МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ» (ДГТУ)**

Кафедра Выч. системы и инфор. безопасность
Направление 09.04.02 Информационные системы и технологии
(магистратура)

Дисциплина Эвристические алгоритмы

# КОНСПЕКТ ЛЕКЦИЙ

## Лекция №1

**Тема: Введение. Основные термины, определения и понятия.**

**Вопросы:**
1. **Определение алгоритма.**
2. **Отличие эвристического алгоритма от других алгоритмов.**

**Алгоритм** - точное предписание исполнителю совершить определенную последовательность действий для достижения поставленной цели за конечное число шагов.

Одним из фундаментальных понятий в информатике является понятие алгоритма. Происхождение самого термина «алгоритм» связано с математикой. Это слово происходит от Algorithmi – латинского написания имени Мухаммеда аль-Хорезми (787 – 850) выдающегося математика средневекового Востока. В своей книге "Об индийском счете" он сформулировал правила записи натуральных чисел с помощью арабских цифр и правила действий над ними столбиком. В дальнейшем алгоритмом стали называть точное предписание, определяющее

последовательность действий, обеспечивающую получение требуемого результата из исходных данных.

Алгоритм может быть предназначен для выполнения его человеком или автоматическим устройством. Создание алгоритма, пусть даже самого простого, - процесс творческий. Он доступен исключительно живым существам, а долгое время считалось, что только человеку. В XII в. был выполнен латинский перевод его математического трактата, из которого европейцы узнали о десятичной позиционной системе счисления и правилах арифметики многозначных чисел. Именно эти правила в то время называли алгоритмами.

Данное выше определение алгоритма нельзя считать строгим – не вполне ясно, что такое «точное предписание» или «последовательность действий, обеспечивающая получение требуемого результата».

Поэтому обычно формулируют несколько **общих свойств алгоритмов**, позволяющих отличать алгоритмы от других инструкций.

Такими свойствами являются:

• ***Дискретность*** (прерывность, раздельность) – алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов. Каждое действие, предусмотренное алгоритмом, исполняется только после того, как закончилось исполнение предыдущего.

• ***Определенность*** – каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче.

• ***Результативность (конечность)*** – алгоритм должен приводить к решению задачи за конечное число шагов.

• ***Массовость*** – алгоритм решения задачи разрабатывается в общем виде, то есть, он должен быть применим для некоторого класса задач, различающихся только исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма.

На основании этих свойств иногда дается определение алгоритма, например: *“Алгоритм – это последовательность математических, логических*

*или вместе взятых операций, отличающихся детерменированностью, массовостью, направленностью и приводящая к решению всех задач данного класса за конечное число шагов”.*

Такая трактовка понятия “алгоритм” является неполной и неточной.

Во-первых, неверно связывать алгоритм с решением какой-либо задачи. Алгоритм вообще может не решать никакой задачи.

Во-вторых, понятие “массовость” относится не к алгоритмам как к таковым, а к математическим методам в целом. Решение поставленных практикой задач математическими методами основано на абстрагировании – мы выделяем ряд существенных признаков, характерных для некоторого круга явлений, и строим на основании этих признаков математическую модель, отбрасывая несущественные признаки каждого конкретного явления. В этом смысле любая математическая модель обладает свойством массовости. Если в рамках построенной модели мы решаем задачу и решение представляем в виде алгоритма, то решение будет “массовым” благодаря природе математических методов, а не благодаря “массовости” алгоритма.

## Виды алгоритмов

Виды алгоритмов как логико-математических средств отражают указанные компоненты человеческой деятельности и тенденции, а сами алгоритмы в зависимости от цели, начальных условий задачи, путей ее решения, определения действий исполнителя подразделяются следующим образом:

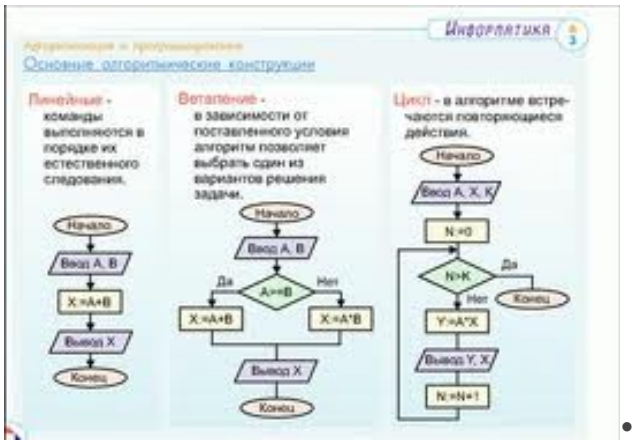
• ***Механические алгоритмы***, или иначе детерминированные, жесткие (например, алгоритм работы машины, двигателя и т.п.);

• ***Гибкие алгоритмы***, например стохастические, т.е. вероятностные и эвристические. Механический алгоритм задает определенные действия, обозначая их в единственной и достоверной последовательности, обеспечивая тем самым

однозначный требуемый или искомый результат, если выполняются те условия процесса, задачи, для которых разработан алгоритм.

• ***Вероятностный*** (стохастический) алгоритм дает программу решения задачи несколькими путями или способами, приводящими к вероятному достижению результата.

• ***Эвристический*** алгоритм (от греческого слова “эврика”) – это такой алгоритм, в котором достижение конечного результата программы действий однозначно не предопределено, так же как не обозначена вся последовательность действий, не выявлены все действия исполнителя. К эвристическим алгоритмам относят, например, инструкции и предписания. В этих алгоритмах используются универсальные логические процедуры и способы принятия решений, основанные на аналогиях, ассоциациях и прошлом опыте решения схожих задач.




• ***Линейный алгоритм*** – набор команд (указаний), выполняемых последовательно во времени друг за другом.

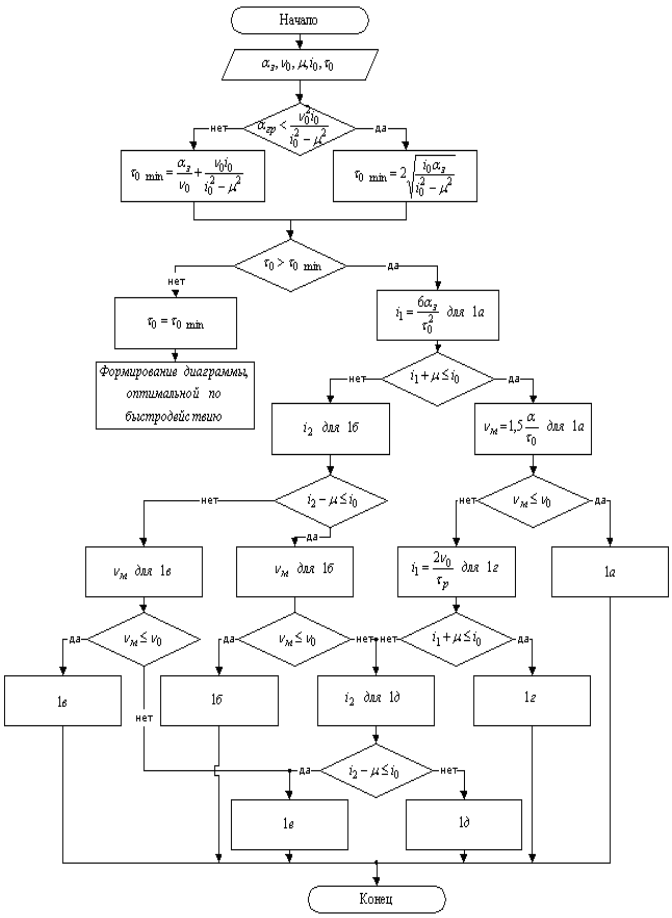
• ***Разветвляющийся*** алгоритм – алгоритм, содержащий хотя бы одно условие, в результате проверки которого ЭВМ обеспечивает переход на один из двух возможных шагов.

• ***Циклический алгоритм*** – алгоритм, предусматривающий многократное повторение одного и того же действия (одних и тех же операций) над новыми исходными данными. К циклическим алгоритмам сводится большинство методов вычислений, перебора вариантов.

Цикл программы – последовательность команд (серия, тело цикла), которая может выполняться многократно (для новых исходных данных) до удовлетворения некоторого условия.

***Вспомогательный*** (подчиненный) алгоритм (процедура) – алгоритм, ранее разработанный и целиком используемый при алгоритмизации конкретной задачи. В некоторых случаях при наличии одинаковых последовательностей указаний (команд) для различных данных с целью сокращения записи также выделяют вспомогательный алгоритм.

На всех этапах подготовки к алгоритмизации задачи широко используется структурное представление алгоритма.




***Структурная*** (блок-, граф-) схема алгоритма – графическое изображение алгоритма в виде схемы связанных между собой с помощью стрелок (линий перехода) блоков – графических символов, каждый из которых соответствует одному шагу алгоритма. Внутри блока дается описание соответствующего действия.

Графическое изображение алгоритма широко используется перед программированием задачи вследствие его наглядности, т.к. зрительное восприятие обычно облегчает процесс написания программы, ее корректировки при возможных ошибках, осмысливание процесса обработки информации.

Можно встретить даже такое утверждение: “Внешне алгоритм представляет собой схему – набор прямоугольников и других символов, внутри которых записывается, что вычисляется, что вводится в машину и что выдается на печать и другие средства отображения информации “. Здесь форма представления алгоритма смешивается с самим алгоритмом.

## Требования, предъявляемые к алгоритму

***Первое правило*** – при построении алгоритма прежде всего необходимо задать множество объектов, с которыми будет работать алгоритм. Формализованное (закодированное) представление этих объектов носит название данных. Алгоритм приступает к работе с некоторым набором данных, которые называются входными, и в результате своей работы выдает данные, которые называются выходными. Таким образом, алгоритм преобразует входные данные в выходные. Это правило позволяет сразу отделить алгоритмы от “методов” и “способов”. Пока мы не имеем формализованных входных данных, мы не можем построить алгоритм.

***Второе правило*** – для работы алгоритма требуется память. В памяти размещаются входные данные, с которыми алгоритм начинает работать, промежуточные данные и выходные данные, которые являются результатом работы алгоритма. Память является дискретной, т.е. состоящей из отдельных ячеек. Поименованная ячейка памяти носит название переменной. В теории алгоритмов размеры памяти не ограничиваются, т. е. считается, что мы можем предоставить алгоритму любой необходимый для работы объем памяти. В школьной “теории алгоритмов” эти два правила не рассматриваются. В то же время практическая работа с алгоритмами (программирование) начинается именно с реализации этих правил.

В языках программирования распределение памяти осуществляется декларативными операторами (операторами описания переменных). В языке

Бейсик не все переменные описываются, обычно описываются только массивы. Но все равно при запуске программы транслятор языка анализирует все идентификаторы в тексте программы и отводит память под соответствующие переменные.

***Третье правило*** – дискретность. Алгоритм строится из отдельных шагов (действий, операций, команд). Множество шагов, из которых составлен алгоритм, конечно.

***Четвертое правило*** – детерменированность. После каждого шага необходимо указывать, какой шаг выполняется следующим, либо давать команду остановки. Пятое правило – сходимость (результативность). Алгоритм должен завершать работу после конечного числа шагов. При этом необходимо указать, что считать результатом работы алгоритма.

***Минимаксный алгоритм***

Дерево игры просматривается только вплоть до некоторой глубины (обычно на несколько ходов), а затем для всех концевых вершин дерева поиска вычисляются оценки при помощи некоторой оценочной функции. Идея состоит в том, чтобы, получив оценки этих терминальных поисковых вершин, не продвигаться дальше и получить тем самым экономию времени. Далее, оценки терминальных позиций распространяются вверх по дереву поиска в соответствии с минимаксным принципом. В результате все вершины дерева поиска получают свои оценки. И наконец, игровая программа, участвующая в некоторой реальной игре, делает свой ход - ход, ведущий из исходной (корневой) позиции в наиболее перспективного (с точки зрения оценки) ее преемника.

Обратите внимание на то, что мы здесь делаем определенное различие между "деревом игры" и "деревом поиска". Дерево поиска - это только часть дерева игры (его верхняя часть), т. е. та его часть, которая была явным образом порождена в процессе поиска. Таким образом, терминальные поисковые позиции совсем не обязательно должны совпадать с терминальными позициями самой игры.

