго Рехенбергом и Хансом-Паулем Швефелем в Берлинском техническом университете в середине 1960-х[6]. ЭС используют простую процедуру селекции, называемую **селекцией усечением** (***Truncation Selection***), а в качестве оператора для улучшения особей применяется (обычно) только мутация.

Одним из простейших алгоритмов ЭС является **(**m, l**)** алгоритм. В большинстве случаев начинаем с популяции из l особей, созданных случайно. Итерации эволюционного поиска производятся следующим образом. Сначала вычисляем значения приспособленности для всех особей. Затем оставляем в популяции только m самых приспособленных особей (это и называется Селекцией усечением). Каждая из этих m особей производит l /m потомков с

[3] Операцию `Join` можно рассматривать как разновидность селекции, которая используется для определения того, какие родительские особи и потомки будут формировать следующее поколение, хотя на практике эта операция обычно работает проще. Такой общий взгляд на операцию `Join` часто называют **борьба за выживание** (**survival selection**), при этом селекция на этапе размножения часто называется **селекцией родительских особей** (**parent selection**).

[4] В оригинале было более неоднозначное слово «bias». – *Прим. перев.*

[5] И опять отход от оригинала, там было «**seed**». – *Прим. перев.*

[6] Ingo Rechenberg, 1973, Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Fromman-Holzbook, Stuttgart, Germany. На немецком!

использованием обычного оператора мутации. После этого получаем l потомков. Операция `Join` выглядит просто: потомки замещают родительских особей, которые предаются забвению. Переход на новую итерацию.

Говоря по-простому, m – это количество выживающих родительских особей, а l – количество потомков, создаваемых m родителями. Заметим, что l должно быть кратно m. На практике для обозначения ЭС обычно используются конкретные значения m и l. Например, при $m = 5$ и $l = 20$ имеем эволюционную стратегию (5, 20). Вот псевдокод алгоритма:

Algorithm 18 *The (μ, λ) Evolution Strategy*

```
1: μ ← number of parents selected
2: λ ← number of children generated by the parents

3: P ← {}
4: for λ times do                                          ▷ Build Initial Population
5:     P ← P ∪ {new random individual}
6: Best ← □
7: repeat
8:     for each individual P_i ∈ P do
9:         AssessFitness(P_i)
10:        if Best = □ or Fitness(P_i) > Fitness(Best) then
11:            Best ← P_i
12:    Q ← the μ individuals in P whose Fitness( ) are greatest        ▷ Truncation Selection
13:    P ← {}                          ▷ Join is done by just replacing P with the children
14:    for each individual Q_j ∈ Q do
15:        for λ/μ times do
16:            P ← P ∪ {Mutate(Copy(Q_j))}
17: until Best is the ideal solution or we have run out of time
18: return Best
```

Отметим, что вместо `Tweak` используется функция мутации `Mutation`, поскольку в популяционных методах существует множество способов реализации операции улучшения `Tweak` особи. Основных способа два: **мутация**, которая работает так же, как и встреченные ранее варианты `Tweak` – изменение одной особи путем применения (обычно небольших) случайных изменений; и **рекомбинация** или **кроссинговер**, в которых несколько (как правило, две) особи перемешиваются для создания потомков. **Далее будем использовать эти термины**, подразумевая операцию `Tweak`.

Алгоритм (m, l) имеет три «рукоятки управления», с помощью которых можно настраивать соотношение между исследованием пространства поиска и использованием найденных решений. На рис. 8 иллюстративно показаны эффекты от соответствующих операций:

- Значение l – очевидно, контролирует размер популяции и имеет такой же смысл, как и переменная n в алгоритме локального поиска с наискорейшим подъемом с замещением. В предельном случае, когда l стремится к бесконечности, алгоритм больше использует исследование пространства решений (случайный поиск).

- Значение m – управляет тем, насколько *избирательным* (*selective*) является алгоритм. При малых по сравнению с l значениях m алгоритм в большей степени использует уже найденные решения так как выживают только наилучшие особи.

- Степень, с которой производится мутация. Если `Mutate` привносит значительный шум, то потомки «падают далеко от яблони» и являются в достаточно большой степени случайными, вне зависимости от значения m, определяющего избирательность алгоритма.

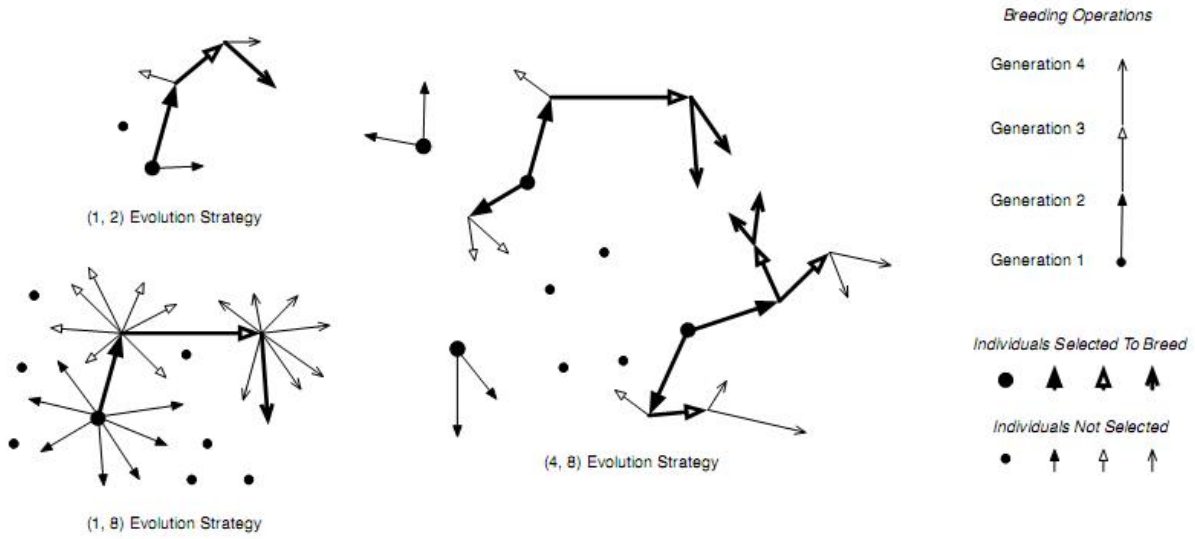

Рис. 8. Три варианта (m, l) ЭС. В каждом поколении выбирается m особей для скрещивания, каждая создает l/m потомков, что в сумме дает l особей-потомков

Еще один алгоритм ЭС называется **(m + l)** алгоритм. Отличается от первого только в одном: как работает операция `Join`. Напомним, что в (m, l) алгоритме родители просто замещались в следующем поколении потомками. А в алгоритме ($m + l$) в следующее поколение попадают и m родителей, и l потомков. Таким образом, родители конкурируют с потомками и во всех поколениях после начального размер популяции равен $m + l$. Алгоритм выглядит вот так:

Algorithm 19 *The $(\mu + \lambda)$ Evolution Strategy*

```
1: μ ← number of parents selected
2: λ ← number of children generated by the parents

3: P ← {}
4: for λ times do
5:     P ← P ∪ {new random individual}
6: Best ← □
7: repeat
8:     for each individual P_i ∈ P do
9:         AssessFitness(P_i)
10:        if Best = □ or Fitness(P_i) > Fitness(Best) then
11:            Best ← P_i
12:    Q ← the μ individuals in P whose Fitness( ) are greatest
13:    P ← Q                          ▷ The Join operation is the only difference with (μ, λ)
14:    for each individual Q_j ∈ Q do
15:        for λ/μ times do
16:            P ← P ∪ {Mutate(Copy(Q_j))}
17: until Best is the ideal solution or we have run out of time
18: return Best
```

В общем случае алгоритм $(m + l)$ в большей степени использует найденные решения, чем (m, l) алгоритм, т.к. высокоприспособленные родители остаются для конкуренции с потомками. Здесь есть своя потенциальная опасность: приспособленный родитель может превосходить всех остальных особей из поколения в поколение, что приведет к тому, что популяция **преждевременно сойдется** (***Premature Convergence***), и будет содержать только непосредственных потомков этой родительской особи. Это будет означать, что процесс поиска «застрял» в локальном оптимуме, соответствующем данному родителю.

Если приглядеться, то $(m + l)$ напоминает алгоритм локального поиска с наискорейшим подъемом тем, что в оба алгоритма разрешают конкурировать с потомками за выживание в следующем поколением. В то же время (m, l) похож на алгоритм локального поиска с наискорейшим подъемом с замещением, поскольку родители заменяются лучшими потомками. Это больше чем просто совпадения, алгоритмы локального поиска представляют существенно упрощенные версии алгоритмов ЭС. Вспомним, что при использовании определенного оператора `Tweak` «чистокровный» алгоритм локального поиска становится (1+1) алгоритмом, алгоритм локального поиска с наискорейшим подъемом с замещением становится $(1, l)$, а алгоритм локального поиска с наискорейшим подъемом – $(1+l)$. Вооружившись приведенным выше описанием алгоритмов, должно быть более понятно, как такое получается.

3.1.1. МУТАЦИЯ И ЭВОЛЮЦИОННОЕ ПРОГРАММИРОВАНИЕ

Исторически сложилось, что в эволюционных стратегиях используется представление данных в виде вещественного вектора фиксированной длины. Обычно такие вектора инициализируют с использованием алгоритма подобного Алгоритму 7. Мутация традиционно производится с помощью гауссовской свертки (Алгоритм 11).

Гауссовская свертка зависит в основном от значения дисперсии s^2, которое известно в ЭС как **вероятность мутации** (***mutation rate***) и определяет шумовую составляющую операции `Mutation`. Каким образом, подбирается значение s^2? Можно установить его заранее, или можно медленно его уменьшать, или можно адаптивно изменять s^2 в зависимости от текущих показателей системы. Если система начинает делать слишком большой упор на использование найденных решений, то можно повысить s^2, чтобы увеличить исследовательскую составляющую (либо понизить, чтобы еще больше увеличить использование решений). Само понятие изменения s^2 известно как **адаптивная вероятность мутации**. В общем случае подобные **адаптивные** операторы размножения подстраивают параметры в течение времени в зависимости от характеристик текущего процесса оптимизации[7].

