НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 9:
Тестирование белого и чёрного ящиков. Тестовое покрытие

ВМиК МГУ им. М.В. Ломоносова,
Кафедра АСВК, Лаборатория Вычислительных Комплексов
к.ф.-м.н., доцент Волканов Д.Ю.

План лекции

- Модель “белого (прозрачного) ящика”
- Модель “чёрного ящика”
- Методы покрытия тестов

Введение

- Модель программы в виде "белого ящика" предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления.
- Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный вид тестирования часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing).
- Структурные критерии тестирования базируются на основных элементах управляющего графа программы - операторах, ветвях и путях.

Управляющий граф программы

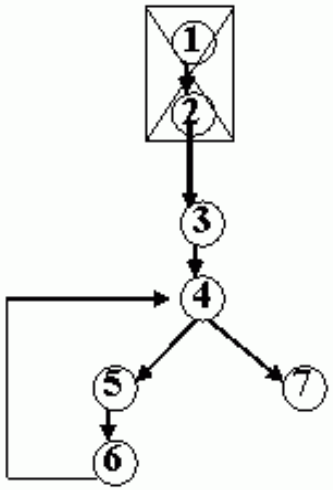
- ***Управляющий граф программы (УГП)*** отображает поток управления программы. Это граф G(V, A), где V(V1,… Vm) – множеств вершин (операторов), A(A1,… An) – множество дуг (управлений), соединяющих вершины.
- ***Путь*** – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины Vi и приходит в вершину Vj
- ***Ветвь*** – *путь*(V1, V2, … Vk), где V1 - либо первый, либо условный оператор, Vk - либо условный оператор, либо оператор выхода из программы, а все остальные операторы – безусловные.
- Число путей в программе может быть не ограничено (пути, различающиеся хотя бы числом прохождений цикла – разные). Ветви - линейные участки программы, их конечное число.
- Существуют реализуемые и нереализуемые пути в программе, в нереализуемые пути в обычных условиях попасть нельзя.

Реализуемые и нереализуемые пути

```
float Calc(float x, float y)  {
    float H;
1   if (x*x+y*y+2<=0)
2   H = 17;
3   else H = 64;
4   return H*H+x*x;  }
```

Пример

```
/* Функция вычисляет неотрицательную
     степень n числа x */
1    double Power(double x, int n){
2    double z=1; int i;
3    for ( i = 1;
4          i <= n;
5          i++ )
6            { z = z*x; }  /* Возврат в п.4 */
7     return z;
     }
```



Управляющий граф программы

примеры путей: (3,4,7), (3,4,5,6,4,5,6), (3,4), (3,4,5,6)
примеры ветвей: (3,4) (4,5,6,4) (4,7)

Примеры необозримого множества входных значений

1. Если программа P(x:int, y:int) реализована в машине с 64-разрядными словами, то мощность множества тестов ||(X,Y)||=2**64 (для перебора при 1Ггц потребуется ~ 3К лет)

2. Программа управления схватом робота, где интервал между моментами срабатывания схвата не определен (пример требует прогона бесконечного множества последовательностей входных значений).

Основные проблемы тестирования

- *Тестирование* программы на всех входных значениях невозможно.
- Невозможно *тестирование* и на всех *путях*.
- Следовательно, надо отбирать конечный *набор тестов*, позволяющий проверить программу **на основе интуитивных представлений**
- **Требование к тестам** - программа на любом из них должна останавливаться, т.е. не зацикливаться.
  В теории алгоритмов доказано, что не существует общего метода для решения этого вопроса, а также вопроса, достигнет ли программа на данном тесте заранее фиксированного оператора.
- Задача о *выборе конечного набора тестов* (X,Y) для проверки программы в общем случае неразрешима.

Требования к идеальному критерию тестирования

- **Критерий должен быть достаточным**, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
- **Критерий должен быть полным**, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
- **Критерий должен быть надежным**, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны обнаруживать или не обнаруживать ошибки программы
- **Критерий должен быть легко проверяемым**, например вычисляемым на тестах

Для нетривиальных классов программ в общем случае **не существует полного и надежного критерия**, зависящего от программ или спецификаций.