Очень многое зависит от оценочной функции, которая для большинства игр, представляющих интерес, является приближенной эвристической оценкой шансов

на выигрыш одного из участников игры. Чем выше оценка, тем больше у него шансов выиграть и чем ниже оценка, тем больше шансов на выигрыш у его противника. Один из участников игры всегда стремится к высоким оценкам, а другой - к низким, мы дадим им имена МАКС и МИН соответственно. МАКС всегда выбирает ход с максимальной оценкой; в МИН всегда выбирает ход с минимальной оценкой. На рисунке видно, что уровни позиций с ходом МАКС'а чередуются с уровнями позиций с ходом МИН'а. Оценки вершин нижнего уровня определяются при помощи оценочной функции. Оценки всех внутренних вершин можно определить, двигаясь снизу вверх от уровня к уровню, пока мы не достигнем корневой вершины. В результате оценка корня оказывается равной 4 (рис), и, соответственно, лучшим ходом МАКС'а из позиции a - a-b. Лучший ответ МИН'а на этот ход - b-d, и т.д. Эту последовательность ходов называют также основным вариантом. Основной вариант показывает, какова "минимаксно-оптимальная" игра для обоих участников. Оценки всех позиций, входящих в основной вариант, совпадают.

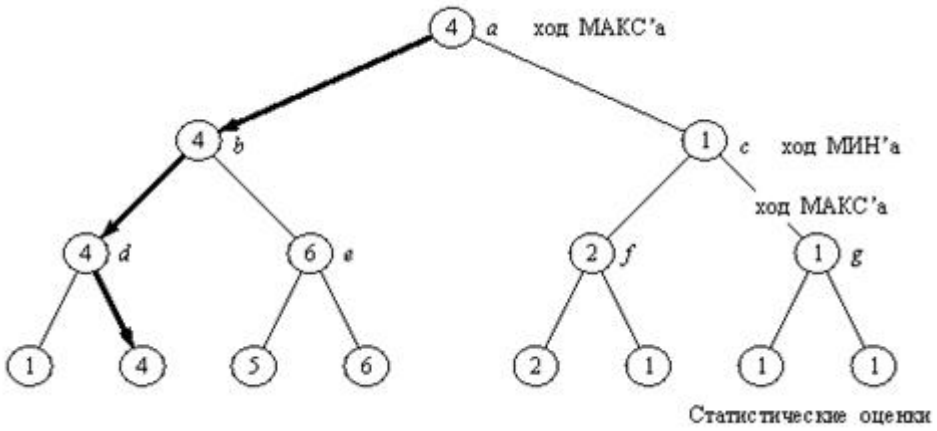



Рис. Статические (нижний уровень) и минимаксные рабочие оценки вершин дерева поиска. Выделенные ходы образуют основной вариант, т. е. минимаксно-оптимальную игру с обеих сторон.

Мы различаем два вида оценок: оценки вершин нижнего уровня и оценки внутренних вершин (рабочие оценки). Первые из них называются также "статическими", так как они вычисляются при помощи "статической" оценочной

функции, в противоположность рабочим оценкам, получаемым "динамически" при распространении статических оценок вверх по дереву.

Правила распространения оценок можно сформулировать следующим образом. Будем обозначать статическую оценку позиции P через v( P), а ее рабочую оценку - через V( P). Пусть P1, ..., Pn - разрешенные преемники позиции P. Тогда соотношения между статическими и рабочими оценками можно записать так: V( P) = v( P), если P - терминальная позиция дерева поиска (n=0); V( P) = max V( Pi ), если P - позиция с ходом МАКС'а; V( P) = min V( Pi ), если P - позиция с ходом МИН'а

## Лекция№2

**Тема: Задача обеспечения связности.**

**Вопросы:**

**1. Определение задачи связности.**

**2. Абстрактные операции.**

Выбор наилучшего алгоритма выполнения конкретной задачи может оказаться сложным процессом, возможно, требующим сложного математического анализа. Направление компьютерных наук, занимающееся изучением подобных вопросов, называется анализом алгоритмов. Анализ многих изучаемых алгоритмов показывает, что они имеют прекрасную производительность; о хорошей работе других известно просто из опыта их применения. Наша основная цель — изучение приемлемых алгоритмов выполнения важных задач, хотя значительное внимание будет уделено также сравнительной производительности различных методов. Не следует использовать алгоритм, не имея представления о ресурсах, которые могут потребоваться для его выполнения, поэтому мы стремимся знать,

как могут выполняться используемые алгоритмы.

## 1.2 Пример задачи: связность

Предположим, что имеется последовательность пар целых чисел, в которой каждое целое число представляет объект некоторого типа, а пара p-^ интерпретируется в значении "p связано с я". Мы предполагаем, что отношение "связано с" является транзитивным: если p связано с q, а я связано с г, то p связано с г. Задача состоит в написании программы для исключения лишних пар из набора: когда программа вводит пару p-ц, она должна выводить эту пару только в том случае, если просмотренные до данного момента пары не предполагают, что p связано с q. Если в соответствии с ранее просмотренными парами следует, что p связано с программа должна игнорировать пару p^ и переходить ко вводу следующей пары. Пример такого процесса показан на рис. 1.1.

Задача состоит в разработке программы, которая может запомнить достаточный объем информации о просмотренных парах, чтобы решить, связана ли новая пара объектов. Достаточно неформально задачу разработки такого метода мы называем задачей связности. Эта задача возникает в ряде важных приложений. Для подтверждения всеобщего характера этой задачи мы кратко рассмотрим три примера.

Например, целые числа могли бы представлять компьютеры в большой сети, а пары могли бы представлять соединения в сети. Тогда такая программа могла бы использоваться для опреде-

3-4 3-4

4-9 4-9

8-0 8-0

2-3 2-3

5-6 5-6

2-9

5-9 5-9

7-3 7-3

4-8 4-8

5-6

0-2

6-1 6-1

2-3-4-9

5-6

0-8-4-3-2

РИСУНОК 1.1 ПРИМЕР СВЯЗНОСТИ

При заданной последовательности пар целых чисел, представляющих связь между объектами (слева) задачей алгоритма связности заключается в выводе тех пар, которые обеспечивают новые связи (в центре). Например, пара 2-9 не должна выводиться, поскольку связь 2-3-4-9 определяется ранее указанными связями (подтверждение этого показано справа).

Часть 1. Анализ

того, нужно ли устанавливать новое прямое соединение между p и я, чтобы иметь возможность обмениваться информацией, или же можно было бы использовать существующие соединения для установки коммуникационного пути. В подобных приложениях может потребоваться обработка миллионов точек и миллиардов или более соединений. Как мы увидим, решить задачу для такого приложения было бы невозможно без эффективного алгоритма.

Аналогично, целые числа могли бы представлять контакты в электрической сети, а пары могли бы представлять связывающие их проводники. В этом случае программу можно было бы использовать для определения способа соединения всех точек без каких-либо избыточных соединений, если это возможно. Не существует никакой гарантии, что ребер списка окажется достаточно для соединения всех точек — действительно, вскоре мы увидим, что определение факта, так ли это, может быть основным применением нашей программы.

Рисунок 1.2 иллюстрирует эти два типа приложений на более сложном примере. Изучение этого рисунка дает представление о сложности задачи связности: как можно быстро выяснить, являются ли любые две заданные точки в такой сети связанными?

РИСУНОК 1.2 БОЛЬШОЙ ПРИМЕР СВЯЗНОСТИ

Объекты, задействованные в задаче связности, могут представлять точки соединений, а пары могут быть соединениями между ними, как показано в этом идеализированном примере, который мог бы представлять провода, соединяющие здания в городе или компоненты в компьютерной микросхеме. Это графическое представление позволяет человеку выявить несвязанные узлы, но алгоритм должен работать только с переданными ему парами целых чисел. Связаны ли два узла, помеченные черными точками ?

Такие приложения, как задача установления эквивалентности имен переменных, описанная в предыдущем абзаце, требует, чтобы целое число было сопоставлено с каждым отдельным именем переменной. Это сопоставление подразумевается также в описанных приложениях сетевого соединения и соединения в электрической цепи.. Таким образом, в этой главе без ущерба для общности можно предположить, что имеется N объектов с целочисленными именами от 0 до 1.

Нам требуется программа, которая выполняет конкретную, вполне определенную задачу. Существует множество других связанных с этой задач, решение которых также может потребоваться. Одна из первых задач, с которой приходится сталкиваться при разработке алгоритма — необходимость убедиться, что задача определена приемлемым образом. Чем больше требуется от алгоритма, тем больше времени и объема памяти может потребоваться для выполнения задачи. Это соотношение невозможно в точности определить заранее, и часто определение задачи приходится изменять, когда выясняется, что ее трудно решить либо решение требует слишком больших затрат, или же когда, при удачном стечении обстоятельств, выясняется, что алгоритм может предоставить более полезную информацию, чем требовалось от него в исходном определении. Например, приведенное определение задачи связности требует только, чтобы программа как-либо узнавала, является ли данная пара p-ц связанной, но не была способна демонстрировать любой или все способы соединения этой пары. Упомянутые в предыдущем абзаце определения требуют больше информации, чем первоначальное; может также требоваться меньше информации. Например, может требоваться просто ответить на вопрос: "Достаточно ли M связей для соединения

всех N объектов?". Эта задача служит иллюстрацией того, что для разработки эффективных алгоритмов часто требуется выполнение умозаключений об абстрактных обрабатываемых объектах на высоком уровне. В данном случае из фундаментальных положений теории графов следует, что все N объектов связаны тогда и только тогда, когда количество пар, образованных алгоритмом решения задачи связности, равно точно УУ— 1 (см. раздел 5.4). Иначе говоря, алгоритм решения задачи связности никогда не образует более N — 1 пар, поскольку как только он образует N — 1 пару, любая встретившаяся после этого пара будет уже связанной. Соответственно, можно создать программу, отвечающую "да-нет" на только что поставленный вопрос, изменив программу, которая решает задачу связности, на такую, которая увеличивает значение



счетчика, а не записывает ранее не связанную пару, отвечая "да", когда значение счетчика достигает jV — 1, и "нет", если это не происходит. Этот вопрос — всего лишь один из множества, которые могут возникнуть относительно связности. Входной набор пар называется графом (Graph), а выходной набор пар — остовным деревом (spanning tree) этого графа, которое связывает все объекты. Свойства графов, остов-ных деревьев и всевозможные связанные с ними алгоритмы будут рассматриваться в части 7.

Имеет смысл попытаться определить основные операции, которые будут выполняться, и таким образом сделать любой алгоритм, разрабатываемый для решения задачи связности, полезным для ряда аналогичных задач. В частности, при получении каждой новой пары вначале необходимо определить, представляет ли она новое соединение, а затем внедрить информацию об обнаруженном соединении в общую картину о связности объектов для проверки соединений, которые будут наблюдаться в будущем. Мы инкапсулируем эти две задачи в виде абстрактных операций, считая целочисленные вводимые значения представляющими элементы в абстрактных наборах, а затем разработаем алгоритмы и структуры данных, которые могут:

¦ находить набор, содержащий данный элемент

¦ замещать наборы, содержащие два данных элемента, их объединением.

Организация алгоритмов посредством этих абстрактных операций, похоже, не препятствует никаким опциям решения задачи связности, а сами эти операции могут оказаться полезными при решении других задач. Разработка уровней абстракции с еще большими возможностями — основной процесс в компьютерных науках в целом и в разработке алгоритмов в частности, и в этой книге мы будем обращаться к нему многократно. В этой главе мы используем неформальное абстрактное представление для разработки программ решения задачи связности; внедрение абстракций в код C++ демонстрируется в главе 4. Задача связности легко решается посредством абстрактных операций find (поиск) и union (объединение). После считывания новой пары p-q из ввода мы выполняем операцию find для каждого члена пары. Если члены пары находятся в одном наборе, мы переходим к следующей паре; если нет, то выполняем операцию union и записываем пару. Наборы представляют связанные компоненты: поднаборы объектов, характеризующиеся тем, что любые два объекта в данном компоненте связаны. Этот подход сводит разработку алгоритмического решения задачи связности к задачам определения структуры данных, которая представляет наборы, и разработке алгоритмов union и find, которые эффективно используют эту структуру данных.