Одно старое правило адаптивного изменения s^2 известно, как **правило одной пятой** (***One-Fifth Rule***) Инго Рехенберга[8] и выглядит следующим образом:

- Если более 1/5 потомков приспособленнее своих родителей, то алгоритм слишком сосредоточен на локальном оптимуме и следует увеличить s^2.

[7] Длительное время эволюционные стратегии ассоциировались с **самоадаптивными операторами** (***self-adaptive operators***), которые стохастически настраивались вместе с эволюцией особей. К примеру, особь может хранить информацию о «своих» процедурах мутации, которые и сами могут мутировать наряду с мутацией особи.

[8] Также описывается в его книге по эволюционным стратегиям! (см. сноску 6 на стр. 3)

- Если менее 1/5 потомков приспособленнее своих родителей, то алгоритм слишком активно исследует пространство поиска и следует уменьшить s^2.
- Если ровно 1/5 потомков приспособленнее своих родителей, то ничего не нужно менять.

Данное правило было выведено на основании результатов экспериментов с (1+1) ЭС на определенных простых задачах. Для более сложных ситуаций оно может и не быть оптимальным, однако является неплохой «точкой отсчета».

ЭС работают не только с векторами. На самом деле, практически идентичный подход немного ранее был разработан Ларри Фогелем (*Larry Fogel*) в Национальном научном фонде (г. Вашингтон, округ Колумбия) и позднее развит в Сан Диего[9]. Метод, названный **Эволюционным программированием** (**ЭП**) (***Evolutionary Programming***, ***EP***) отличается от ЭС по двум аспектам. Во-первых, он использует только $(m + l)$ стратегию причем $m = l$. Т.е. половина популяции уничтожается и заполняется потомками[10]. Во вторых ЭП было применено для практически любого способа кодирования. С самого начала Фогеля интересовала эволюция структуры графов (в частности, конечных автоматов, отсюда и «программирование»). Поэтому операция `Mutation` добавляла или удаляла ребро или вершину, изменяла метку ребра или вершины и т.д.

Такие операции необходимы, т.к. у них есть две особенности. Во-первых, чтобы гарантировать, что алгоритм оптимизации является глобальным, нужно гарантировать, пусть даже с малой вероятностью, что родитель может произвести *любого* потомка. Во-вторых, мы должны сохранить такую особенность, что *обычно* производятся *небольшие* изменения, чтобы не вызывать больших изменений в приспособленности, а *крупные* изменения особи случаются крайне редко. Частота небольших изменений может регулироваться, как, например, s^2.

3.2 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

Генетический алгоритм (ГА) (**Genetic Algorithm, GA**), который также известен как *генетические алгоритмы*, разработан Джоном Холландом (*John Holland*) в университете Мичигана в 1970-х годах[11]. Во многом этот алгоритм похож на (m, l) эволюционную стратегию: каждая итерация включают вычисление приспособленности, селекцию, размножение и формирование популяции следующего поколения. Основная разница заключается в том, как осуществляются селекции и размножение: в то время как в эволюционных стратегиях сначала выбираются родительские особи, и только *потом* формируются все потомки, в генетическом алгоритме селекция и генерация потомков производятся «малыми порциями», до тех пор, пока не будет сформирована популяция потомков.

Операция размножения начинает работу с пустой популяции потомков. Затем выбираются два родителя из исходной популяции, скрещиваются, и производится мутация результата. Получаются два потомка, которые добавляются к популяции потомков. Такой процесс

[9] Диссертация Ларри Фогеля была без сомнения первой диссертацией по эволюционным вычислениям, а вомозможно и самой первой большой работой. *Lawrence Fogel, 1964, On the Organization of Intellect, Ph.D. thesis, University of California, Los Angeles.*

[10] По большому счету это не одно и то же с $(m + l)$. – *Прим. перев.*

[11] Книга Холланда является одной из самых известных в своей области: *John Holland,1975, Adaptation in Natural and Artificial Systems, University of Michigan Press.*

осуществляется до тех пор, пока популяция потомков не будет заполнена полностью. Псевдокод алгоритма:

Algorithm 20 *The Genetic Algorithm (GA)*

```
popsize ← desired population size                    ▷ This is basically λ. Make it even.
P ← {}
for popsize times do
    P ← P ∪ {new random individual}
Best ← □
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i)
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
    Q ← {}                                           ▷ Here's where we begin to deviate from (μ, λ)
    for popsize/2 times do
        Parent P_a ← SelectWithReplacement(P)
        Parent P_b ← SelectWithReplacement(P)
        Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
        Q ← P ∪ {Mutate(C_a), Mutate(C_b)}
    P ← Q                                            ▷ End of deviation
until Best is the ideal solution or we have run out of time
return Best
```

Хотя этот алгоритм может быть использован для любых векторов (и соответственно различных способов кодирования), классически ГА применяют для *булевских* векторов фиксированной длины, точно также как и ЭС часто использовались для *вещественных* векторов. Давайте на момент рассмотрения генерации новых особей станем педантами. Если особь представлена вещественным вектором, то создание нового вектора может быть произведено, как и в ЭС (т.е. с помощью Алгоритма 7). Если же используется представление в виде булевского вектора, можно поступить так:

Algorithm 21 *Generate a Random Bit-Vector*

```
v⃗ ← a new vector ⟨v_1, v_2, ...v_l⟩
for i from 1 to l do
    if 0.5 > a random number chosen uniformly between 0.0 and 1.0 inclusive then
        v_i ← true
    else
        v_i ← false
return v⃗
```

3.2.1 КРОССИНГОВЕР И МУТАЦИЯ

Еще раз отметим, что ГА сильно похож на (m, l) алгоритм за исключением стадии размножения, для которой нам потребуются две новые функции: `SelectWithReplacement` и `Crossover`; и, конечно, `Mutation`. Начнем с последней. Мутацию вещественного вектора можно произвести с помощью гауссовской свертки (Алгоритм 11). Как же применить `Mutate` к булевскому вектору? Простым способом является **битовая мутация** (**bit-flip mutation**): проходим по всему вектору и с некоторой

вероятностью «подбрасываем монетку» (часто вероятность берут равной $1/L$, где L – длина вектора). Если выпал «орел», то инвертируем соответствующий бит:

Algorithm 22 *Bit-Flip Mutation*

1: $p \leftarrow$ probability of flipping a bit ▷ Often p is set to $1/l$
2: $\vec{v} \leftarrow$ boolean vector $\langle v_1, v_2, \ldots v_l \rangle$ to be mutated

3: **for** i from 1 to l **do**
4: **if** $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
5: $v_i \leftarrow \neg(v_i)$
6: **return** $\vec{v}$

Кроссинговер (**crossover**) является уникальным оператором для ГА[12]. Он используется для перемешивания частей родительских особей для создания потомков. Как производится такое перемешивание зависит от используемого способа кодирования. Существует три классических кроссинговера для векторов: **1-точечный** (***One-Point***), **2-точечный** (***Two-Point***) и **однородный** (***Uniform***).

Предположим, имеется вектор длины L. Для однородного кроссинговера выбирается число c от 1 до L и элементы с индексами, большими c, меняются местами, как показано на рис. 9.

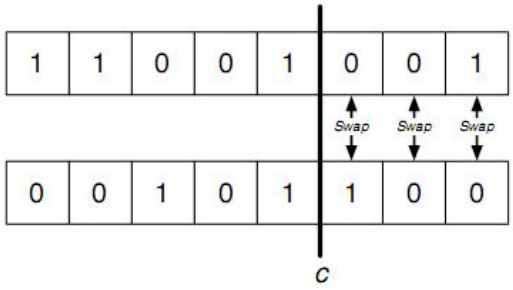

Рис. 9. Одноточечный кроссинговер

Соответствующий алгоритм:

Algorithm 23 *One-Point Crossover*

1: $\vec{v} \leftarrow$ first vector $\langle v_1, v_2, \ldots v_L \rangle$ to be crossed over
2: $\vec{w} \leftarrow$ second vector $\langle w_1, w_2, \ldots w_L \rangle$ to be crossed over

3: $c \leftarrow$ random integer chosen uniformly from 1 to L inclusive
4: **for** i from c to L **do**
5: Swap the values of v_i and w_i
6: **return** $\vec{v}$ and $\vec{w}$

Отметим, что при $c = 1$ или $c = L$, то кроссинговер фактически не срабатывает: если $c = 1$, то *все* элементы меняются местами, а если $c = L$, то обмен элементов *не производится*. В любом случае особи не меняются. Проблема использования одноточечного кроссинговера заключается в возможной **связанности** (***linkage***) элементов вектора. Прежде всего, заметим, что элементы v_1 и v_L будут с высокой вероятностью разорваны кроссинговером, т.к. к этому приводит почти любое значение c. Точно также, вероятность разрыва элементов v_1 и v_2 достаточно мала, т.к. это событие произойдет только при $c = 1$. Если вектор построен таким

[12] Хотя он уже длительное время используется и в эволюционных стратегиях.