Классы критериев

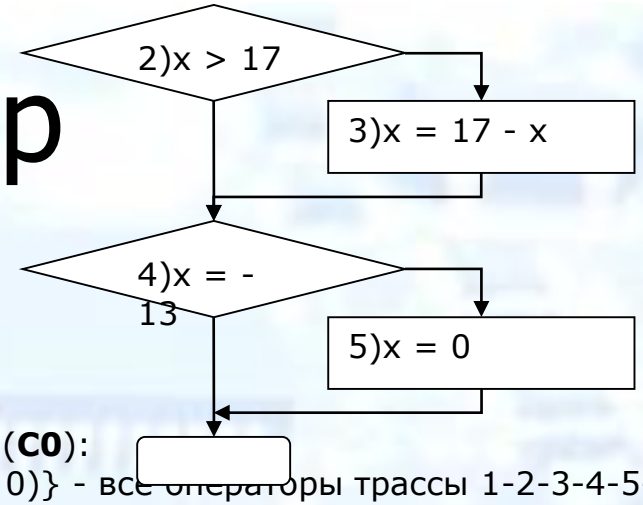
- *Структурные критерии* используют информацию о структуре программы (критерии "белого ящика")
- *Функциональные критерии* формулируются в описании требований к программному изделию (критерии «черного ящика")
- Критерии *стохастического тестирования* формулируются в терминах проверки наличия заданных свойств у тестируемого приложения средствами проверки некоторой статистической гипотезы.
- *Мутационные критерии*.

Структурные критерии

Используются на этапах модульного и интеграционного тестирования (Unit testing, Integration testing).

- **Тестирование команд** (критерий C0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза.
- **Тестирование ветвей** (критерий C1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза.
- **Тестирование путей** (критерий C2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раз. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой.
- **Тестирование условий** - покрытие всех булевских условий в программе. Критерии покрытия решений (ветвей - C1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

Пример




```
1  public void Method
   (ref int x) {
2  if (x>17)
3        x = 17-x;
4  if (x==-13)
5        x = 0;
6  }
```

критерий команд (**C0**):
(вх, вых) = {(30, 0)} - все операторы трассы 1-2-3-4-5-6

критерий ветвей (**C1**):
(вх, вых) = {(30, 0),
(17, 17)}

критерий путей (**C2**):
(вх, вых) = {(30,0), (17,17),
(-13,0), (21,-4)}

Условия операторов if

	(30,0)	(17,17)	(-13,0)	(21,-4)
2 if (x>17)	>	<=	<=	>
4 if (x==-13)	=	!=	=	!=

Недостаток структурных критериев

- Критерий ветвей С2 проверяет программу более тщательно, чем критерии - С1, однако даже если он удовлетворен, нет оснований утверждать, что программа реализована в соответствии со спецификацией.

Например, если спецификация задает условие, что|x|<100, невыполнимость которого можно подтвердить на тесте (-177,-177).

- ***Структурные критерии не проверяют соответствие спецификации,*** если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию С2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

Функциональные критерии

- *Функциональный критерий* - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением.
- При функциональном тестировании преимущественно используется модель "черного ящика»
- Проблема функционального тестирования - это, прежде всего, трудоемкость

Частные виды функциональных критериев

1. **Тестирование пунктов спецификации** - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.
2. **Тестирование классов входных данных** - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза.
3. **Тестирование правил** - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и