**Лекция№3**

**Тема: Алгоритмы обьединения-поиска.**

**Вопросы:**

1. . **Реализация алгоритма быстрого поиска.**
2. **Реализация быстрого обьединения.**

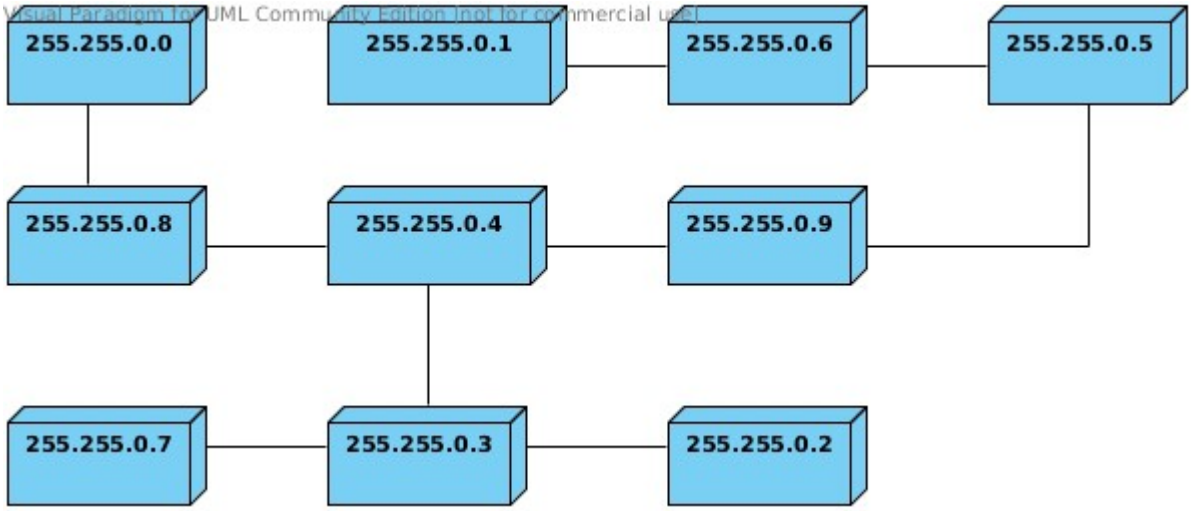
Суть задачи связности

Имеется последовательность пар неких элементов, каждая пара value1 - value2 интерпретируется, как “value1 связано с value2”. Отношение “связано с” является транзитивным, то есть, если value1 связано с value2, а value2 связано с value3, то

value1 связано с value3. Задача заключается написании программы, которая отфильтровывает существующие связи из набора.

Рассмотрим задачу на примере

Структура, состоящая из пар элементов, могла бы представлять собой компьютерную сеть, в которой элементы - это компьютеры, которые связаны между собой сетевым кабелем:




Предположим, что на вход нашей программы постоянно поступают по два IP-адреса, нам нужно определить, стоит ли устанавливать новое соединение между соответствующими компьютерами, или же можно воспользоваться уже существующим соединением. При этом не забываем, что следует учитывать не только прямые, но и транзитивные связи.

Первое решение, которое приходит в голову - это запоминать связи каждого узла со всеми другими узлами сети. Вариант, конечно, но это могут быть очень большие объемы данных, которые нужно запомнить в памяти, ведь узлов и связей может быть намного больше, чем на нашей простой схеме.

Решением данной задачи может быть применение алгоритма быстрого поиска, решающего задачу связности. Суть алгоритма основывается на использовании связывающего массива и том утверждении, что i-й и j-й узлы связаны между собой тогда и только тогда, когда i-е и j-е значения массива одинаковы.

## Метод быстрого объединения

В основе метода быстрого объединения лежит массив, индексами элементов массива являются наши IP-адреса, а значение каждого элемента указывает на другой элемент, таким образом строится некая нециклическая структура - дерево. Если объекты находятся в одном дереве, значит они уже связаны между собой. Для того, чтоб определить, находятся ли объекты в одном дереве, нужно “подняться” вверх по дереву и проверить или у обоих элементов корневой элемент общий.

Проиллюстрируем данный алгоритм:

Важно понимать, что узлы в формируемой структуре не обязательно будут связаны в том порядке, в котором они поступают на вход приложения, так как связываются не сами узлы, а их корневые элементы. В отличии от метода быстрого поиска, метод быстрого объединения требует меньше вычислений для объединения узлов, за счет большего количества вычислений для поиска, собственно, отсюда и его название.

## Лекция№4

## Тема: Алгоритмы поиска.

**Вопросы:**

1. **Линейный поиск.**
2. **бинарный поиск.**

Последовательный (линейный) поиск

**Последовательный (линейный) поиск** – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

*Алгоритм последовательного поиска*

Шаг 1. Полагаем, что значение переменной цикла i=0.

Шаг 2. Если значение элемента массива x[i] равно значению ключа key, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу i=i+1.

Шаг 3. Если i<k, где k – число элементов массива x, то выполняется Шаг 2, в противном случае – работа алгоритма завершена и возвращается значение равное -1.

При наличии в массиве нескольких элементов со значением key данный алгоритм находит только первый из них (с наименьшим индексом).

```
int LinearSearch(int *x, int k, int key){
  int i = 0;
  for ( i = 0 ; i < k ; i++ )
    if ( x[i] == key )
      break;
  return i < k ? i : -1;
}
```

Время выполнения данного алгоритма поиска для вещественных чисел $n/\varepsilon$, где n – количество элементов множества, а $\varepsilon$ – точность. Поиск на дискретном множестве из n элементов осуществляется в худшем случае за n итераций, а в среднем этот алгоритм требует n/2 итераций цикла. Следовательно, временная сложность *последовательного поиска* пропорциональна O(n). Никаких ограничений на порядок элементов в массиве данный алгоритм не накладывает.

Недостатком рассматриваемого алгоритма поиска является то, что в худшем случае осуществляется просмотр всего массива. Поэтому данный алгоритм используется, если множество содержит небольшое количество элементов.

Достоинства *последовательного поиска* заключаются в том, что он прост в реализации, не требует сортировки значений множества, дополнительной памяти и дополнительного анализа функций. Следовательно, может работать в потоковом режиме при непосредственном получении данных из любого источника.

Существует модификация алгоритма *последовательного поиска*, которая ускоряет поиск. Эта модификация является небольшим усовершенствованием рассмотренного алгоритма поиска.

Идея *поиска с барьером* состоит в том, чтобы не проверять каждый раз в цикле условие, связанное с границами множества. Это можно обеспечить, установив в данном множестве так называемый барьер. Под барьером понимается любой элемент, который удовлетворяет *условию поиска*. Тем самым будет ограничено изменение индекса.

Выход из цикла, в котором теперь остается только условие поиска, может произойти либо на найденном элементе, либо на барьере. Существует два способа установки барьера: дополнительным элементом или вместо крайнего элемента массива.

```
//описание функции последовательного поиска с барьером
int LinearSearchWithBarrier(int *x, int k, int key){
   x = (int *)realloc(x,(k+1)*sizeof(int));
   x[k] = key;
   int i = 0;
   while ( x[i] != key )
      i++;
   return i < k ? i : -1;
}
```

Заметим, что поиск с барьером работает быстрее, но временная сложность алгоритма остается такой же O(n), где n – количество элементов множества. Гораздо больший интерес представляют методы, не только работающие быстро, но и реализующие алгоритмы с меньшей сложностью.

Бинарный (двоичный) поиск

**Бинарный (двоичный, дихотомический) поиск** – это поиск заданного элемента на упорядоченном множестве, осуществляемый путем неоднократного деления этого множества на две части таким образом, что искомый элемент попадает в одну из этих частей. Поиск заканчивается при совпадении искомого элемента с

элементом, который является границей между частями множества или при отсутствии искомого элемента.

*Бинарный поиск* применяется к отсортированным множествам и заключается в последовательном *разбиении множества* пополам и поиска элемента только в одной половине на каждой итерации.

Таким образом, идея этого метода заключается в следующем. Поиск нужного значения среди элементов упорядоченного массива (по возрастанию или по убыванию) начинается с определения значения центрального элемента этого массива. Значение данного элемента сравнивается с искомым значением и в зависимости от результатов сравнения предпринимаются определенные действия. Если искомое и центральное значения оказываются равны, то поиск завершается успешно. Если искомое значение меньше центрального или больше, то формируется массив, состоящий из элементов, находящихся слева или справа от центрального соответственно. Затем поиск повторяется в новом массиве (рис. 37.1).

*Алгоритм бинарного поиска*

Шаг 1. Определить номер среднего элемента массива middle=(high+*low*)/2.

Шаг 2. Если значение среднего элемента массива равно искомому, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу.

Шаг 3. Если искомое значение больше значения среднего элемента, то возьмем в качестве массива все элементы справа от среднего, иначе возьмем в качестве массива все элементы слева от среднего (в зависимости от характера упорядоченности). Перейдем к Шагу 1.

В массиве может встречаться несколько элементов со значениями, равными ключу. Данный алгоритм находит первый совпавший с ключом элемент, который в порядке следования в массиве может быть ни первым, ни последним среди равных ключу. Например, в массиве чисел 1, 5, 5, 5, 5, 5, 5, 7, 8 с ключом key =5 совпадет элемент с порядковым номером 4, который не относится ни к первому, ни к последнему.

Существуют две модификации рассматриваемого алгоритма для поиска первого и последнего вхождения. Все зависит от того, как выбирается средний элемент: округлением в меньшую или большую сторону. В первом случае средний элемент относится к левой части массива, а во втором – к правой.

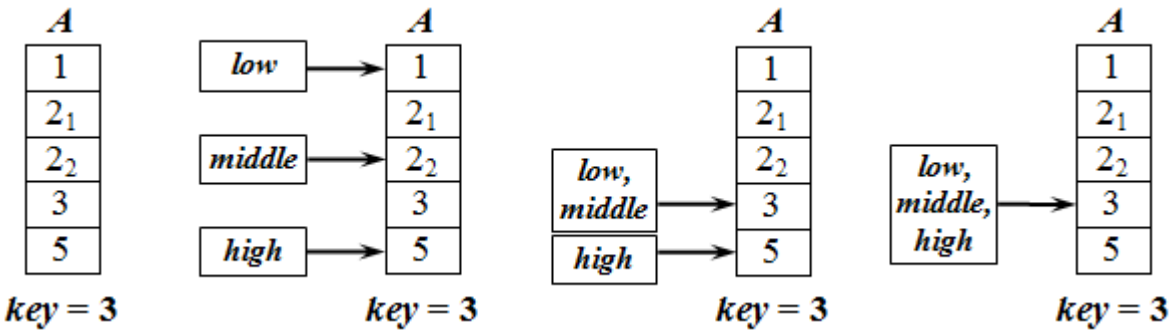



**Рис. 37.1.** Демонстрация алгоритма бинарного поиска

```
//описание функции бинарного поиска
int BinarySearch(int *x, int k, int key){
  bool found = false;
  int high = k - 1, low = 0;
  int middle = (high + low) / 2;
  while ( !found && high >= low ){
    if (key == x[middle])
      found = true;
    else if (key < x[middle])
      high = middle - 1;
    else
      low = middle + 1;
      middle = (high + low) / 2;
  }
  return found ? middle : -1 ;
}
```

В процессе работы алгоритма *бинарного поиска* размер фрагмента, где этот поиск должен продолжаться, каждый раз уменьшается примерно в два раза. Это

обеспечивает сложность алгоритма пропорциональную O(log n), где n – количество элементов множества.

Время выполнения алгоритма *бинарного поиска*: если функция имеет вещественный аргумент, найти решение с точностью до $\varepsilon$ можно за время $\log \frac{1}{\varepsilon}$, а если аргумент дискретен, то поиск решения займет 1 + log n времени.

Достоинством данного алгоритма является относительная быстрота выполнения поиска, по сравнению с алгоритмом *последовательного поиска*. Недостаток заключается в том, что *бинарный поиск* может применяться только на упорядоченном множестве.

Ключевые термины

**Бинарный (двоичный, дихотомический) поиск** – это поиск заданного элемента на упорядоченном множестве, осуществляемый путем неоднократного деления этого множества на две части таким образом, что искомый элемент попадает в одну из этих частей.