образом, что элементы v_1 и v_L должны «работать» вместе для обеспечения высокой приспособленности, то в процессе работы алгоритма хорошие пары значений этих элементов будут постоянно разрушаться. Проблема связанности может быть частично решена с применением двухточечного кроссинговера: выбираем *два* числа c и d и меняем местами элементы, индексы которых попадают между этими числами. На рис. 10 показана общая идея, и ниже дан псевдокод:

Algorithm 24 *Two-Point Crossover*

1: $\vec{v} \leftarrow$ first vector $\langle v_1, v_2, \ldots v_l \rangle$ to be crossed over
2: $\vec{w} \leftarrow$ second vector $\langle w_1, w_2, \ldots w_l \rangle$ to be crossed over

3: $c \leftarrow$ random integer chosen uniformly from 1 to l inclusive
4: $d \leftarrow$ random integer chosen uniformly from 1 to l inclusive
5: **if** $c > d$ **then**
6: Swap c and d
7: **for** i from c to $d - 1$ **do**
8: Swap the values of v_i and w_i
9: **return** $\vec{v}$ and $\vec{w}$

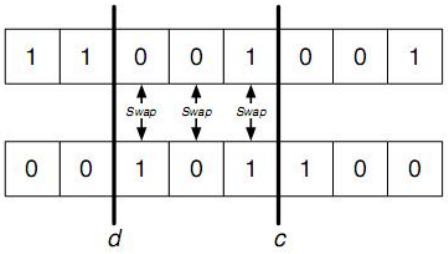

Рис. 10. Двухточечный кроссинговер

На первый взгляд польза от такого подхода неочевидна. Но представьте, что векторы имеют вид *колец* (когда v_L находится по соседству с v_1). Двухточечный кроссинговер разрывает кольца в двух местах и производит обмен сегментами. Поскольку v_1 и v_L расположены рядом, то единственный способ разделить их, заключается в том, что c или d попадут между ними. Т.е. аналогично v_1 и v_2[13].

Однако и сейчас проблема связанности не исчезает. Теперь v_1 и v_L рассматриваются единообразно, но как быть с v_1 и $v_{L/2}$? Появление точки разрыва на больших участках все еще более вероятно, чем на малых, вроде v_1 и v_2 (или v_1 и v_L). Можно рассматривать все гены с одинаковой позиции с учетом связанности, чтобы точки разрыва располагались независимо друг от друга, с использованием однородного кроссинговера. Просто проходим по векторам и меняем местами элементы, если монетка упадет «орлом» с некоторой вероятностью p[14].

[13] Можно обобщить двухточечный кроссинговер в виде **многоточечного** (***Multi-Point***): выбираем n случайных точек разрыва и сортируем их в порядке возрастания. Получим c_1, c_2, …, c_n. Теперь меняем местами участки родительских хромосом между индексами , c_1 и c_2, c_3 и c_4, c_5 и c_6 и т.д.

[14] Оригинальный однородный кроссинговер использовал p = 1/2, и был впервые предложен в статье David Ackley,1987, A Connectionist Machine for Genetic Hillclimbing, Kluwer Academic Publishers. В более общем случае, когда p произвольно, говорят о *параметризованном* однородном кроссинговере.

Algorithm 25 *Uniform Crossover*

1: $p \leftarrow$ probability of swapping an index ▷ Often p is set to $1/l$. At any rate, $p \leq 0.5$
2: $\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
3: $\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

4: **for** i from 1 to l **do**
5: **if** $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
6: Swap the values of v_i and w_i
7: **return** $\vec{v}$ and $\vec{w}$

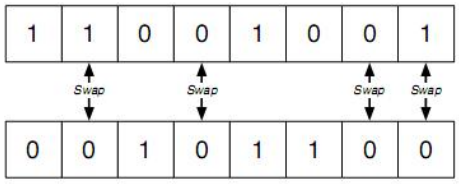

Рис. 11. Однородный кроссинговер

Кроссинговер не является глобальной мутацией. Если скрещивать два вектора, то получить любые возможные векторы при этом не получится. Представим векторы точками в пространстве. Теперь построим на этих точках гиперкуб, так чтобы противоположные углы совпали с точками. Например, на рис. 12 показан иллюстративный случай для 3-мерных векторов. Поэтому результаты работы всех операций кроссинговера будут ограничены: новые векторы будут располагаться в одном из углов гиперкуба.

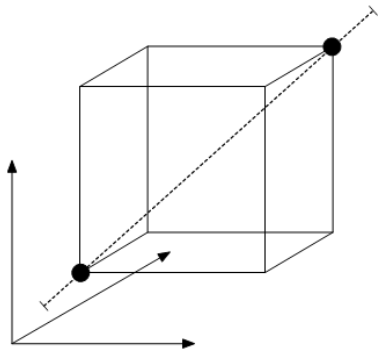

Рис. 12. Куб в пространстве, сформированном двумя 3-мерными векторами (черные кружки). Пунктирная линия соединяет исходные векторы

Более того, если постоянно применять к популяции кроссинговер и селекцию, то наступит ситуация, когда некоторые аллели (значения в определенных позициях вектора) практически полностью исчезнут. Рано или поздно популяция **выродится** (***converge***), или, что часто случается, (к сожалению) наступит **преждевременное вырождение** (***premature convergence***), когда популяция будет состоять из копий одной особи. В данной ситуации выхода нет: когда особь скрещивается сама с собой, не производится ничего нового[15]. Поэтому чтобы реализовать с помощью генетического алгоритма глобальный поиск необходимо использовать операцию `Mutation`.

Зачем же тогда нужен кроссинговер? Изначально кроссинговер был разработан в предположении, что высокоприспособленные особи часто обладают одинаковыми

[15] Операторы кроссинговера, которые не производят новой информации при скрещивании особи с собой, называются **гомологичными** (***homologous***).

свойствами, которые называют **строительными блоками** (***building blocks***). Для особей, представляющих векторы фиксированной длины, строительный блок часто определяют как набор генов, с определенными аллелями. К примеру, для булевской особи 10110101, строительным блоком вполне мог бы быть ***101*1 (где позиции, обозначенные символом «*», не входят в строительный блок). Во многих задачах, в которых использование кроссинговера оказалось полезным, приспособленность данной особи часто слабо коррелировала с наличием строительных блоков, а кроссинговер способствует распространению строительных блоков по особям популяции. Строительные блоки были основной темой многих ранних аналитических исследований генетических алгоритмов, формализованных в области, называемой **теория шаблонов** (***schema theory***).

В любом случае, такова идея. Использование кроссинговера также предполагает относительно низкую связанность между генами особи. Другими словами вклад каждого свойства в приспособленность *мало*/*не зависит* от вклада других свойств. До сих пор мы решали задачи, полагая, что связанность между генами распределена неравномерно. Но мы не можем игнорировать эвристического предположения о связанности, используемого для кроссинговера. Соответствует ли это предположение вашей задаче? Во многих случаях так и есть, но уверенным до конца быть нельзя.

Теоретически можно применять однородный кроссинговер с несколькими векторами сразу и производить потомков, как комбинацию всех этих векторов[16]. Чтобы избежать полной случайности результата, можно ограничить перемешивание, поэтому вероятность обмена элементов в каждой позиции вектора не должна быть слишком высокой. Хотя на практике что-то подобное используется очень редко. Чтобы реализовать этот прием нам для начала требуется определить, как однородно и случайно перемешать элементы вектора. К удивлению это не так уж и очевидно, как может показаться:

Algorithm 26 *Randomly Shuffle a Vector*

1: $\vec{p} \leftarrow$ elements to shuffle $\langle p_1, ..., p_l \rangle$

2: **for** i from l down to 2 **do** ▷ Note we don't do 1
3: $j \leftarrow$ integer chosen at random from 1 to i inclusive
4: Swap p_i and p_j

Вооружившись таким случайным перемешивателем (который, кстати, будет использован и в других алгоритмах в будущем), можем скрестить сразу K векторов, путем обмена элементами и одновременным созданием K потомков.

Algorithm 27 *Uniform Crossover among K Vectors*

1: $p \leftarrow$ probability of swapping an index ▷ Ought to be very small
2: $W \leftarrow \{W_1, ..., W_k\}$ vectors to cross over, each of length l

3: $\vec{v} \leftarrow$ vector $\langle v_1, ..., v_k \rangle$
4: **for** i from 1 to l **do**
5: **if** $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
6: **for** j from 1 to k **do** ▷ Load $\vec{v}$ with the ith elements from each vector in W
7: $\vec{w} \leftarrow W_j$
8: $v_j \leftarrow w_i$
9: Randomly Shuffle $\vec{v}$

[16] Ничего нового в этом нет: подобный подход использовал Ханс-Поль Швефель в ранних ЭС.

```
10:     for j from 1 to k do                          ▷ Put back the elements, all mixed up
11:         w_i ← v_j
12:         W_j ← w⃗
13: return W
```

3.2.2 СНОВА О РЕКОМБИНАЦИИ

Пока что мы рассматривали только такие операторы кроссинговера, которые производят обмен элементами. Но если использовать вещественные векторы, то рекомбинация должна давать в результате что-то более «размытое», например, среднее значение двух элементов, вместо простого обмена местами. Нарисуем линию между двумя точками и выберем на этой линии две новые точки между исходными. Можно также продолжить линию за точки, как показано пунктиром на рис. 12. Получим алгоритм, называемый **линейной рекомбинацией** (***Line Recombination***), разработанный Хайнцем Мюленбайном (*Heinz Muhlenbein*) и Дирком Шлиркампом-Вусеном (*Dirk Schlierkamp-Voosen*). Алгоритм зависит от переменной *p*, которая определяет, как далеко друг от друга будут располагаться на прямой потомки. При p = 0 потомки попадут на линию внутри гиперкуба (т.е. между исходных точек). Если же $p > 0$, то потомки могут оказаться в любой точке на линии снаружи гиперкуба.

Algorithm 28 *Line Recombination*

```
1: p ← positive value which determines how far long the line a child can be located   ▷ Try 0.25
2: v⃗ ← first vector ⟨v_1, v_2, ...v_l⟩ to be crossed over
3: w⃗ ← second vector ⟨w_1, w_2, ...w_l⟩ to be crossed over

4: α ← random value from −p to 1 + p inclusive
5: β ← random value from −p to 1 + p inclusive
6: for i from 1 to l do
7:     t ← αv_i + (1 − α)w_i
8:     s ← βw_i + (1 − β)v_i
9:     if t and s are within bounds then
10:        v_i ← t
11:        w_i ← s
12: return v⃗ and w⃗
```

Можно расширить алгоритм, добавив дополнительные переменные *a* и *b* для каждой позиции вектора. В результате потомки окажутся либо внутри гиперкуба, либо слегка «выпадут» наружу (при $p > 0$). Мюленбайн и Шлиркампом-Вусен назвали этот алгоритм *промежуточной рекомбинацией* (***Intermediate Recombination***)[17].