**Ключ поиска** – это поле записи, по значению которого происходит поиск

**Поиск** – это процесс нахождения конкретной информации в ранее созданном множестве данных.

**Поиск с барьером** – это модификация алгоритма *последовательного поиска*, ускоряющая процесс путем определения граничного элемента.

**Последовательный (линейный) поиск** – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

## Лекция№5

## Тема: Алгоритмы обхода графа.

**Вопросы:**

1. **Поиск в ширину.**

## 2. Поиск в глубину.

Процедура поиска в ширину

Работа всякого алгоритма обхода состоит в последовательном посещении вершин и исследовании ребер. Какие именно действия выполняются при посещении вершины и исследовании ребра - зависит от конкретной задачи, для решения которой производится обход. В любом случае, однако, факт посещения вершины запоминается, так что с момента посещения и до конца работы алгоритма она считается посещенной. Вершину, которая еще не посещена, будем называть *новой*. В результате посещения вершина становится *открытой* и остается такой, пока не будут исследованы все инцидентные ей ребра. После этого она превращается в *закрытую*.

Идея *поиска в ширину* состоит в том, чтобы посещать вершины в порядке их удаленности от некоторой заранее выбранной или указанной стартовой вершины $a$. Иначе говоря, сначала посещается сама вершина $a$, затем все вершины, смежные с $a$, то есть находящиеся от нее на расстоянии $1$, затем вершины, находящиеся от $a$ на расстоянии $2$, и т.д.

Рассмотрим алгоритм *поиска в ширину* с заданной стартовой вершиной $a$. Вначале все вершины помечаются как новые. Первой посещается вершина $a$, она становится единственной открытой вершиной. В дальнейшем каждый очередной шаг начинается с выбора некоторой открытой вершины $x$. Эта вершина становится *активной*. Далее исследуются ребра, инцидентные активной вершине. Если такое ребро соединяет вершину $x$ с новой вершиной $y$, то вершина $y$ посещается и превращается в открытую. Когда все ребра, инцидентные активной вершине, исследованы, она перестает быть активной и становится закрытой. После этого выбирается новая активная вершина, и описанные действия повторяются. Процесс заканчивается, когда множество открытых вершин становится пустым.

Основная особенность *поиска в ширину*, отличающая его от других способов обхода графов, состоит в том, что в качестве активной вершины выбирается та из

открытых, которая была посещена раньше других. Именно этим обеспечивается главное свойство *поиска в ширину*: чем ближе вершина к старту, тем раньше она будет посещена. Для реализации такого правила выбора активной вершины удобно использовать для хранения множества открытых вершин очередь - когда новая вершина становится открытой, она добавляется в конец очереди, а активная выбирается в ее начале. Схематически процесс изменения статуса вершин изображен на рис. 4.1. Черным кружком обозначена активная вершина.


**Рис. 4.1.**

Опишем процедуру *поиска в ширину* (*BFS* - от английского названия этого алгоритма - Breadth First Search) из заданной стартовой вершины $a$. В этом описании $V(x)$ обозначает множество всех вершин, смежных с вершиной $x$, $Q$ - очередь открытых вершин. Предполагается, что при посещении вершины она помечается как посещенная и эта пометка означает, что вершина уже не является новой.

**Procedure BFS(a)**

1. посетить вершину $a$
2. $a \Rightarrow Q$
3. **while** $Q \neq \varnothing$ **do**
4. $x \Leftarrow Q$
5. **for** $y \in V(x)$ **do**
6. исследовать ребро $(x, y)$
7. **if** вершина $y$ новая
8. **then** посетить вершину $y$

9. $y \Rightarrow Q$

Отметим некоторые свойства процедуры *BFS*.

1. *Процедура BFS заканчивает работу после конечного числа шагов*.

   Действительно, при каждом повторении цикла **while** из очереди удаляется одна вершина. Вершина добавляется к очереди только тогда, когда она посещается. Каждая вершина может быть посещена не более одного раза, так как посещаются только новые вершины, а в результате посещения вершина перестает быть новой. Таким образом, число повторений цикла **while** не превосходит числа вершин.

2. *В результате выполнения процедуры BFS будут посещены все вершины из компоненты связности, содержащей вершину a, и только они*.

   Очевидно, что вершина может быть посещена только в том случае, когда существует путь, соединяющий ее с вершиной $a$ (так как посещается всегда вершина, смежная с уже посещенной). То, что каждая такая вершина будет посещена, легко доказывается индукцией по расстоянию от данной вершины до вершины $a$.

3. *Время работы процедуры BFS есть* $O(m)$, где $m$ - число ребер в компоненте связности, содержащей вершину $a$.

Из предыдущих рассуждений видно, что каждая вершина из этой компоненты становится активной точно один раз. Внутренний цикл **for** для активной вершины $x$ выполняется $\deg(x)$ раз. Следовательно, общее число повторений внутреннего цикла будет равно $\sum_{x \in VG} \deg(x) = 2m$.

Итак, процедура *BFS*( $a$ ) производит обход компоненты связности, содержащей вершину $a$. Чтобы перейти к другой компоненте, достаточно выбрать какую-нибудь новую вершину (если такие вершины еще имеются), в качестве стартовой. Пусть $V$ - множество вершин графа. Следующий алгоритм осуществляет полный *обход графа* методом *поиска в ширину*.

**Алгоритм 1**. *Поиск в ширину*.

1. пометить все вершины как новые
2. создать пустую очередь $Q$
3. **for** $a \in V$ **do if** $a$ новая **then** *BFS*( $a$ )

Учитывая, что цикл **for** в строке $3$ повторяется $n$ раз, где $n$ - число вершин графа, получаем общую оценку трудоемкости $O(m+n)$. Необходимо отметить, что эта оценка справедлива в предположении, что время, требуемое для просмотра окрестности вершины, пропорционально степени этой вершины. Это имеет место, например, если граф задан *списками смежности*. Если же граф задан матрицей смежности, то для просмотра окрестности любой вершины будет затрачиваться время, пропорциональное $n$. В этом случае общее время работы алгоритма будет оцениваться как $O(n^2)$. Наибольшее значение величины $m$ при данном $n$ равно $\frac{n(n-1)}{2}$, т.е. имеет порядок $n^2$. Таким образом, трудоемкость алгоритма поиска в ширину при задании графа *списками смежности* не выше, чем при задании матрицей смежности. В целом же первый способ задания предпочтительнее, так как дает выигрыш для графов с небольшим числом ребер.

В качестве простейшего примера применения *поиска в ширину* для графа рассмотрим задачу выявления компонент связности. Допустим, мы хотим получить ответ в виде таблицы, в которой для каждой вершины $x$ указан номер $\mathrm{comp}(x)$ компоненты, которой принадлежит эта вершина. Компоненты будут получать номера в процессе обхода. Для решения этой задачи достаточно ввести переменную $c$ со значением, равным текущему номеру компоненты, и каждый раз при посещении новой вершины $x$ полагать $\mathrm{comp}(x) = c$. Значение $c$ первоначально устанавливается равным $0$ и модифицируется при каждом вызове процедуры *BFS*.

Пусть G – неориентированный связный граф. В процессе поиска в глубину вершинам графа G присваиваются номера (ПГ – номера), а его ребра помечаются. В начале ребра не помечены, вершины не имеют ПГ – номеров. Начинаем с

произвольной вершины V0, присваиваем ей ПГ – номер ПГ(V0) = 1 и выбираем произвольное ребро (V0,W). Ребро (V0,W) помечается как “прямое”, а вершина W получает (из V0) ПГ – номер ПГ(W) = 2. После этого переходим в вершину W. Пусть в результате выполнения нескольких шагов этого процесса пришли в вершину X, и пусть k – последний присвоенный ПГ – номер. Далее действуем в зависимости от ситуации следующим образом.

1. Имеется непомеченное ребро (X,Y). Если у вершины Y уже есть ПГ – номер, то ребро (X,Y) помечаем как “обратное” и продолжаем поиск непомеченного ребра, инцидентного вершине X. Если вершина Y ПГ – номера не имеет, то полагаем ПГ(Y) = k + 1, ребро (X,Y) помечается как “прямое” и переходим в вершину Y. Вершина Y считается получившей свой ПГ – номер из вершины X. На следующем шаге будем просматривать ребра, инцидентные вершине Y.
2. Все ребра, инцидентные вершине X, помечены. В этом случае возвращаемся в вершину, из которой x получила свой ПГ – номер.

Процесс закончится, когда все ребра будут помечены и произойдет возвращение в вершину V0.

Описанный процесс можно реализовать так, чтобы время работы соответствующего алгоритма составляло $O(|EG|+|G|)$ , где |EG| - число ребер графа, |G| - число вершин графа.

Пусть граф G задан списками смежности, т.е. NV – список вершин, инцидентных вершине V, V0 – исходная вершина, с которой начинается поиск. В процессе работы алгоритма каждая вершина графа ровно один раз включается в список Q и исключается из него. Вершина включается в этот список сразу после получения ПГ – номера, и исключается, как только произойдет возвращение из этой вершины. Включение и исключение вершин производится всегда с конца списка, т.е. Q – стек. Результат работы алгоритма – четыре списка ПГ, F, T, B: ПГ(V) – ПГ – номер вершины V; F(V) – имя вершины, из которой вершина V получила свой ПГ – номер; T и B – соответственно списки ориентированных “прямых” и “обратных” ребер графа G. Эти ребра получают ориентацию в процессе работы

алгоритма. Именно, если ребро (X,Y) помечается из вершины X как “прямое”, то в T заносится дуга (X,Y), а если как “обратное”, то эта дуга заносится в B.

Алгоритм поиска в глубину в неориентированном связном графе

1. ПГ(V0) := 1, Q(1):= V0, F(V0) = 0, T := 0, B := 0, k:=1, p:=1 [ k – последний присвоенный ПГ – номер, p – указатель конца стека Q, т.е. Q(p) – вершина стека Q].
2. V := Q(p)
3. Просматривая список NV, найти такую вершину W, что ребро (V,W) не помечено, и перейти к п. 4. Если таких вершин нет, то перейти к п. 5.
4. Если вершина W имеет ПГ – номер, то пометить ребро (V,W) как обратное и занести в список B. Перейти к п. 3 и продолжить просмотр списка NV. Иначе k := k+1, ПГ(W) := k, F(W) := V, ребро (V,W) пометить как “прямое” и занести в список T, p := p+1, Q(p) := W [вершина получила ПГ – номер и занесена в стек Q]. Перейти к п. 2.
5. p := p-1 [вершина v вычеркнута из Q]. Если p = 0, то конец. Иначе перейти к п. 2.

## Лекция№6

## Тема: Генетические и параллельные алгоритмы.

**Вопросы:**

1. **Основные понятия ГА.**
2. **Подходы к решению оптимизационных задач.**

***Генетические Алгоритмы (ГА)*** *– это адаптивные методы функциональной оптимизации, основанные на компьютерном* ***имитационном моделировании биологической эволюции***. Основные принципы ГА были сформулированы Голландом (Holland, 1975), и хорошо описаны во многих работах и на ряде сайтов в Internet.

В настоящее время существует ряд теорий биологической эволюции (Ж.-

Б.Ламарка, П.Тейяра де Шардена, К.Э.Бэра, Л.С.Берга, А.А.Любищева, С.В.Мейена и др.), однако, ни одна из них не считается общепризнанной. Наиболее известной и популярной, конечно, является теория Чарльза Дарвина, которую он представил в работе "Происхождение Видов" в 1859 году.