[17] Ну ладно, они назвали эти алгоритмы *Расширенная* (*Extended*) линейная и *Расширенная* промежуточная рекомбинация в статье Heinz Muhlenbein and Dirk Schlierkamp-Voosen, 1993, Predictive models for the breeder genetic algorithm: I. continuous parameter optimization, Evolutionary Computation, 1(1). Эти методы длительное время использовались в эволюционных вычислениях, но под разными именами: примечательно, что Ханс-Поль Швефель использовал (помимо всего прочего) в оригинальных работах по эволюционным стратегиям линейную рекомбинацию с $p = -0{,}5$, но он, как и другие, называл ее *промежуточной рекомбинацией*. Швефель также применял и вариацию: для каждого гена потомка выбирались два случайных родителя и усреднялись значения их генов.

Algorithm 29 *Intermediate Recombination*

$p \leftarrow$ positive value which determines how far long the line a child can be located ▷ Try 0.25
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, \ldots v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, \ldots w_l \rangle$ to be crossed over

for i from 1 to l **do**
    repeat
        $\alpha \leftarrow$ random value from $-p$ to $1 + p$ inclusive ▷ We just moved these two lines!
        $\beta \leftarrow$ random value from $-p$ to $1 + p$ inclusive
        $t \leftarrow \alpha v_i + (1 - \alpha) w_i$
        $s \leftarrow \beta w_i + (1 - \beta) v_i$
    until t and s are within bounds
    $v_i \leftarrow t$
    $w_i \leftarrow s$
return $\vec{v}$ and $\vec{w}$

Поскольку для каждого элемента используются различные a и b, то если значения элементов выходят за границы, то вместо отбрасывания этих элементов, можно просто выбрать другие a и b.

Зачем могут понадобиться значения $p > 0$? Предположим, что операция `Mutate` отсутствует и используем либо линейную, либо промежуточную рекомбинацию. Каждый раз, когда выбираются родители, создаются потомки, которые оказываются внутри гиперкуба. Произвести потомка *вне ограничивающего куба для всей популяции* невозможно. Поэтому если необходимо исследовать неизвестные области в пространстве поиска, нужен способ создания потомков, отличных от родителей.

Другие представления. На данный момент мы использовали только векторы. Далее будут рассмотрены и другие способы представления. Пока что нужно запомнить, что если хорошо уяснить сущность мутации и операции Mutate, то можно брать любое представление. Как нам быть со структурами графов? А с наборами? Списками произвольной длины? Деревьями?

3.2.3 СЕЛЕКЦИЯ

Наконец, настала очередь обсудить функцию `SelectWithReplacement`. В эволюционных стратегиях селекция работала просто: самые плохие особи отбраковывались. Но поскольку в генетическом алгоритме селекция, кроссинговер и мутация производятся итеративно, то появляется больше различных вариантов.

Оригинальный метод селекции в генетическом алгоритме называется **пропорциональная селекция** (***Fitness-Proportionate Selection***), также известная как **рулеточная селекция** (***Roulette Selection***). В этом методе особи выбираются пропорционально значению приспособленности: если приспособленность особи выше, то она будет чаще выбрана для скрещивания[18].

Для этого устанавливаем «размер» особей в соответствии с их приспособленностью, как показано на рис. 13[19]. Пусть $s = \sum_i f_i$ – сумма приспособленностей всех особей. Случайное число в диапазоне от 0 до s определяет, какая особь будет выбрана.

[18] Предполагаем, что значение приспособленности ≥ 0. И как обычно: чем больше, тем лучше.
[19] Также следуя Джону Холланду. См. сноску 11 на стр. 7.

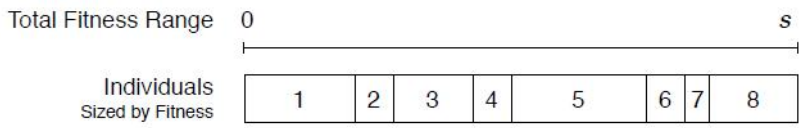

Рис. 13. Массив диапазонов, занимаемых особями, в пропорциональной селекции

Algorithm 30 *Fitness-Proportionate Selection*

```
1: perform once per generation
2:     global p⃗ ← population copied into a vector of individuals ⟨p_1, p_2, ..., p_l⟩
3:     global f⃗ ← ⟨f_1, f_2, ..., f_l⟩ fitnesses of individuals in p⃗ in the same order as p⃗
4:     if f⃗ is all 0.0s then                          ▷ Deal with all 0 fitnesses gracefully
5:         Convert f⃗ to all 1.0s
6:     for i from 2 to l do    ▷ Convert f⃗ to a CDF. This will also cause f_l = s, the sum of fitnesses.
7:         f_i ← f_i + f_{i−1}
8: perform each time
9:     n ← random number from 0 to f_l inclusive
10:    for i from 2 to l do                ▷ This could be done more efficiently with binary search
11:        if f_{i−1} < n ≤ f_i then
12:            return p_i
13:    return p_1
```

Отметим, что в пропорциональной селекции есть этап предобработки: преобразование всех приспособленностей (или, в действительности, их копий) в кумулятивное распределение. Это необходимо сделать один раз в течение поколения. Кроме этого, приведенный код производит линейный поиск в массиве приспособленностей, хотя более разумным было бы использовать бинарный поиск, вычислительная сложность которого составляет $O(\lg n)$.

Вариантом пропорционального отбора является **стохастический универсальный отбор** (***Stochastic Universal Sampling, SUS***), разработанный Джеймсом Бейкером (*James Baker*). В SUS селекция производится пропорционально приспособленности, но таким образом, что «хорошие» особи будут выбраны как минимум один раз. Данный алгоритм обладает свойством выборки с низкой вариабельностью (*low variance sampling*) и я привожу его здесь, потому что в настоящее время он достаточно популярен и в других областях, помимо эволюционных вычислений (самые известные из них: **фильтр частиц** (***Particle Filter***) для **марковских процессов принятия решений** (***Markov Decision Processes***))[20].

В методе SUS выбираются сразу n особей (обычно n равно размеру популяции в следующем поколении, и в рассматриваемом случае $n = l$). Сначала, как и раньше создаем массив приспособленностей. Затем случайным образом выбираем позицию (число) от 0 до s/n, и выбираем особь, соответствующую полученной позиции. Затем сдвигаем позицию на s/n и повторяем выбор особи (n раз). С каждым сдвигом выбираем ту особь в диапазон приспособленности которой мы попали. Данный процесс показан на рис. 14. Алгоритм:

[20] В этих областях на данный метод не ссылаются. Он был представлен в следующей статье James Edward Baker, 1987, Reducing bias and inefficiency in the selection algorithm, in John Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms (ICGA)*, pages 14–21, Lawrence Erlbaum Associates, Hillsdale.

Algorithm 31 *Stochastic Universal Sampling*

perform once per n individuals produced ▷ Usually $n = l$, that is, once per generation
  global $\vec{p} \leftarrow$ copy of vector of individuals (our population) $\langle p_1, p_2, ..., p_l \rangle$, shuffled randomly
    ▷ To shuffle a vector randomly, see Algorithm 26
  global $\vec{f} \leftarrow \langle f_1, f_2, ..., f_l \rangle$ fitnesses of individuals in $\vec{p}$ in the same order as $\vec{p}$
  global $index \leftarrow 0$
  if $\vec{f}$ is all 0.0s **then**
    Convert $\vec{f}$ to all 1.0s
  for i from 2 to l **do** ▷ Convert $\vec{f}$ to a CDF. This will also cause $f_l = s$, the sum of fitnesses.
    $f_i \leftarrow f_i + f_{i-1}$
  global $value \leftarrow$ random number from 0 to f_l / n inclusive
perform each time
  while $f_{index} < value$ **do**
    $index \leftarrow index + 1$
  $value \leftarrow value + f_l / n$
  return p_{index}

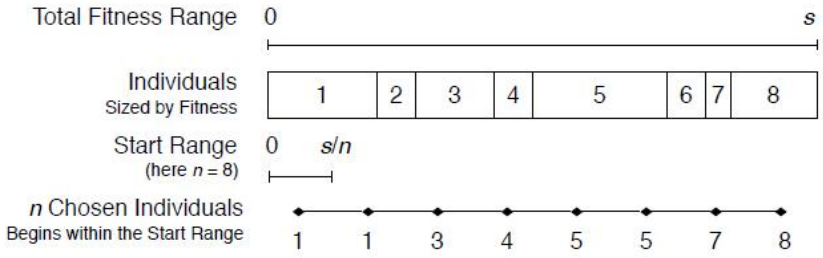

Рис. 14. Массив диапазонов особей, стартовый диапазон и выбранные точки в стохастическом универсальном отборе

При использовании SUS можно выделить два преимущества. Во-первых, выбор n особей производится за $O(n)$ операций, вместо $O(n \lg n)$ для пропорциональной селекции. Раньше это было важно, но теперь потеряло актуальность, поскольку львиная доля времени в большинстве алгоритмах оптимизации тратится на вычисление приспособленности, а не на процессы селекции и размножения. Второе и более интересное преимущество SUS заключается в том, что этот метод гарантирует выбор хорошо приспособленной особи (занимает в массиве диапазон больше s/n), причем возможен и многократный выбор. В пропорциональной селекции даже самые приспособленные особи могут быть не выбраны.

Для описанных методов есть и серьезная общая проблема: в них предполагается, что значение приспособленности означает нечто действительно важное. Однако часто функция приспособленности выбирается таким образом, что большие значения просто лучше, чем меньшие, не подразумевая под этим ничего конкретного. Даже если функция приспособленности выбирается тщательно, можно рассмотреть следующую ситуацию. Пусть значения функции приспособленности ограничены от 0 до 10. Ближе к концу запуска все особи будут иметь приспособленности вроде 9,97, 9,98, 9,99 и т.д. Нам необходимо найти максимум приспособленности, поэтому мы *хотим* выбрать особь с приспособленностью 9,99. Однако для пропорциональной селекции (и для SUS) все особи будут выбраны с практически одинаковой вероятностью. Другими словами система просто сошлась к случайной селекции.

Чтобы исправить подобную ситуацию можно **масштабировать** (***scale***) значение функции приспособленности, чтобы сделать ее более чувствительной к значениям, близким к оптимальным. Однако чтобы по-настоящему устранить этот недостаток необходимо

использовать **непараметрический** (***non-parametric***) алгоритм селекции, который игнорирует значение приспособленности и учитывает только отношения порядка (больше, меньше), т.е. просто упорядочивает (ранжирует) особей по приспособленности. Это можно сделать и селекцией усечением, но наиболее популярный метод в настоящее время – это **Турнирная селекция** (***Tournament Selection***)[21], которая реализуется чрезвычайно простым алгоритмом:

Algorithm 32 *Tournament Selection*

```
P ← population
t ← tournament size, t ≥ 1

Best ← individual picked at random from P with replacement
for i from 2 to t do
    Next ← individual picked at random from P with replacement
    if Fitness(Next) > Fitness(Best) then
        Best ← Next
return Best
```

В результате функция возвращает самую приспособленную особь из t случайных из данной популяции. Вот и все! Турнирная селекция стала **основным методом селекции в генетическом алгоритме** и многих других по различным причинам. Во-первых, она нечувствительна к особенностям вычисления значений приспособленности. Во-вторых, она чертовски проста, не требует предобработки и может быть распараллелена безо всяких заморочек. В-третьих, она настраиваемая: **размер турнира** (***tournament size***) t определяет давление селекции. В крайнем случае, когда $t = 1$ имеем случайный отбор. При очень больших t (намного больше, чем даже размер популяции) вероятность победы в турнире наиболее приспособленной особи приближается к 1, поэтому турнирная селекция будет каждый раз просто выбирать одну и ту же самую приспособленную особь (другими словами, уподобится селекции усечением при $m = 1$).

В генетическом алгоритме наибольшей популярностью пользуется $t = 2$. Для некоторых способов кодирования информации (таких как в генетическом программировании, см. Раздел 4.3) часто можно встретить более избирательное $t = 7$. Чтобы реализовать давление селекции ниже, чем при $t = 2$, и при этом не допустить случайности выбора, потребуется что-то вроде небольшого фокуса. Один способов, который я бы использовал, – это возможность задавать вещественные значения t в диапазоне от 1 до 2. В этом случае с вероятностью $t - 1$ проводим турнирную селекцию размера $t = 2$, иначе выбираем из популяции произвольную особь.

3.3 ВАРИАЦИИ НА ТЕМУ ИСПОЛЬЗОВАНИЯ НАЙДЕННЫХ РЕШЕНИЙ

Общая тенденция в новых алгоритмах – увеличение роли найденных решений[22]. Вот некоторые варианты: **Элитаризм** (***Elitism***), **Устойчивый генетический алгоритм** (***The Steady-State Genetic Algorithm***) (и методы с **Разрывом поколений** (***Generation Gap***)), генетический алгоритм со **схемой генетического программирования с кодированием деревьями** (***Tree-Style Genetic Programming Pipeline***), **Гибридные эволюционно-**

[21] Турнирная селекция вполне может быть и «народным» алгоритмом, однако самое первое упоминание встречается в работе Anne Brindle, 1981, Genetic Algorithms for Function Optimization, Ph.D. thesis, University of Alberta. Она использовала бинарную турнирную селекцию ($t = 2$).

[22] В оригинале используется устоявшееся «exploitative», что переводится на русский язык как «эксплуатационный». Аналогично, далее слово «explorative» будет переведено как «исследовательский». Правда, оба варианта странно звучат и весьма посредственно отражают суть явления. Однако для краткости далее будет использоваться именно такой перевод. Все несогласные, пожалуйте в почту, только просьба быть конструктивными. – *Прим. перев.*

локальные алгоритмы (***Hybrid Evolutionary and Hill-Climbing Algorithms***) и родственный метод, называемый **Распределенный поиск с переопределением путей** (***Scatter Search with Path Relinking***). Первые три являются эксплуатационными, поскольку приспособленные родители остаются в популяции и могут конкурировать с потомками, как в алгоритме $m + l$. Два последних просто «обновляют» достижения эволюции с помощью локальных методов поиска.

3.3.1 ЭЛИТАРИЗМ

Элитаризм работает просто: генетический алгоритм модифицируется таким образом, что наиболее приспособленная особь или особи предыдущего поколения сразу попадают в следующее поколение[23]. Эти особи называются **элитными** (***elites***). Этот алгоритм напоминает ($m + l$) тем, что сохраняет особь (особей) с наилучшей приспособленностью для следующего поколения. Таким образом, повышается использование найденных решений, требующее дополнительных настроек (возможно, повышением шума от кроссинговера и мутации, либо ослаблением давления селекции).

Небольшой подвох заключается в следующем. Если необходимо установить размер популяции равным `popsize`, и при этом используется кроссинговер, то на самом деле нужно, чтобы `popsize` минус количество элитных особей было кратно двум, как в следующем алгоритме[24]:

Algorithm 33 *The Genetic Algorithm with Elitism*

```
1: popsize ← desired population size
2: n ← desired number of elite individuals                ▷ popsize − n should be even
3: P ← {}
4: for popsize times do
5:     P ← P ∪ {new random individual}
6: Best ← □
7: repeat
8:     for each individual P_i ∈ P do
9:         AssessFitness(P_i)
10:        if Best = □ or Fitness(P_i) > Fitness(Best) then
11:            Best ← P_i
12:    Q ← {the n fittest individuals in P, breaking ties at random}
13:    for (popsize − n)/2 times do
14:        Parent P_a ← SelectWithReplacement(P)
15:        Parent P_b ← SelectWithReplacement(P)
16:        Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
17:        Q ← Q ∪ {Mutate(C_a), Mutate(C_b)}
18:    P ← Q
19: until Best is the ideal solution or we have run out of time
20: return Best
```

3.3.2 УСТОЙЧИВЫЙ ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

Альтернативой традиционному поколенческому подходу в генетическом алгоритме является использование **устойчивого** (***steady-state***) подхода, в котором популяция обновляется

[23] Элитаризм был разработан Кеном Де Йонгом в его диссертации (см. сноску 20 на стр. 26)

[24] На самом деле это не обязательное требование, т.к. достаточно просто удалять лишние особи-потомки. – *Прим. перев.*

частями, а не вся сразу. Этот подход был популяризован Дарреллом Уитли (Darrell Whitley) и Джоан Кауф (Joan Kauth) с помощью системы GENITOR. Общая идея заключается в итеративном создании одного или двух потомков и добавление их непосредственно в популяцию, предварительно убив некоторых уже существующих особей, чтобы высвободить необходимые места.

Устойчивый генетический алгоритм имеет два важных свойства. Во-первых, в нем используется в два раза меньше памяти, чем в генетическом алгоритме, поскольку всегда рассматривается только одна популяция (нет Q, есть только P). Второе, он существенно более эксплуатационный, по сравнению с традиционным подходом: родители остаются в популяции на, потенциально, очень длительное время и поэтому, подобно $m+l$ и элитаризму, это вызывает риск преждевременного вырождения популяции, когда в ней большинство особей является копиями нескольких приспособленных. Этот фактор может быть усилен тем, как мы решаем проводить функцию `SelectForDeath`. Если мы выбираем для удаления *неприспособленных* (*unfit*) особей (например, использую турнирную селекцию, выбирающую *самую неприспособленную* особь в турнире), то это может способствовать уменьшению разнообразия в популяции. Более распространенным является случайный выбор уничтожаемых особей. Таким образом, приспособленные особи, виновные в преждевременной сходимости, могут быть случайно убраны из популяции[25]. Если необходимо уменьшить эксплуатационные свойства, можно прибегнуть к стандартным приемам: использовать достаточно малоселективный оператор для SelectWithReplacement, и добавить шум в кроссинговер и мутацию. Ниже приведена версия, использующая кроссинговер и поэтому создающая двух потомков:

Algorithm 34 *The Steady-State Genetic Algorithm*

```
1:  popsize ← desired population size
2:  P ← {}
3:  for popsize times do
4:      P ← P ∪ {new random individual}
5:  Best ← □
6:  for each individual P_i ∈ P do
7:      AssessFitness(P_i)
8:      if Best = □ or Fitness(P_i) > Fitness(Best) then
9:          Best ← P_i
10: repeat
11:     Parent P_a ← SelectWithReplacement(P)        ▷ We first breed two children C_a and C_b
12:     Parent P_b ← SelectWithReplacement(P)
13:     Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
14:     C_a ← Mutate(C_a)
15:     C_b ← Mutate(C_b)
16:     AssessFitness(C_a)                           ▷ We next assess the fitness of C_a and C_b
17:     if Fitness(C_a) > Fitness(Best) then
18:         Best ← C_a
19:     AssessFitness(C_b)
20:     if Fitness(C_b) > Fitness(Best) then
21:         Best ← C_b
22:     Individual P_d ← SelectForDeath(P)
23:     Individual P_e ← SelectForDeath(P)           ▷ P_d must be ≠ P_e
```

[25] Вот любопытный вопрос: если в нашем распоряжении есть достаточно памяти, зачем тогда вообще удалять особей?

24: $P \leftarrow P - \{P_d, P_e\}$ ▷ We then delete P_d and P_e from the population
25: $P \leftarrow P \cup \{C_a, C_b\}$ ▷ Finally we add C_a and C_b to the population
26: **until** *Best* is the ideal solution or we have run out of time
27: **return** *Best*

Естественно, данный алгоритм можно обобщить таким образом, чтобы замещать не 2 особей, а n. Методы, использующие большие значения n (50% от размера популяции и более), часто называют методами с разрывом поколений[26]. Впервые их использовал Кен Де Йонг. По мере приближения n к 100% алгоритм становится все больше похож на обычный поколенческий.