Эта теория, как и другие, содержит довольно много ***нерешенных проблем***, глубокое рассмотрение которых далеко выходит за рамки данной работы. Здесь мы можем отметить лишь некоторые наиболее известные из них. Как это ни парадоксально, но несмотря на то, что сам Чарльз Дарвин назвал свою работу "Происхождение Видов" но как раз именно *происхождения видов* она и не объясняет. Дело в том, что возникновение нового вида "по алгоритму Дарвина" является крайне маловероятным событием, т.к. для этого требуется случайное возникновение в одной точке пространства и времени сразу не менее 100 особей нового вида, т.е. особей, которые могли бы иметь плодовитое потомство. При меньшем количестве особей вид обречен на вымирание. Поэтому процесс видообразования на основе ***случайных*** мутаций должен был бы занять несуразно много времени (по некоторым оценкам даже в намного раз больше, чем время существования Вселенной). Кроме того, "алгоритм Дарвина" не объясняет явной***системности*** в многообразии возникающих форм, типа *закона гомологичных рядов* Н.И. Вавилова. Поэтому Л.С. Берг предложил очень интересную концепцию ***номогенеза*** – ***закономерной*** или ***направленной*** эволюции живого. В этой концепции предполагается, что филогенез имеет определенное направление и *смена форма является не случайной, а задается некоторым вектором*, природа которого не ясна. Идеи номогенеза глубоко разработал и развил А.А. Любищев, высказавший гипотезу о математических закономерностях, которые определяют многообразие живых форм. Кроме того, Дарвин не смог показать ***механизм наследования***, при котором поддерживается и закрепляется изменчивость. Это было на пятьдесят лет до того, как генетическая теория наследственности начала распространяться по миру, и за тридцать лет до того, как "эволюционный синтез" укрепил связь

между теорией эволюции и молодой генетикой.

Тем ни менее и не смотря на свои недостатки, ***именно теория Дарвина традиционно и моделируется в ГА***, хотя, конечно, это не исключает возможности моделирования и других теорий эволюции в ГА. *Более того, возможно именно такое компьютерное моделирование и сравнение его результатов с картиной реальной эволюции жизни на Земле может быть и сыграет положительную роль в дальнейшей разработке наиболее адекватной теории биологической эволюции.*

Теория Дарвина применима не к отдельным особям, а к ***популяциям*** – большому количеству особей ***одного вида***, т.е. способных давать плодовитое потомство, находящейся в определенной статичной или динамичной внешней среде.

В основе модели эволюции Дарвина лежат ***случайные изменения*** отдельных материальных элементов живого организма при переходе от поколения к поколению. Целесообразные изменения, которые облегчают выживание и производство потомков в данной конкретной внешней среде, сохраняются и передаются потомству, т.е.***наследуются***. Особи, не имеющие соответствующих приспособлений, погибают, не оставив потомства или оставив его меньше, чем приспособленные (считается, что количество потомства пропорционально степени приспособленности). Поэтому в результате ***естественного отбора*** возникает популяция из наиболее приспособленных особей, которая может стать основой нового вида.

***Естественный отбор*** происходит в условиях ***конкуренции*** особей популяции, а иногда и различных видов, друг с другом за различные ***ресурсы***, такие, например, как пища или вода. Кроме того, члены популяции одного вида часто конкурируют за привлечение брачного партнера. Те особи, которые наиболее приспособлены к окружающим условиям, будут иметь относительно больше шансов воспроизвести потомков. Слабо приспособленные особи либо совсем не произведут потомства, либо их потомство будет очень немногочисленным. Это означает, что гены от высоко адаптированных или приспособленных особей будут распространятся в увеличивающемся

количестве потомков на каждом последующем поколении.

Таким образом, по сути дела каждый конкретный ***генетический алгоритм представляют имитационную модель некоторой определенной теории биологической эволюции или ее варианта.*** Вместе с тем необходимо отметить, что сами исследователи биологической эволюции пока еще не до конца определились с критериями и методами определения степени существенности для поддерживаемой ими теории эволюции тех или иных биологических процессов, которые собственно и моделируются в генетических алгоритмах.

### *1.1.6.2. Пример работы простого генетического алгоритма*

На рисунке 85 приведен пример простого генетического алгоритма.

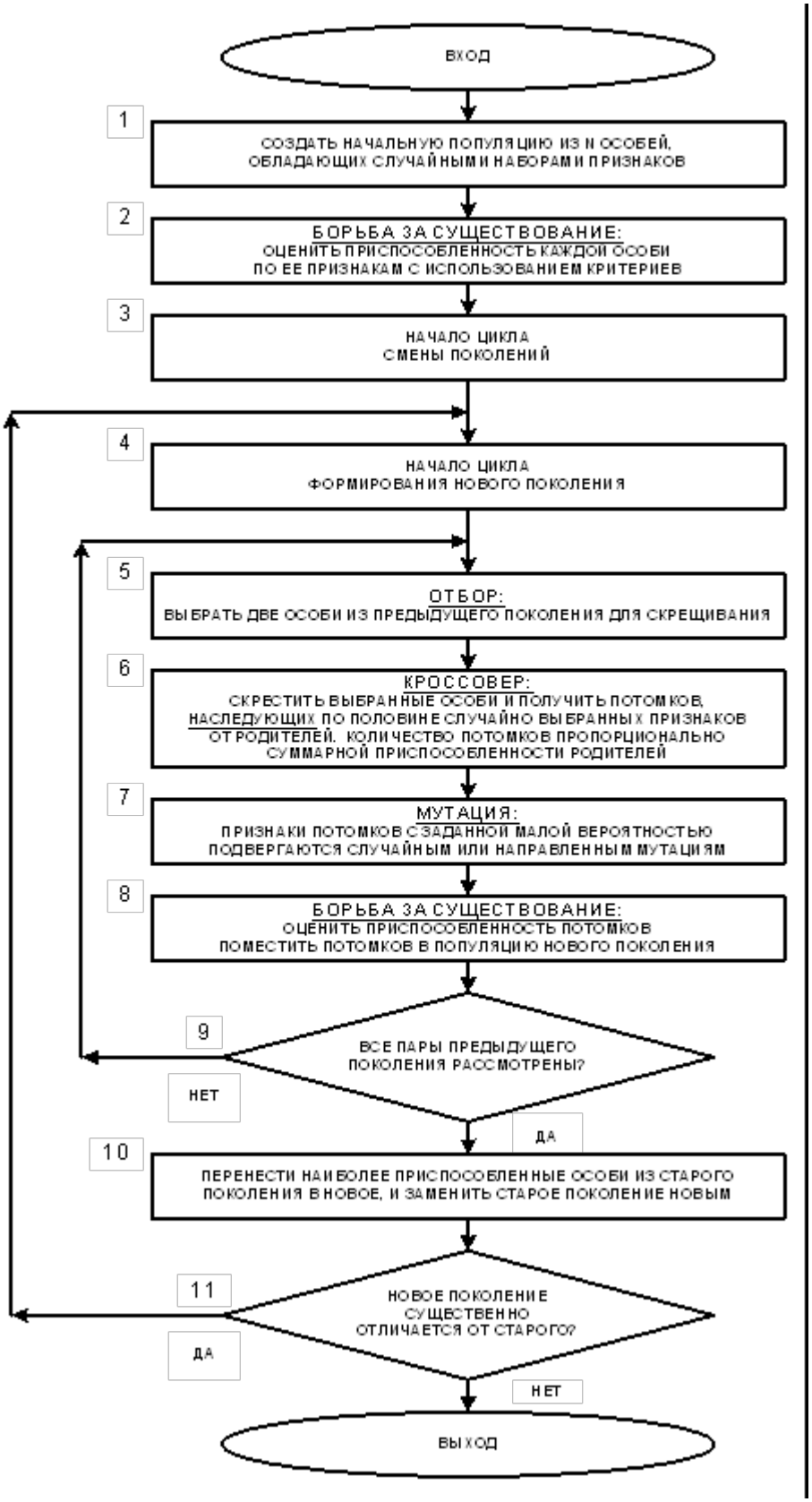



Рисунок 85. Простой генетический алгоритм

Работа ГА представляет собой итерационный процесс, который продолжается до тех пор, пока поколения не перестанут существенно отличаться друг от друга, или не пройдет заданное количество поколений или заданное время. Для каждого поколения реализуются отбор, кроссовер

(скрещивание) и мутация. Рассмотрим этот алгоритм.

***Шаг 1:*** генерируется начальная популяция, состоящая из ***N*** особей со случайными наборами признаков.

***Шаг 2 (борьба за существование):*** вычисляется *абсолютная приспособленность* каждой особи популяции к условиям среды *f(i)* и суммарная приспособленность особей популяции, характеризующая приспособленность всей популяции. Затем при ***пропорциональном отборе*** для каждой особи вычисляется ее *относительный вклад в суммарную приспособленность популяции* $P_s(i)$, т.е. отношение ее абсолютной приспособленности *f(i)* к суммарной приспособленности всех особей популяции (3):

$$P_s(i) = \frac{f(i)}{\sum_{i=1}^{N} f(i)} \qquad (3)$$

В выражении (3) сразу обращает на себя внимание возможность сравнения абсолютной приспособленности *i-й* особи *f(i)* не с суммарной приспособленностью всех особей популяции, а со средней абсолютной приспособленностью особи популяции (4):

$$\bar{f} = \frac{1}{N}\sum_{i=1}^{N} f(i) \qquad (4)$$

Тогда получим (5):

$$P(i) = \frac{f(i)}{\bar{f}} = \frac{f(i)}{\frac{1}{N}\sum_{i=1}^{N} f(i)} \qquad (5)$$

Если взять логарифм по основанию 2 от выражения (5), то получим ***количество информации, содержащееся в признаках особи о том, что она выживет и даст потомство*** (6).

$$I(i) = Log_2 \frac{f(i)}{\bar{f}} \qquad (6)$$

Необходимо отметить, что эта формула совпадает с формулой для семантического количества информации Харкевича, если целью считать ***индивидуальное выживание и продолжение рода***. Это значит, что даже чисто формально ***приспособленность особи представляет собой количество информации, содержащееся в ее фенотипе о продолжении ее генотипа в последующих поколениях.***

Поскольку количество потомства особи пропорционально ее приспособленности, то естественно считать, что *если это количество информации:*

– *положительно,* то данная особь выживает и дает потомство, численность которого пропорциональна этому количеству информации;

– *равно нулю,* то особь доживает до половозрелого возраста, но потомства не дает (его численность равна нулю);

– *меньше нуля,* то особь погибает до достижения половозрелого возраста.

Таким образом, можно сделать фундаментальный вывод, имеющий даже мировоззренческое звучание, о том, что ***естественный отбор представляет собой процесс генерации и накопления информации о выживании и продолжении рода в ряде поколений популяции, как системы.***

Это накопление информации происходит на различных уровнях иерархии ***популяции, как системы***, включающей:

– элементы системы: отдельные особи;

– взаимосвязи между элементами: отношения между особями в популяции, обеспечивающие передачу последующим поколениям максимального количества информации об их выживании и продолжении рода (путем скрещивания наиболее приспособленных особей и наследования рациональных приобретений);

– цель системы: сохранение и развитие популяции, реализуется через цели особей: индивидуальное выживание и продолжение рода.

Фенотип соответствует генотипу и представляет собой его внешнее проявление в признаках особи. Особь взаимодействует с окружающей средой и другими особями в соответствии со своим фенотипом. В случае, если это взаимодействие удачно, то особь передает генетическую информацию, определяющую фенотип, последующим поколениям.

***Шаг 3:*** начало цикла смены поколений.

***Шаг 4:*** начало цикла формирования нового поколения.

***Шаг 5 (отбор)***: осуществляется *пропорциональный* ***отбор*** особей,

которые могут участвовать в продолжении рода. Отбираются только те особи популяции, у которых количество информации в фенотипе и генотипе о выживании и продолжении рода положительно, причем вероятность выбора пропорциональна этому количеству информации.

***Шаг 6 (кроссовер)***: отобранные для продолжения рода на предыдущем шаге особи с заданной вероятностью $P_c$ подвергаются ***скрещиванию*** или ***кроссоверу***(***рекомбинации***).

Если кроссовер происходит, то потомки получают по половине случайным образом определенных признаков от каждого из родителей. Численность потомства пропорциональна суммарной приспособленности родителей. В некоторых вариантах ГА потомки после своего появления заменяют собой родителей и переходят к мутации.

Если кроссовер не происходит, то исходные особи – несостоявшиеся родители, переходят на стадию мутации.

***Шаг 7 (мутация):*** выполняются операторы ***мутации***. При этом признаки потомков с вероятностью $P_m$ случайным образом изменяются на другие. Отметим, что использование механизма случайных мутаций роднит генетические алгоритмы с таким широко известным методом имитационного моделирования, как ***метод Монте-Карло***.

***Шаг 8 (борьба за существование):*** оценивается приспособленность потомков (по тому же алгоритму, что и на шаге 2).

***Шаг 9:*** проверяется, все ли отобранные особи дали потомство.

Если нет, то происходит переход на шаг 5 и продолжается формирование нового поколения, иначе – переход на следующий шаг 10.