3.3.3 АЛГОРИТМ СО СХЕМОЙ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ С КОДИРОВАНИЕМ ДЕРЕВЬЯМИ

Генетическое программирование применяется для поиска высоко приспособленных *компьютерных программ* с использованием метаэвристик. Наиболее распространенный вид генетического программирования – **генетическое программирование с деревьями**, использующее деревья для кодирования информации. Традиционным в этом виде алгоритма, хотя и не обязательным, является использование варианта генетического алгоритма со специальным методом размножения, разработанного Джоном Козой[27]. Вместо выполнения кроссинговера и мутации в этом алгоритме сначала «подкидывается монетка», по которой с 90% вероятностью выбираются два родителя, и производится кроссинговер. В противном случае выбирается только один родитель и копируется в новую популяцию, что делает этот вариант алгоритма сильно эксплуатационным.

Сделаем несколько замечаний. Во-первых, мутации нет: это не глобальный алгоритм поиска. Однако некоторые версии кроссинговера в генетическом программировании с деревьями вносят столько разнообразия, что потребность в мутации практически отпадает. Во-вторых, в этом алгоритме может создаваться на одного потомка больше, чем нужно. В этом случае его просто нужно отбросить. В-третьих, процедура селекции традиционно обладает очень большим давлением: в генетическом программировании обычно применяется турнирная селекция с размером турнира $t = 7$. Итак:

Algorithm 35 *The Genetic Algorithm (Tree-Style Genetic Programming Pipeline)*
1: *popsize* ← desired population size
2: r ← probability of performing direct reproduction ▷ Usually $r = 0.1$
3: $P \leftarrow \{\}$
4: **for** *popsize* times **do**
5: $P \leftarrow P \cup \{\text{new random individual}\}$
6: *Best* ← □
7: **repeat**
8: **for** each individual $P_i \in P$ **do**

[26] У этих методов весьма богатая история. Ранние версии ЭС использовали ныне не встречаемую $(m+1)$ эволюционную стратегию, в которой m родительских особей (из текущей популяции) совместно создавали 1 потомка (см. сноску 6 на стр.3). Кен Де Йонг сделал ранние исследования методов с разрывом поколений в диссертации Kenneth De Jong, 1975, An Analysis of the Behaviour of a Class of Genetic Adaptive Systems, Ph.D. thesis, University of Michigan. Позже GENITOR популяризовал понятие устойчивых алгоритмов. Darrell Whitley and Joan Kauth, 1988, GENITOR: A different genetic algorithm, Technical Report CS-88-101, Colorado State University

[27] John R. Koza, 1992, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press.

```
        AssessFitness(P_i)
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
    Q ← {}
    repeat                          ▷ Here's where we begin to deviate from The Genetic Algorithm
        if r ≥ a random number chosen uniformly from 0.0 to 1.0 inclusive then
            Parent P_i ← SelectWithReplacement(P)
            Q ← Q ∪ {Copy(P_i)}
        else
            Parent P_a ← SelectWithReplacement(P)
            Parent P_b ← SelectWithReplacement(P)
            Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
            Q ← Q ∪ {C_a}
            if ||Q|| < popsize then
                Q ← Q ∪ {C_b}
    until ||Q|| = popsize                                          ▷ End Deviation
    P ← Q
until Best is the ideal solution or we have run out of time
return Best
```

3.3.4 ГИБРИДНЫЕ ЭВОЛЮЦИОННЫЕ И ЛОКАЛЬНЫЕ АЛГОРИТМЫ

Существует великое *множество* способов для создания гибридных версий различных метаэвристических алгоритмов, но, возможно, наиболее популярный подход – это гибрид эволюционных вычислений и локального оптимизатора, такого как метод локального поиска. Обычно ЭА расширяется локальным методом путем переопределения этапа вычисления приспособленности, при этом содержимое особи оптимизируется, и обновленный вариант замещает исходный в текущей популяции. Таким образом, можно модифицировать любой ЭА, ниже приведен фрагмент ЭА из Алгоритма 17, преобразованный в гибридный алгоритм:

Algorithm 36 *An Abstract Hybrid Evolutionary- and Hill-Climbing Algorithm*

```
t ← number of iterations to Hill-Climb

P ← Build Initial Population
Best ← □
repeat
    AssessFitness(P)
    for each individual P_i ∈ P do
        P_i ← Hill-Climb(P_i) for t iterations                     ▷ Replace P_i in P
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
    P ← Join(P, Breed(P))
until Best is the ideal solution or we have run out of time
return Best
```

Продолжительность *t*, конечно, настраивается, чтобы регулировать степень эксплуатационности алгоритма. При очень больших *t* локальному поиску уделяется больше внимания, что делает алгоритм более эксплуатационным; в то же время при малых *t* больше времени тратится на «внешний» алгоритм, что приводит к большему исследованию пространства поиска.

Существуют различные способы комбинирования эксплуатационного (и скорее всего локального) алгоритма с исследовательским (обычно глобальным) алгоритмом. Один пример

мы уже встречали: Локальный поиск со случайными перезапусками (Алгоритм 10), который комбинирует алгоритм локального поиска (Hill-Climbing) с глобальным алгоритмом (случайный поиск). Другой гибрид: Итеративный локальный поиск (Алгоритм 16), в котором локальный алгоритм работает внутри другого, более исследовательского локального алгоритма. Естественно, что алгоритм для локальных улучшений не обязан быть метаэвристическим: это может быть, например, алгоритм машинного обучения или эвристический алгоритм. В общем случае, семейство алгоритмов, где *некоторым* образом комбинируются *некоторый* алгоритм глобальной оптимизации с *некоторым* алгоритмом локальных улучшений часто обобщается под плохо определенным названием **Меметичные алгоритмы** (***Memetic Algorithms***)[28]. Несмотря на то, что этот термин охватывает достаточно широкий круг понятий, я считаю, что львиная доля меметичных алгоритмов, встречаемых в статьях, рассматривает гибриды глобальной оптимизации (здесь часто применяются эволюционные вычисления) и локального поиска, и, думаю, с такими алгоритмами они обычно и ассоциируются.

Возможно более удачным термином для этих алгоритмов является «**Ламарковские алгоритмы**» (***Lamarckian Algorithms***). Жан-Батист Ламарк был французским биологом, современником Американской революции, предложившим раннее, но ошибочное понятие эволюции. Его идея заключалась в том, что особи, улучшив себя в процессе жизни, затем генетически передавали потомкам свои приобретенные черты. К примеру, африканские животные, подобные лошадям, могли тянуться, чтобы достать фрукты на деревьях, растягивая свои шеи. Более длинные шеи затем передавали по наследству потомкам. Через несколько поколений – получите жирафа. В аналогичных гибридных алгоритмах особи улучшают себя в процессе вычисления приспособленности, а затем передают полученные улучшения своим потомкам. Еще одним подходящим названием было бы «**Алгоритмы с эффектом Болдуина**» (***Baldwin Effect Algorithm***), названные в честь более правдоподобного варианта ламаркианизма, который проявился в существующей теории эволюции. Намного позднее мы разберем еще один ламарковский алгоритм, когда будем рассматривать в Разделе 10.3 **Samuel** – алгоритм для оптимизации правил поведения со специальными операторами локальных улучшений.

Еще один подход к гибридизации заключается в переключении между двумя несвязанными алгоритмами. Например, **Модель обучаемой эволюции** (***Learnable Evolution Model, LEM***), рассматриваемая далее в Разделе 9.1, переключается между эволюцией и методом машинного обучения.

3.3.5 РАСПРЕДЕЛЕННЫЙ ПОИСК

Придуманный Фредом Гловером (*Fred Glover*) **Распределенный поиск с переопределением путей** (***Scatter Search with Path Relinking***)[29] комбинирует эволюционный и локальный поиск,

---

[28] По моему мнению, Меметичные алгоритмы имеют мало общего с *мемами* (*memes*), термином введенным Ричардом Доукинсом (Richard Dawkins) и обозначающим *идеи*, которые размножаются путем передачи от одного реципиента к другому. Примеры могут включать что угодно, начиная от религии и заканчивая цепочками «писем счастья». Использование термина *меметичные алгоритмы* оправдывается тем, что особи в этих алгоритмах локально улучшаются, точно также как и мемы могут быть «улучшены», прежде чем будут переданы другим людям. Однако я считаю, что отличительной чертой мемов является отнюдь не локальное улучшение, а репликация, иногда реализованная даже и в паразитической форме. В меметичных алгоритмах ничего такого нет.
Ричард Доукинс впервые представил термин *мем* в книге Richard Dawkins, 1976, The Selfish Gene, Oxford University Press. Термин меметичный алгоритм был придуман Пабло Москато (*Pablo Moscato*) в работе Pablo Moscato, 1989, On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, Technical Report 158–79, Caltech Concurrent Computation Program, California Institute of Technology.

[29] Гловер также разработал Поиск с запретом (Раздел 2.5). А также ввел термин «метаэвристика». Найти самую первую статью о распределенном поиске довольно тяжело, однако вот неплохое учебное пособие из числа

линейную рекомбинацию, $(m + l)$ и еще и явную процедуру добавления разнообразия (исследования) в один коктейль! Стандартный распределенный поиск с переопределением путей сложный и замысловатый, но мы будем рассматривать упрощенный вариант. Алгоритм комбинирует эксплуатационные механизмы (гибридный поиск, устойчивую эволюцию) с явным методом повышения разнообразия (в удачном случае, исследования) в работе системы. Алгоритм начинает работу с набора инициализированных особей, заданных пользователем. После этого алгоритм пытается произвести большое количество случайных особей, которые бы сильно отличались друг от друга и от исходных особей. Полученные особи плюс исходные формируют популяцию. Затем для улучшения каждой особи, производится локальный поиск.

После этого совершается следующая итерация. Сначала популяция сокращается до небольшого размера и включает только самых приспособленных особей и самых оригинальных (*diverse*) особей (для поддержания разнообразия). Затем производится некоторое подобие разбиения на пары и выполняется кроссинговер (обычно с использованием линейной рекомбинации) для этой меньшей популяции. В нашей версии линейная рекомбинация будет производиться для каждой пары особей в популяции, и дополнение будет применяться мутация. Затем производим локальный поиск для новых особей для их улучшения, добавляем их в популяцию и повторяем итерацию.

Для функции `ProduceDiverseIndividual` и процедуры определения самых оригинальных особей в Q (строка 17), понадобятся мера расстояния между особями: например, если две особи представляют вещественные векторы $\vec{u}$ и $\vec{v}$, то можно использовать Евклидово расстояние: $\sqrt{\sum_i (u_i - v_i)^2}$. Часто используются расстояния-метрики (обсуждается далее в Нишинге, Раздел 6.4). Отсюда можно определить оригинальность особи как сумму расстояний до остальных особей, т.е. для популяции P разнообразие P_i будет $\sum_j \text{distance}(P_i, P_j)$.

Итак, имеется способ определения наиболее «самой оригинальной» особи. Но создание «оригинальной» особи – без пяти минут произвол: Я вот сейчас нагенерирую много особей, а затем выберу подмножество, используя турнирную селекцию, основанную на непохожести на исходные особи. Можно было бы просто найти нетипичные значения генов в исходной популяции и сгенерировать новые особи с этими генами. Вот упрощенный алгоритм:

последних: Fred Glover, Manuel Laguna, and Rafael Marti´ı, 2003, Scatter search, in Ashish Ghosh and Shigeyoshi Tsutsui, editors, Advances in Evolutionary Computing: Theory and Applications, pages 519–538, Springer. Гловер также предпринял попытку дать полное описание общей схемы всего процесса в статье Fred Glover, 1998, A template for scatter search and path relinking, in Proceedings of the Third European Conference on Artificial Evolution, pages 1–51, Springer. Описанные здесь алгоритмы опираются на эти статьи.

Algorithm 37 *A Simplified Scatter Search with Path Relinking*

```
1: Seeds ← initial collection of individuals, defined by you
2: popsize ← desired population size
3: initsize ← initial sample size                ▷ The size of the initial population before truncation
4: t ← number of iterations to Hill-Climb
5: n ← number of individuals to be selected based on fitness
6: m ← number of individuals to be selected based on diversity

7: P ← Seeds
8: for initsize − ||Seeds|| times do
9:     P ← P ∪ {ProduceDiverseIndividual(P)}   ▷ Make an individual very different from what's in P
10: Best ← □
11: for each individual P_i ∈ P do                                   ▷ Do some hill-climbing
12:     P_i ← Hill-Climb(P_i) for t iterations                       ▷ Replace P_i in P
13:     AssessFitness(P_i)
14:     if Best = □ or Fitness(P_i) > Fitness(Best) then
15:         Best ← P_i
16: repeat                                                           ▷ The main loop
17:     B ← the fittest n individuals in P
18:     D ← the most diverse m individuals in P   ▷ Those as far from others in the space as possible
19:     P ← B ∪ D
20:     Q ← {}
21:     for each individual P_i ∈ P do
22:         for each individual P_j ∈ P where j ≠ i do
23:             Children C_a, C_b ← Crossover(Copy(P_i), Copy(P_j))      ▷ Use Line Recombination
24:             C_a ← Mutate(C_a)          ▷ Scatter Search wouldn't do this normally: but I do
25:             C_b ← Mutate(C_b)                                        ▷ Likewise
26:             C_a ← Hill-Climb(C_a) for t iterations
27:             C_b ← Hill-Climb(C_b) for t iterations
28:             AssessFitness(C_a)                  ▷ We next assess the fitness of C_a and C_b
29:             if Fitness(C_a) > Fitness(Best) then
30:                 Best ← C_a
31:             AssessFitness(C_b)
32:             if Fitness(C_b) > Fitness(Best) then
33:                 Best ← C_b
34:             Q ← Q ∪ {C_a, C_b}
35:     P ← Q ∪ P
36: until Best is the ideal solution or we have run out of time
37: return Best
```

3.4 ДИФФЕРЕНЦИАЛЬНАЯ ЭВОЛЮЦИЯ: АЛГОРИТМ АДАПТИВНОЙ МУТАЦИИ

Алгоритм **дифференциальной эволюции** (***Differential Evolution***, ***DE***) определяет размер `Mutates`, в основном основываясь на текущем разнообразии популяции. Если популяция сильно распределена в пространстве поиска, то `Mutate` будет производить существенные изменения, в противном случае, если популяция сконцентрирована в некой области, то мутации будут небольшими. ДЭ – это алгоритм с адаптивной мутацией (вроде правила одной пятой в эволюционных стратегиях), и был разработан Кеннетом Прайсом (***Kenneth Price***) и Рейнером Сторном (***Rainer Storn***)[30].

[30] Алгоритм ДЭ сформировался из серии работ, среди которых наиболее известной и, возможно, самой ранней является работа Rainer Storn

Операторы мутации в ДЭ используют сложение и вычитание векторов, поэтому алгоритм применим только в метрических векторных пространствах (булевские, целочисленные и вещественные). Разработано большое количество операторов мутации для ДЭ, но один из ранних, описанный здесь, весьма распространен и прост в описании. Для каждого *i*-го члена популяции генерируется потомок, с использованием 3 других особей и векторных операций сложения и вычитания их хромосом. Идея заключается в прибавлении некоторого вектора к вектору $\vec{a}$ одной из этих трех особей (которая принимается за точку отсчета для мутации). Прибавляемый вектор определяется как разность векторов двух других особей: $\vec{b} - \vec{c}$. Если популяция распределена, то, скорее всего, $\vec{b}$ и $\vec{c}$ будут расположены далеко друг от друга и вектор мутации будет большим. В противном случае, мутация получится малой. Таким образом, для распределенной в пространстве поиска популяции мутации будут значительно больше, чем когда алгоритм сойдется к областям с высокой приспособленностью. После этого потомок скрещивается с *i*-й особью, и если новая особь лучше *i*-й, то она заменяет *i*-ю особь (в дифференциальной эволюции есть много других операторов мутации, здесь не упоминаемых).

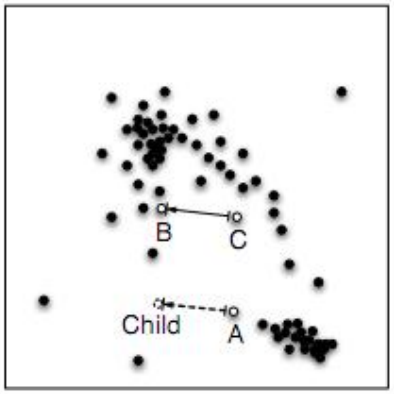

Рис. 15. Основной оператор мутации алгоритма дифференциальной эволюции. Копия особи A мутирует путем прибавления вектора смещения от особи C к особи B, формируя таким образом нового потомка

Текущее положение потомков полностью зависит от существующих родительских особей, а также от их возможных комбинаций для операций сложения и вычитания. Это означает, что алгоритм не является глобальным в смысле возможности достижения произвольной точки в пространстве поиска: последовательно выбирая особей и проводя мутации можно застрять на каком-нибудь «пятачке» в пространстве. Довольно странной особенностью алгоритма является мутация на каждом шаге только одной особи. Возможно было бы лучше осуществлять мутации всех особей в параллельном режиме (как в поколенческих алгоритмах), либо каждый раз выбирать случайное *i* (как в устойчивых алгоритмах).

В Разделе 3.6 будет обсуждаться Алгоритм ройной оптимизации (*Particle Swarm Optimization*), который более серьезно использует возможности мутации: в нем не производится выборочное обновление (*resampling*) популяции, а вместо этого особи мутируют в некотором четко выраженном направлении. На будущее будет полезным помнить о сходствах этого и сейчас рассматриваемого алгоритмов.

Простая реализация дифференциальной эволюции выглядит следующим образом:

---

and Kenneth Price, 1997, Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces, Journal of Global Optimization, 11(4), 341–359. Позднее Прайс, Сторн и Юрни Лампинен (Journi Lampinen) написали на эту тему довольно большую книгу: Kenneth Price, Rainer Storn, and Journi Lampinen, 2005, Differential Evolution: A Practical Approach to Global Optimization, Springer.

Algorithm 38 *Differential Evolution (DE)*

$\alpha \leftarrow$ mutation rate ▷ Commonly between 0.5 and 1.0, higher is more explorative
$popsize \leftarrow$ desired population size

$P \leftarrow \{\}$
for $popsize$ times **do**
    $P \leftarrow P \cup \{$new random individual$\}$
$\overrightarrow{Best} \leftarrow \square$
for each individual $\vec{i} \in P$ **do** ▷ DE assumes individuals are vectors in a metric space
    AssessFitness($\vec{i}$)
    if $\overrightarrow{Best} = \square$ or Fitness($\vec{i}$) > Fitness($\overrightarrow{Best}$) **then**
        $\overrightarrow{Best} \leftarrow \vec{i}$
repeat
    for each individual $\vec{i} \in P$ **do**
        $\vec{a} \leftarrow$ a copy of an individual, other than $\vec{i}$, chosen at random with replacement from P
        $\vec{b} \leftarrow$ a copy of an individual, other than $\vec{i}$ or $\vec{a}$, chosen at random with replacement from P
        $\vec{c} \leftarrow$ a copy of an individual, other than $\vec{i}$, $\vec{a}$, or $\vec{b}$, chosen at random with replacement from P
        $\vec{d} \leftarrow \vec{a} + \alpha(\vec{b} - \vec{c})$ ▷ Since individuals are vectors, mutation is vector arithmetic
        $\vec{j} \leftarrow$ one child from Crossover($\vec{d}$, Copy($\vec{i}$))
        AssessFitness($\vec{j}$)
        if Fitness($\vec{j}$) > Fitness($\vec{i}$) **then**
            $\vec{i} \leftarrow \vec{j}$
        if Fitness($\vec{j}$) > Fitness($\overrightarrow{Best}$) **then**
            $\overrightarrow{Best} \leftarrow \vec{j}$
until $\overrightarrow{Best}$ is the ideal solution or we have run out of time
return $\overrightarrow{Best}$

3.5 ВАРИАЦИИ НА ТЕМУ «ЭКСПЛУАТАЦИЯ ПРОТИВ ИССЛЕДОВАНИЯ»

Описанные эволюционные алгоритмы в большинстве представляют варианты одной и той же идеи: управление обновлением популяции для реализации различных эксплуатационных и исследовательских механизмов. Существуют различные подходы, но, в целом, можно выделить пять основных параметров, в той или иной степени влияющих на эксплуатацию и исследование:

- **Размер популяции**. Очень большая популяция напоминает случайный поиск. Очень маленькая – локальный поиск.