***Шаг 10:*** происходит смена поколений:

– потомки помещаются в новое поколение;

– наиболее приспособленные особи из старого поколения переносятся в новое, причем для каждой из них это возможно не более заданного количества раз;

– полученная новая популяция замещает собой старую.

***Шаг 11:*** проверяется выполнение условия останова генетического

алгоритма. ***Выход*** из генетического алгоритма происходит либо тогда, когда новые поколения перестают существенно отличаться от предыдущих, т.е., как говорят, "алгоритм сходится", либо когда пройдено заданное количество поколений или заданное время работы алгоритма (чтобы не было "зацикливания" и динамического зависания в случае, когда решение не может быть найдено в заданное время ).

*Если ГА сошелся, то это означает, что решение найдено*, т.е. получено поколение, идеально приспособленное к условиям данной фиксированной среды обитания.

Иначе – переход на шаг 4 – начало формирования нового поколения.

*В реальной биологической эволюции этим дело не ограничивается, т.к. любая популяция кроме освоения некоторой экологической ниши пытается также выйти за ее пределы освоить и другие ниши, как правило "смежные"*. Именно за счет этих процессов жизнь вышла из моря на сушу, проникла в воздушное пространство и поверхностный слой почвы, а сейчас осваивает космическое пространство.

Конечно, реальные генетические алгоритмы, на которых проводятся научные исследования, чаще всего мало похожи на приведенный пример. Исследователи экспериментируют с различными параметрами генетических алгоритмов, например: способами отбора особей для скрещивания; критериями приспособленности и жесткостью влияния факторов среды; способами выбора признаков, передающихся от родителей потомкам (рецессивные и не рецессивные гены и т.д.); интенсивностью, видом случайного распределения и направленностью мутаций; различными подходами к воспроизводству и отбору.

Поэтому под термином "генетические алгоритмы" по сути дела надо понимать не одну модель, а довольно широкий класс алгоритмов, подчас мало похожих друг на друга.

В настоящее время рассматривается много различных операторов отбора, кроссовера и мутации: **турнирный отбор** (Brindle, 1981; Goldberg и Deb, 1991), реализует *n*турниров, чтобы выбрать *n* особей, при этом каждый

турнир построен на выборке $k$ элементов из популяции, и выбора лучшей особи среди них (наиболее распространен турнирный отбор с $k$=2); **элитный отбор** (De Jong, 1975) гарантируют, что при отборе обязательно будут выживать лучший или лучшие члены популяции совокупности (наиболее распространена процедура обязательного сохранения только одной лучшей особи, если она не прошла как другие через процесс отбора, кроссовера и мутации);**двухточечный кроссовер** (Cavicchio, 1970; Goldberg, 1989c) и равномерный кроссовер (Syswerda, 1989) отличаются способами наследования потомками признаков родителей.

Не смотря на то, что модели биологической эволюции, реализуемые в ГА, обычно сильно упрощены по сравнению с природным оригиналом, тем ни менее ГА являются мощным средством, которое может с успехом применяться для решения широкого класса прикладных задач, включая те, которые трудно, а иногда и вовсе невозможно, решить другими методами.

*1.1.6.3. Достоинства и недостатки генетических алгоритмов*

Однако, ГА не гарантирует обнаружения глобального решения за приемлемое время. ГА не гарантируют и того, что найденное решение будет оптимальным решением. Тем ни менее они применимы для ***поиска*** "достаточно хорошего" решения задачи за "достаточно короткое время". ГА представляют собой разновидность алгоритмов поиска и имеют преимущества перед другими алгоритмами при очень больших размерностях задач и отсутствия упорядоченности в исходных данных, когда альтернативой им является метод полного перебора вариантов.

В случаях, когда задача может быть решена специально разработанным для нее методом, практически всегда такие методы будут эффективнее ГА как по быстродействию, так и по точности найденных решений.

***Главным же достоинством ГА является то, что они могут применяться для решения сложных неформализованных задач, для которых не разработано специальных методов, т.е. ГА обеспечивают решение проблем.*** Но даже в тех случаях, для которых хорошо работают

существующие методики, можно достигнуть интересных результатов сочетая их с ГА.

*Примеры применения генетических алгоритмов*

В 1994 году Эндрю Кин из университета Саутгемптона использовал генетический алгоритм в дизайне космических кораблей. За основу была взята модель опоры космической станции, спроектированной в NASA из которой после смены 15 поколений, включавших 4.500 вариантов дизайна, получилась модель, превосходящая по тестам тот вариант, что разработали люди.

Аналогичный генетический алгоритм был использован NASA при разработке антенны для спутника.

Джон Коза из Стэнфорда разработал технологию генетического программирования, в которой результатом эволюции становятся не отдельные числовые параметры "особей", а целые имитационные программы, которые являются виртуальными аналогами реальных устройств. Эта технология позволила компании Genetic Programming повторить 15 человеческих изобретений, 6 из которых были запатентованы после 2000 года, то есть представляют собой самые передовые достижения, а один из контроллеров, "выведенных" в GP, даже превосходит аналогичную человеческую разработку.

Сейчас плоды электронной эволюции можно найти в самых разных сферах: от двигателя самолета Boeing 777 до новых антибиотиков.

Генетические алгоритмы представляют собой компьютерное моделирование эволюции. Материальное воплощение сконструированных таким образом систем до сих пор была невозможна без участия человека. Однако интенсивно ведутся работы, результатом которых является уменьшение зависимости машинной эволюции от человека. Эти работы ведутся по двум основным направлениям:

1. Естественный отбор, моделируемый ГА, переносится из виртуального мира в реальный, например, проводятся эксперименты по реальным битвам роботов на выживание.

2. Интеллектуальные системы, основанные на ГА, конструируют

роботов, которые в принципе могут быть изготовлены на автоматизированных заводах без участия человека.

**Пример воплощения ГА в реальной битве роботов на выживание:** в 2002 году в британском центре Magna открылся павильон Live Robots, где боролись за выживание 12 роботов двух видов: "гелиофаги", способные добывать электроэнергию с использованием солнечных батарей; "хищники", которые могли получать электроэнергию только от гелиофагов. Выжившие роботы загружали свои "гены" в погибших и, таким образом, образовывали новые поколения. Те хищники, которые забирали всю энергию у гелиофагов, теряли источник питания и погибали, не передавая свою тактику потомкам, поступавшие же "более разумно" продолжили свой род. В результате возникла равновесная сбалансированная искусственная экосистема с двумя популяциями.

**Пример конструирования роботов роботами:** в Brandeis University была создана программа Golem, которая сама конструировала роботов. В программу была база деталей, а также механизм мутаций и функция пригодности для "отсеивания" неудачников – тех, кто не научился двигаться. После 600 поколений за несколько дней программа получила модели трех ползающих роботов. Показательно, что роботы оказались симметричными, хотя симметрия никак не была явно прописана в правилах эволюции и исходных данных. Это означает, что она появилась в ходе моделирования машинной эволюции как полезная черта, позволяющая двигаться прямолинейно.

## Лекция№7

**Тема: Генетический алгоритм. Свойства.**

**Вопросы:**

**1. Понятие популяции. Свойства. Ген.**
**2. Фенотип. Генотип. Фитнес. Мутация. Кроссовер.**

## Основные понятия генетических алгоритмов

При описании генетических алгоритмов используются определения, заимствованные из генетики, например речь, идет о *популяции особей*, а в качестве понятий применяются *ген, хромосома, генотип, фенотип, аллель.* Также используются соответствующие этим терминам определения из технического лексикона, в частности, *цепь*, *двоичная последовательность, структура.*

**Популяция** – это конечное множество особей

**Особи**, входящие в популяцию, в генетических алгоритмах представляются хромосомами с закодированным в них множеством параметров задачи, т.е. решений, которые иначе называются *точками в пространстве поиска (search points).* В некоторых работах особи называются*организациями.*

**Хромосомы** (другие названия – *цепочки* или *кодовые последовательности*) – это упорядоченные последовательности *генов.*

**Ген** (также называемый *свойством*, *знаком* или *детектором*) – это атомарный элемент *генотипа*, в частности, хромосомы.

Генотип или структура – это набор хромосом данной особи. Следовательно, особями популяции могут быть генотипы, либо единичные хромосомы (в довольно распространенном случае, когда генотип состоит из одной хромосомы).

**Фенотип** – это набор значений, соотвестсвующих данному генотипу, т.е. *декодированная структура* или *множество параметров задачи*(*решение, точка пространства поиска*).

**Аллель** – это значение конкретного гена, также определяемое как *значение свойства* или *вариант свойства*.

**Локус** или позиция указывает место размещение данного гена в хромосоме (цепочке). Множество позиций генов – это *локи*.

Очень важным понятием в генетических алгоритмах считается *функция приспособленности* (*fitness function*), иначе называемая *функцией оценки*. Она представляет меру приспособленности данной особи в популяции. Эта функция играет важнейшую роль, поскольку позволяет оценить степень приспособленности конкретных особей в популяции и выбрать из них наиболее приспособленные (т.е. имеющие наибольшие значения функции приспособленности) в соответствии с эволюционным принципом выживания «сильнейших» (лучше всего приспособившихся). Функция приспособленности также получила свое название непосредственно из генетики. Она оказывает сильное влияние на функционирование генетических алгоритмов и должна иметь точное и корректное определение.

В задачах оптимизации функция приспособленности, как правило оптимизируется (точнее говоря максимизируется) и называется *целевой функцией*. В задачах минимизации целевая функция преобразуется, и проблема сводится к максимизации. В теории управления функция приспособленности может принимать вид *функций погрешности*, а в теории игр *стоимостной функции*. На каждой итерации генетического алгоритма приспособленность каждой особи данной популяции оценивается при помощи функции приспособленности, и на этой основе создается следующая популяция особей, составляющая множество

потенциальных решений проблемы, например, задачи оптимизации. Очередная популяция в генетическом алгоритме называется поколением, а к вновь создаваемой популяции особей применяется термин «новое поколение» или «поколение потомков».

^ **Пример - 24.**

Рассмотрим функцию

$F(x)=2x^2+1$ (1)

И допустим, что *x* принимает целые значения из интервала от 0 до 15. Задача оптимизации этой функции заключается в перемещении по пространству, состоящему из 16 точек со значениями 0, 1, …, 15 для обнаружения той точки в которой функция принимает максимальное (или минимальное) значение.

В этом случае в качестве *параметра задачи* выступает переменная *x*. Множество {0, 1, …, 15} составляет *пространство поиска* и одновременно – множество потенциальных решений задачи. Каждое из 16 чисел, принадлежащих к этому множеству, называется *точкой пространства поиска, решением, значением параметра, фенотипом.* Следует отметить, что решение, оптимизирующее функцию, называется *наилучшим* или*оптимальным* решением. Значения параметра *x* от 0 до 15 можно закодировать следующим образом:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

Это широко известный способ двоичного кодирования, связанный с записью десятичных цифр в двоичной системе. Представленные *кодовые последовательности* также называются *цепями* или *хромосомам*. В рассматриваемом примере они выступают в роли *генотипов*. Каждая из хромосом состоит из 4 генов (иначе можно сказать, что двоичные последовательности состоя т из 4 битов). Значение гена в конкретной позиции называется аллелью, принимающей в данном случае значение 0 или 1. *Популяция* состоит из *особей* выбираемых среди 16 хромосом. Примером популяции с численностью равной 6 может быть, например, множество хромосом{0010, 0101, 0111, 1001, 1100, 1110}, представляющих собой закодированную форму следующих фенотипов: {2, 5, 7, 9, 12, 14}.

^ *Функция приспособленности* в этом примере задается выражением (1). Приспособленность отдельных хромосом в популяции определяется значением этой функции для значения x, соответствующих этим хромосом, т.е. для фенотипов, соответствующих определенным *генотипам* .

## ^ 2 Классический генетический алгоритм

Основной (классический) генетический алгоритм (также называемый элементарным или простым генетическим алгоритмом) состоит из следующих шагов:

1) инициализация, или выбор исходной популяции хромосом;

2) оценка приспособленности хромосом в популяции;

3) проверка условий остановки алгоритма;


4) селекция хромосом;

5) применение генетических операторов;

6) формирование новой популяции;

7) выбор «наилучшей» хромосомы.