- **Насколько более вероятен выбор приспособленного родителя перед неприспособленным (*давление селекции*)**. При высоком давлении селекции приближаемся к локальному поиску. Низкое давление селекции напоминает *случайные блуждания* (*random walk*) (заметьте, не случайный поиск).

- **Сколько потомков генерируются от одной родительской особи.** Если создается много потомков, то они в основном напоминают своих родителей, и алгоритм становится похож на локальный поиск с наискорейшим подъемом (*steepest-ascent hill-climbing*). Если потомков мало, то алгоритм напоминает обычный локальный поиск.

- **Насколько сильно потомки отличаются от своих родителей (*Вероятность мутации*).** Большая вероятность мутации вносит большую неопределенность в

создаваемых потомков и тем самым приближает к случайному поиску. Очень малые вероятности мутации позволяют уточнять локальное решение.

- **Насколько долго приспособленные родители остаются в популяции (*Элитаризм*).** Если родительские особи не остаются в популяции, то алгоритм становится похож на случайные блуждания. В противном случае в значительной степени эксплуатируются оптимумы, в которые попали родители. В «агрессивных» сценариях (как ДЭ), уже для того, чтобы быть включенным в популяцию, потомок должен превзойти родителя.

3.6 ОПТИМИЗАЦИЯ С ИСПОЛЬЗОВАНИЕМ РОЯЩИХСЯ ЧАСТИЦ

Оптимизация с использованием роящихся частиц (ОРЧ) (***Particle Swarm Optimization***, ***PSO***) является методом стохастической оптимизации в чем-то схожим с эволюционными алгоритмами. Этот метод моделирует не эволюцию, а ройной и стайное поведение животных. Метод был разработан Джеймсом Кеннеди (*James Kennedy*) и Расселом Эберхартом (Russell Eberhart) в середине 90-х[31]. В отличие от популяционных методов ОРЧ не обновляет существующие популяции для создания новых. Вместо этого ОРЧ работает с одной статической популяцией, члены которой постепенно улучшаются с появлением информации о пространстве поиска. Данный метод представляет собой вид **направленной мутации** (***directed mutation***).

Как и алгоритм дифференциальной эволюции, ОРЧ работает в основном в многомерных, обычно вещественных, метрических пространствах. Это объясняется тем, что потенциальные решения в ОРЧ мутируют в направлении наилучших найденных решений, что требует рассмотрения метрического пространства (задача мутации, скажем, одного дерева в направлении другого, с формальной точки зрения, является весьма нетривиальной).

В силу использования вещественных пространств и изначального заимствования идей роя и стаи, исследователи ОРЧ обычно рассматривают потенциальные решения не как популяцию, а как **рой частиц** (***swarm of particles***). Эти частицы никогда не умирают (т.к. нет селекции), а изменяются направленной мутацией. Частица состоит из двух составляющих:

- Положение частицы в пространстве $\vec{x} = \langle x_1, x_2, ... \rangle$. Является эквивалентом генотипа в эволюционных алгоритмах.
- Скорость частицы, $\vec{v} = \langle v_1, v_2, ... \rangle$. Определяет скорость и направление, в котором частица перемещается в каждый момент времени. Иначе говоря, если $\vec{x}^{(t)}$ и $\vec{x}^{(t-1)}$ положения частицы в моменты времени t и $(t-1)$ соответственно, то $\vec{v} = \vec{x}^{(t)} - \vec{x}^{(t-1)}$.

Частицы начинают движение со случайно позиции и со случайным вектором скорости, который обычно вычисляется путем задания двух случайных точек в пространстве поиска и взятия вектора половинной длины от 1-й точки до 2-й (другие варианты: вектор с малыми значениями элементов или нулевой вектор). Также нужно отслеживать следующие моменты:

- Наиболее приспособленное положение $\vec{x}^*$, найденное частицей $\vec{x}$.

[31] Одной из самых первых статей по ОРЧ является статья James Kennedy and Russell Eberhart, 1995, Particle swarm optimization, in Proceedings of IEEE International Conference on Neural Networks, pages 1942–1948. Позже Эберхарт, Кеннеди и Юхуи Ши (*Yuhui Shi*) написали на эту тему книгу James Kennedy, Russell Eberhart, and Yuhui Shi, 2001, Swarm Intelligence, Morgan Kaufmann.

- Наиболее приспособленное положение $\mathbf{r}_{x^+}$, найденное всеми **информаторами** (***informants***) частицы $\mathbf{x}$. В ранних версиях алгоритма информаторы составлялись из соседних частиц, и давали информацию о найденных наилучших положениях. В настоящее время информаторы $\mathbf{x}$ обычно представляет небольшое множество частиц, случайно выбираемых на каждой итерации, причем $\mathbf{x}$ всегда входит в это множество.

- Наиболее приспособленное положение $\mathbf{r}_{x^!}$, найденное всеми частицами.

На каждой итерации производятся следующие шаги:

1. Вычисляются приспособленности частиц и при необходимости обновляется информация о наилучших найденных положениях.

2. Определение операции `Mutate`. Для каждой частицы $\mathbf{x}$ производится обновление вектора ее скорости $\mathbf{v}$ прибавлением к нему в некоторой пропорции векторов, указывающих на $\mathbf{x}*$, $\mathbf{r}_{x^+}$ и $\mathbf{r}_{x^!}$, с наложением небольшого шума на каждый вектор (различные случайные значения по каждому из направлений).

3. Мутация частицы в направлении ее вектора скорости.

Алгоритм выглядит следующим образом:

Algorithm 39 *Particle Swarm Optimization (PSO)*

swarmsize ← desired swarm size
α ← proportion of velocity to be retained
β ← proportion of personal best to be retained
γ ← proportion of the informants' best to be retained
δ ← proportion of global best to be retained
ϵ ← jump size of a particle

$P \leftarrow \{\}$
for *swarmsize* times **do**
    $P \leftarrow P \cup \{$new random particle $\vec{x}$ with a random initial velocity $\vec{v}\}$
$\overrightarrow{Best} \leftarrow \square$
repeat
    for each particle $\vec{x} \in P$ with velocity $\vec{v}$ **do**
        AssessFitness($\vec{x}$)
        if $\overrightarrow{Best} = \square$ or Fitness($\vec{x}$) > Fitness($\overrightarrow{Best}$) **then**
            $\overrightarrow{Best} \leftarrow \vec{x}$
    for each particle $\vec{x} \in P$ with velocity $\vec{v}$ **do** ▷ Determine how to Mutate
        $\vec{x}^* \leftarrow$ previous fittest location of $\vec{x}$
        $\vec{x}^+ \leftarrow$ previous fittest location of informants of $\vec{x}$ ▷ (including $\vec{x}$ itself)
        $\vec{x}^! \leftarrow$ previous fittest location any particle
        for each dimension i **do**
            $b \leftarrow$ random number from 0.0 to β inclusive
            $c \leftarrow$ random number from 0.0 to γ inclusive
            $d \leftarrow$ random number from 0.0 to δ inclusive
            $v_i \leftarrow \alpha v_i + b(x_i^* - x_i) + c(x_i^+ - x_i) + d(x_i^! - x_i)$
    for each particle $\vec{x} \in P$ with velocity $\vec{v}$ **do** ▷ Mutate
        $\vec{x} \leftarrow \vec{x} + \epsilon\vec{v}$
until $\overrightarrow{Best}$ is the ideal solution or we have run out of time
return $\overrightarrow{Best}$

Работа алгоритма зависит от пяти параметров:

- a – определяет, какая сохраняется доля исходного вектора скорости.
- b – определяет приоритет лучшего положения, найденного самой частицей. Если b большое, то частица преимущественно движется в направлении своего лучшего положения, что «разбивает» весь рой на отдельных локальных оптимизаторов и уменьшает роль совместного поиска.
- g – определяет приоритет лучшего положения, найденного информаторами. В результате может получиться эффект усреднения влияния b и d. Количество информаторов также влияет на этот коэффициент, т.к. (предполагая, что информаторы выбираются случайно) чем больше у частицы информаторов, тем больше «глобальное» лучшее положение будет превалировать над лучшим положением частицы.
- d – определяет приоритет лучшего положения, найденного всеми частицами (глобальное лучшее). При большом значении d частицы движутся в сторону наилучшего найденного положения, что превращает алгоритм в один большой алгоритм локального поиска вместо набора разделенных локальных алгоритмов. Возможно, из-за угрозы создания слишком сильно эксплуатирующей системы, в современных реализациях часто d устанавливают равным 0.
- e – определяет как быстро частица движется. Если e большое, то частица перемещается большими скачками в сторону областей с высокой приспособленности и случайно может их «перепрыгнуть». Таким образом, большое e значение позволяет системе быстро сходиться к наилучшим областям, но затрудняет уточнение решения, так же, как и в локальном поиске. Обычно, e равно 1.

То, насколько эти параметры влияют на соотношение эксплуатации и исследования, обсуждалось в разделе 3.5.