Блок-схема основного генетического алгоритма изображена на рисунке 1. Рассмотрим конкретные этапы этого алгоритма более подробно.

Рисунок 1 - Блок-схема генетического алгоритма

**Инициализация,** т.е. формирование исходной популяции, заключается в случайном выборе заданного количества хромосом (особей), представленных двоичными последовательностями фиксированной длины.

^ **Оценивание приспособленности** хромосом в популяции состоит в расчете функции приспособленности для каждой хромосомы этой популяции. Чем больше значение этой функции, тем выше «качество» хромосомы.

Форма функции приспособленности зависит от характера решаемой задачи. Предполагается, что функция приспособленности всегда принимает неотрицательное значение и, кроме того, что для решения оптимизационной требуется максимизировать эту функцию. Если исходная форма функции приспособленности не удовлетворяет этим условиям, то выполняется соответствующее преобразование (например, задачу минимизации функции можно легко свести к задаче максимизации).

^ **Проверка условия остановки алгоритма.** Определение условия установки генетического алгоритма зависит от его конкретного применения. В оптимизационных задачах, если известно максимальное (или минимальное) значение функции приспособленности, то остановка алгоритма может произойти после достижение ожидаемого оптимального значения, возможно-с заданной точностью. Остановка алгоритма также может произойти в случае, когда его выполнение не приводит к улучшению уже достигнутого значения. Алгоритм может быть остановлен по истечении определенного времени выполнения либо после выполнения заданного количества итераций. Если условие остановки выполнено, то производится переход к завершающему этапу выбора «наилучшей» хромосомы. В противном случае на следующем шаге выполняется селекция.

^ **Селекция хромосом** заключается в выборе (по рассчитанным на втором этапе значениям функции приспособленности) тех хромосом, которые будут участвовать в создании потомков для следующей популяции, т.е. для очередного поколения. Такой выбор производится согласно принципу естественного отбора, по которому наибольшие шансы на участие в создании новых особей имеют хромосомы с наибольшими значениями функции приспособленности. Существуют различные методы селекции. Наиболее популярным считается так называемый метод методы рулетки (roulette wheel selection), который свое название получил по аналогии с известной азартной игрой. Каждой хромосоме может быть сопоставлен сектор колеса рулетки, величина которого устанавливается пропорциональной значению функции приспособленности данной хромосомы. Поэтому чем больше значение

функции приспособленности, тем больше сектор на колесе рулетки. Все колесо рулетки соответствует сумме значений функции приспособленности всех хромосом рассматриваемой популяции. Каждой хромосоме, обозначаемой $ch_i$ для $i=1,2,\ldots,N$(где $N$ обозначает численность популяции) соответствует сектор колеса v($ch_i$), выраженной в процентах согласно формуле

$$v(ch_i) = p_s(ch_i)100\%, \quad (2)$$

где

$$p_s(ch_i) = \frac{F(ch_i)}{\sum_{i=1}^{N} F(ch_i)} \quad (3)$$

причем $F(ch_i)$ – значение функции приспособленности хромосомы $ch_i$, а $p_s(ch_i)$ – вероятность селекции хромосомы $ch_i$. Селекция хромосомы может быть представлена как результат поворота колеса рулетки, поскольку «выигравшая» (т.е. выбранная) хромосома относится к выпавшему сектору этого колеса. Очевидно, что чем больше сектор, тем больше вероятность «победы» соответствующей хромосомы. Поэтому вероятность выбора данной хромосомы оказывается пропорциональной значению её функции приспособленности. Если всю окружность рулетки представить в виде цифрового интервала [0,100], то выбор хромосомы можно отождествить с выбором числа из интервала [a,b], где a и b обозначают соответственно начало и окончание фрагмента окружности, соответствующего этому сектору колеса; очевидно, что $0 \le a < b \le 100$. В этом случае выбор с помощью колеса рулетки сводится к выбору числа из интервала [0,100], которое соответствует конкретной точке на окружности колеса.

В результате процесса селекции создается *родительская популяция*, также

называемая *родительским пулом (mating pool)* с численностью $N$, равной численности текущей популяции.

**Применение генетических операторов** к хромосомам, отобранным с помощью селекции, приводит к формированию новой популяции потомков от созданной на предыдущем шаге родительской популяции.

В классическом генетическом алгоритме применяются два основных оператора: *оператор скрещивания (crossover)* и *оператор мутации (mutation).* Однако следует отметить, что оператор мутации играет явно второстепенную роль по сравнению с оператором скрещивания. Это означает, что скрещивание в классическом генетическом алгоритме производится практически всегда, тогда как мутация – достаточно редко. Вероятность скрещивания, как правило, достаточно велика (обычно $0{,}5 \leq p_c \leq 1$), тогда как вероятность мутации устанавливается весьма малой (чаще всего $0 \leq p_m \leq 0{,}1$). Это следует из аналогии с миром живых организмов, где мутации происходят чрезвычайно редко.

В генетическом алгоритме мутация хромосом может выполнятся на популяции родителей перед скрещиванием либо на популяции потомков, образованных в результате скрещивания.

^ **Оператор скрещивания.** На первом этапе скрещивания выбираются пары хромосом из родительской популяции (родительского пула). Это временная популяция, состоящая из хромосом, отобранных в результате селекции и предназначенных для дальнейших преобразований операторами скрещивания и мутации с целью формирования новой популяции потомков. На данном этапе хромосомы из родительской популяции объединяются в пары. Это производится случайным способом в соответствии с вероятностью скрещивания $p_c$. Далее для каждой пары отобранных таким образом родителей разыгрывается позиция гена *(локус)* в хромосоме, определяющая так называемую *точку скрещивания.* Если хромосома каждого из родителей состоит из $L$ генов, то

очевидно, что точка скрещивания $l_k$ представляет собой натуральное число, меньшее $L$. Поэтому фиксация точки скрещивания сводится к случайному выбору пары из интервала [1, $L$-1]. В результате скрещивания пары родительских хромосом получается следующая пара потомков:

1) потомок, хромосома которого на позициях от 1 до $l_k$ состоит из генов первого родителя, а на позиция от $l_k$ + 1 до $L$ – из генов второго родителя;

2)потомок, хромосома которого на позициях от 1 до $l_k$ состоит из генов второго родителя, а на позиция от $l_k$ + 1 до $L$ – из генов первого родителя.

**Оператор мутации** с вероятностью $p_m$ изменяет значение гена в хромосоме на противоположное (т.е. с 0 на 1 или обратно). Например, если в хромосоме [10011010010] мутации подвергается ген на позиции 7, то его значение, равное 1, изменяется на 0, что приводит к образованию хромосомы [10011000110]. Как уже упоминалось выше, вероятность мутации обычно очень мала, и именно от нее зависит, будет данный ген мутировать или нет. Вероятность $p_m$ мутации может эмулироваться, например случайным выбором числа из интервала [0,1] для каждого гена и отбором для выполнения этой операции тех генов, для которых разыгранное число оказывается меньшим или равным значению $p_m$.

^ **Формирование новой популяции.** Хромосомы, полученные в результате применения генетических операторов к хромосомам временной родительской популяции, включаются в состав новой популяции. Она становится так называемой текущей популяцией для данной итерации генетического алгоритма. На каждой очередной итерации рассчитываются значения функции приспособленности для всех хромосом этой популяции, после чего проверяется условие остановки алгоритма и либо фиксируется результат в виде хромосомы с наибольшим значением функции приспособленности, либо осуществляется переход к следующему шагу генетического алгоритма, т.е. к селекции. В классическом генетическом алгоритме вся предшествующая популяция хромосом

замещается новой популяцией потомков, имеющей ту же численность.

^ **Выбор «наилучшей» хромосомы.** Если условие остановки алгоритма выполнено, то следует вывести результат работы, т.е. представить искомое решение задачи. Лучшим решением считается хромосома с наибольшим значением функции приспособленности.

Генетические алгоритмы унаследовали свойства естественного эволюционного процесса, состоящие в генетических изменениях популяции организмов с течением времени.

Главный фактор эволюции – это естественный отбор (т.е. природная селекция), который приводит к тому, что среди генетически различающихся особей одной и той же популяции выживают только наиболее приспособленные к окружающей среде. В генетических алгоритмах также выделяется этап селекции, на котором из текущей популяции выбираются и включаются в родительскую популяцию особи, имеющие наибольшие значения функции приспособленности. На следующем этапе, который иногда называется *эволюцией*, применяются генетические операторы скрещивания и мутации, выполняющие рекомбинацию генов в хромосомах.

Операция скрещивания заключается в обмене фрагментами цепочек между двумя родительскими хромосомами. Пары родителей для скрещивания выбираются из родительского пула случайным образом так, чтобы вероятность выбора конкретной хромосомы для скрещивания была равна вероятности $p_c$. Например, если в качестве родителей случайным образом выбираются две хромосомы из родительской популяции численностью *N,* то $p_c=2/N$. Аналогично, если из родительской популяции численностью *N* выбирается *2z* хромосом ($z \le N/2$), которые образуют *z*пар родителей, то $p_c=2z/N$. Если все хромосомы текущей популяции объединены в пары до скрещивания, то $p_c=1$. После операции скрещивания родители в родительской популяции замещаются их потомками.

Операция мутации изменяет значения генов в хромосомах с заданной вероятностью $p_m$ способом, представленным при описании соответствующего оператора. Это приводит к инвертированию значений отобранных генов с 0 на 1 и обратно. Значение $p_m$, как правило, очень мало, поэтому мутации подвергается лишь небольшое количество генов. Скрещивание – это ключевой оператор генетических алгоритмов, определяющий их возможности и эффективность. Мутация играет наиболее ограниченную роль. Она вводит в популяцию некоторое разнообразие и предупреждает потери, которые могли бы произойти вследствие исключения какого-нибудь значимого гена в результате скрещивания.

Основной (классический) генетический алгоритм известен в литературе в качестве инструмента, в котором выделяется три вида операций:*репродукции, скрещивания и мутации*. Термины *селекция* и *репродукция* в данном тексте используются в качестве синонимов. При этом репродукция в данном случае связывается скорее с созданием копии хромосом родительского пула, тогда как более распространенное содержание этого понятия обозначает процесс формирования новых особей, происходящих от конкретных родителей.

## ^ 3 Выполнения классического генетического алгоритма

Рассмотрим выполнение описанного в предыдущем подразделе классического генетического алгоритма на простом примере, состоящим в нахождении хромосом с максимальным количеством единиц.

**Пример**

Допустим, что хромосомы состоят из 12 генов, а популяция насчитывает 8 хромосом. Понятно, что наилучший будет хромосома, состоящая из 12 единиц.

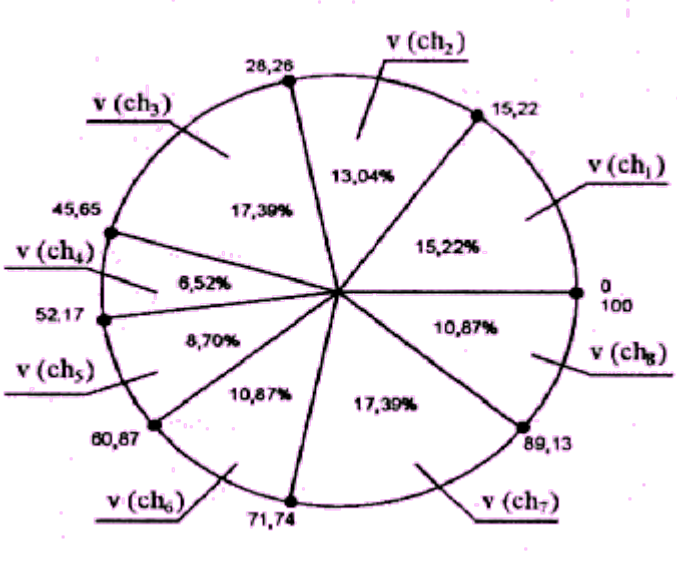
Посмотрим, как протекает процесс решения этой весьма тривиальной задачи с помощью генетического алгоритма.

^ **Инициализация, или выбор исходной популяции хромосом.** Необходимо случайным образом сгенерировать 8 двоичных последовательностей длиной 12 битов. Это можно достигнуть, например, подбрасыванием монеты (96 раз, при выпадении «орла» приписываем значение 1, а в случае «решки» - 0). Таким образом можно сформировать исходную популяцию

| | |
|---|---|
| $ch_1$=[111001100101] | $ch_5$=[010001100100] |
| $ch_2$=[001100111010] | $ch_6$=[010011000101] |
| $ch_3$=[011101110011] | $ch_7$=[101011011011] |
| $ch_4$=[001000101000] | $ch_8$=[000010111100] |

^ **Оценка приспособленности хромосом в популяции.** В рассматриваемом упрощенном примере решается задача нахождения такой хромосомы, которая содержит наибольшее количество единиц. Поэтому функция приспособленности определяет количество единиц в хромосоме. Обозначим функцию приспособленности символом *F*. Тогда ее значение для каждой хромосомы из исходной популяции будут такие:

| | |
|---|---|
| $F(ch_1)=7$ | $F(ch_5)=4$ |
| $F(ch_2)=6$ | $F(ch_6)=5$ |
| $F(ch_3)=8$ | $F(ch_7)=8$ |
| $F(ch_4)=3$ | $F(ch_8)=5$ |




Хромосомы $ch_3$ и $ch_7$ характеризуются наибольшими значениями функции принадлежности. В этой популяции они считаются наилучшими кандидатами на решение задачи. Если в соответствии с блок-схемой генетического алгоритма условие остановки алгоритма не выполняется, то на следующем шаге производится селекция хромосом из текущей популяции.

^ **Селекция хромосом.** Селекция производится методом рулетки. На основании формул (2) и (3) для каждой из 8 хромосом текущей популяции (в нашем случае – исходной популяции, для которой N = 8) получаем секторы колеса рулетки, выраженные в процентах (рисунок 2)

| | |
|---|---|
| $v(ch_1)=15.22$ | $v(ch_5)=8.7$ |
| $v(ch_2)=13.04$ | $v(ch_6)=10.87$ |
| $v(ch_3)=17.39$ | $v(ch_7)=17.39$ |
| $v(ch_4)=6.52$ | $v(ch_8)=10.87$ |

Рисунок 2 - Колесо рулетки для селекции

Розыгрыш с помощью колеса рулетки сводится к случайному выбору числа из интервала [0,100], указывающего на соответствующий сектор на колесе, т. е. на

конкретную хромосому. Допустим, что разыграны следующие 8 чисел:

79 4 9 74 44 86 48 23

Это означает выбор хромосом

$ch_7$ $ch_3$ $ch_1$ $ch_7$ $ch_3$ $ch_7$ $ch_4$ $ch_2$

Как видно, хромосома $ch_7$ была выбрана трижды, а хромосома $ch_3$ – дважды. Заметим, что именно эти хромосомы имеют наибольшее значение функции приспособленности. Однако выбрана и хромосома $ch_4$ с наименьшим значением функции приспособленности.

Первая пара родителей: Первая пара потомков:

[001100111010]
[101011011011] ◊скрещивание [001111011011]
[101000111010]

$l_k$=4

Вторая пара родителей: Вторая пара потомков:

[111001100101]
[101011011011] ◊скрещивание [111011011011]
[101001100101]

$l_k$=3

Третья пара родителей: Третья пара потомков:

[011101110011]
[001000101000] ◊скрещивание [011101110010]
[001000101001]

$l_k$=11

Четвертая пара родителей: Четвертая пара потомков:

[011101110011]
[101011011011] ◊скрещивание [011101011011]
[101011110011]

Для первой пары случайным образом выбрана точка скрещивания $l_k$=4, для второй $l_{k=3}$, для третьей $l_k$=11, для четвертой $l_k$=5. При этом процесс скрещивания протекает следующим образом:

В результате выполнения оператора скрещивания получаются четыре пары потомков.

Если бы при случайном подборе пар хромосом для скрещивания были объединены, например,*ch₃* с *ch₃* и *ch₄* с *ch₇* вместо *ch₃* с *ch₄* и *ch₃* с *ch₇*, а другие пары остались без изменения, то скрещивание *ch₃* с $ch_3$ дало бы две такие же хромосомы независимо от разыгранной точки скрещивания. Это означало бы получение двух потомков, идентичных своим родителям. Заметим, что такая ситуация наиболее вероятна для хромосом с наибольшим значением функции приспособленности, т.е. именно такие хромосомы получают наибольшие шансы на переход в новую популяцию.

^ **Формирование новой популяции.** После выполнения операции скрещивания мы получаем следующую популяцию потомков:

$Ch_1$ = [001111011011] $Ch_5$ = [011101110010]

$Ch_2$ = [101000111010] $Ch_6$ = [001000101001]

$Ch_3$ = [111011011011] $Ch_7$ = [011101011011]

$Ch_4$ = [101001100101] $Ch_8$ = [101011110011]

Для отличия от хромосом предыдущей популяции обозначения вновь сформированных хромосом начинаются с заглавной буквы *C*.

Согласно блок-схеме генетического алгоритма производят возврат ко второму этапу, т.е. к оценке приспособленности хромосом из вновь сформированной популяции, которая становится текущей. Значения функций приспособленности хромосом этой популяции составляют:

$F(Ch_1)=8$ $F(Ch_5)=7$

$F(Ch_2)=6$ $F(Ch_6)=4$

$F(Ch_3)=9$ $F(Ch_7)=8$

$F(Ch_4)=6$ $F(Ch_8)=8$

Заметно, что популяция потомков характеризуется гораздо более высоким значением функции приспособленности, чем популяция родителей. Обратим внимание, что в результате скрещивания получена хромосома $Ch_3$ с наибольшим значением функции приспособленности, которым не обладала ни одна хромосома из родительской популяции. Однако могло произойти и обратное, поскольку после скрещивания на первой итерации хромосома, которая в родительской популяции характеризовалась наибольшим значением функции приспособленности, могла просто «потеряться». Помимо этого «средняя» приспособленность новой популяции все равно оказалась бы выше предыдущей, а хромосомы с большими значениями функции приспособленности имели бы шанс появится в следующих поколениях.

**Лекция№8**

**Тема: Генетический алгоритм. Модель Холланда.**

**Вопросы:**

**1. Алгоритмизация модели Холланда.**

**2. Основные свойства модели. Изучение останова алгоритма.**

***1. Canonical GA (J. Holland)***

Данная модель алгоритма является классической. Она была предложена Джоном Холландом в его знаменитой работе "Адаптация в природных и исусственных средах" (1975). Часто можно встретить описание *простого ГА* (Simple GA, D. Goldberg), он отличается от канонического тем, что использует либо рулеточный, либо турнирный отбор. Модель канонического ГА имеет следующие характеристики:

- Фиксированный размер популяции.
- Фиксированная разрядность генов.
- Пропорциональный отбор.
- Особи для скрещивания выбираются случайным образом.
- Одноточечный кроссовер и одноточечная мутация.
- Следующее поколение формируется из потомков текущего поколения без "элитизма". Потомки занимают места своих родителей.

В терминах теории расписаний [2] сформулируем минимаксную задачу, алгоритмы решений которой исследуются в данной работе. Имеется вычислительная система (ВС), состоящая из $m$ несвязанных идентичных процессоров (приборов, машин и т.п.) $P=\{p_1, p_2, \ldots, p_m\}$. На обслуживание в ВС поступает набор из $n$ независимых параллельных заданий (требований, работ и

т.п.) $T=\{t_1, t_2, \ldots, t_n\}$. Известно время решения $\tau(t_i)$ задания $t_i$ на любом из процессоров. При этом каждое задание может быть обслужено любым процессором, в каждый момент времени отдельный процессор обслуживает не более одного задания и выполнение задания не прерывается для передачи его на другой процессор. Требуется найти такое распределение задании по процессорам, т.е. расписание, при котором время завершения выполнения заданий (или длина расписания) было бы минимальным. Под расписанием следует понимать отображение $A_R: T \to P$, такое что, если $A_R(t_i)=p_j$, то говорят что задание $t_i \in T$ расписании $R$ назначено на процессор $p_j \in P$. При сделанных выше допущениях, расписание можно представить разбиением множества заданий $T$ на $m$ непересекающихся подмножеств $T_j$ ($j=\overline{1,m}$), т.е. $\bigcup_{j=1}^{m} T_j = T$, $T_k \cap T_l = \varnothing$, $k \neq l$, $k, l \in \{1, \ldots, n\}$, где $T_j$ - подмножество заданий, назначенных на процессор $p_j$.

Критерий, используемый для минимизации времени завершения обслуживания заданий, является минимаксным критерием и определяется в следующем виде:

$$F = \max_{1 \le j \le m} F_j \to \min, \qquad (1)$$

где $F_j = \sum_{t_i \in T_j} \tau(t_i)$ время окончания работы процессора $p_j$.

**Алгоритм Генетический**

Используем следующие параметры ГА:

$n=3$ $m=7$ $N_{chr}=10$ - число особей(хромосом)

$N_{elit}=1$ $N_{\mathbf{lim}}=50$

Вероятности генетических операторов:

$\mathrm{P}_{\mathrm{cross}}=0{,}9$ $P_{mut}=0{,}1$

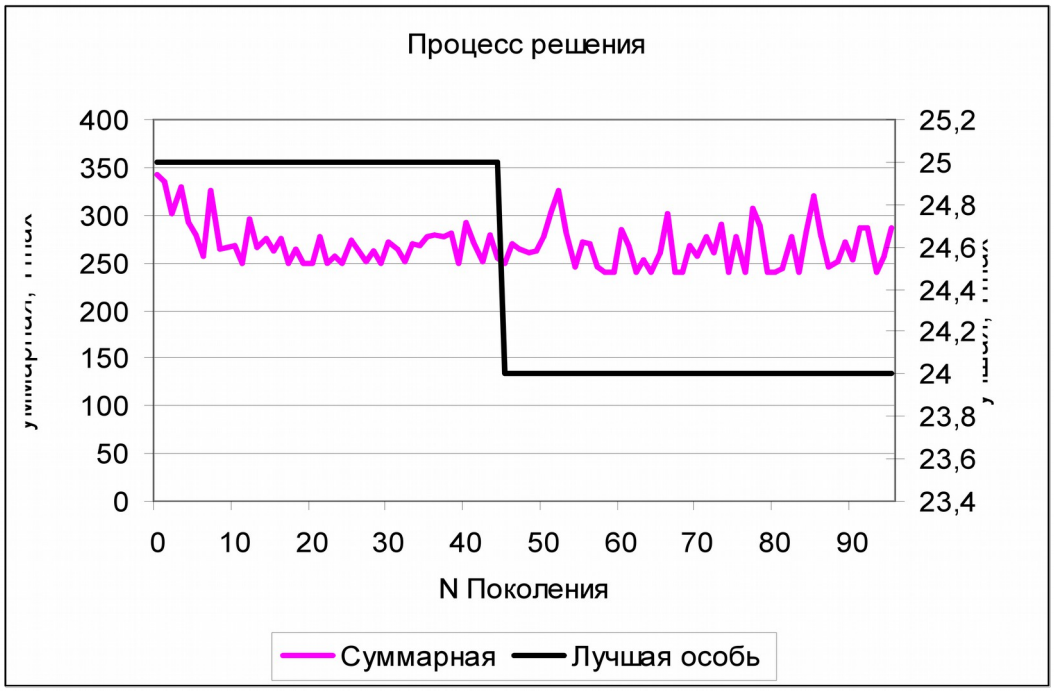



**Рис. 1 Процесс решения методом ГА**

Полученное расписание:

$$R = \begin{matrix} p_1 \\ p_2 \\ p_3 \end{matrix} \begin{matrix} 22 & - & - & - \\ 15 & 7 & - & - \\ 10 & 5 & 5 & 4 \end{matrix} \begin{matrix} , & f_1(p_1) = 22 \\ , & f_2(p_2) = 22 \\ , & f_3(p_3) = 24 \end{matrix}$$

На Рис. 1 показан процесс решения методом ГА, ключевыми моментами является наблюдаемая постоянная изменчивость получаемых решений, которую можно оценить по графику суммарной приспособленности, при этом лучшая особь с лучшей приспособленностью сохраняет свое решения от поколения к поколению до момента скачкообразного перехода к новому значению.