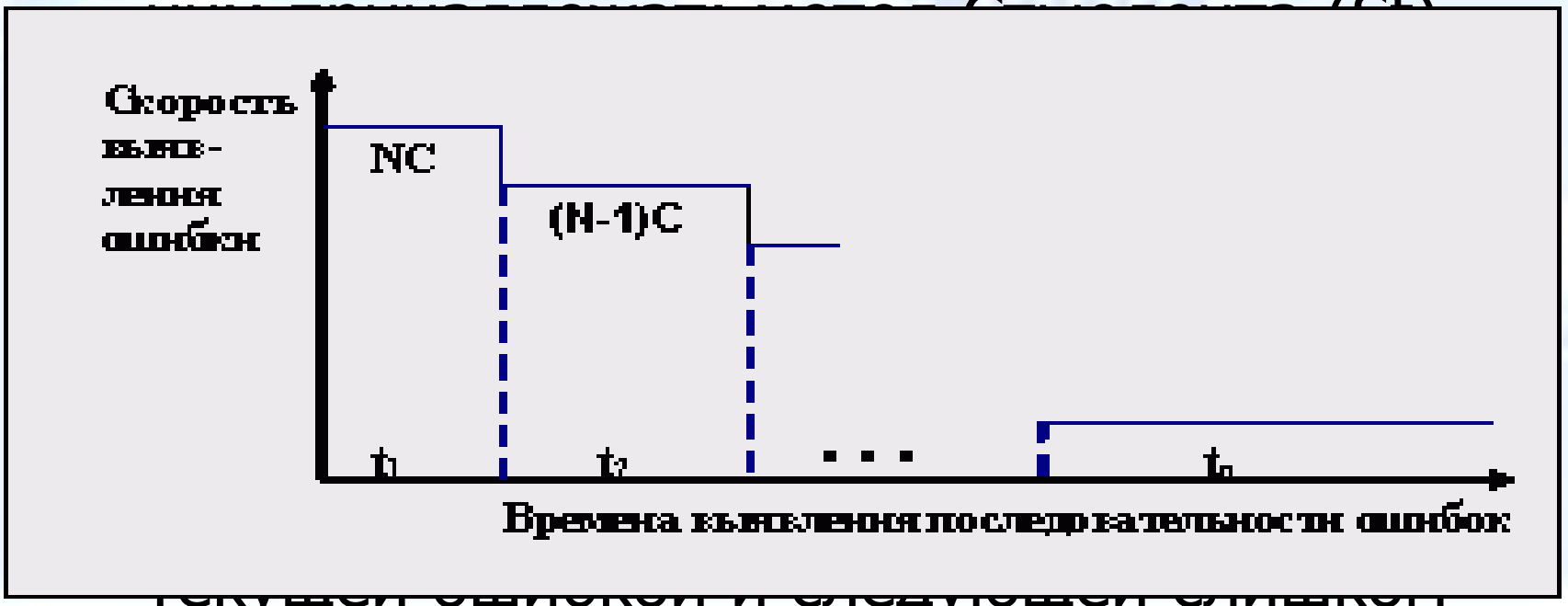
Стохастические критерии

Стохастическое тестирование применяется при тестировании сложных программных комплексов. Когда набор детерминированных тестов (X,Y) имеет громадную мощность и его невозможно разработать и исполнить, можно применить следующую методику:

- Разработать программы-имитаторы случайных последовательностей входных сигналов {x}.
- Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y)

Статистические методы окончания тестирования

- Статистические методы окончания тестирования - стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К



текущей ошибкой и следующей слишком

Мутационный критерий

Подход базируется на следующих понятиях:

- ***Мутации*** - мелкие ошибки в программе.
- ***Мутанты*** - программы, отличающиеся друг от друга *мутациями*.

Метод мутационного тестирования - в разрабатываемую программу P вносят *мутации*, т.е. искусственно

Модульное тестирование (Unit testing)

- ***Модульное тестирование*** - это тестирование программы на уровне отдельно взятых модулей, функций или классов.
- Цель *модульного тестирования* состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень

Принципы создания тестов

- На основе анализа ***потока управления***. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе структурных критериев тестирования С0, С1,С2. К ним относятся вершины, дуги, пути управляющего графа программы (УГП), условия, комбинации условий и т. п.
  К популярным критериям относятся *критерий покрытия функций* программы (каждая функция программы должна быть вызвана хотя бы один раз), и *критерий покрытия вызовов* (каждый вызов каждой

Построение набора тестов

1. Конструирование УГП
2. Выбор тестовых путей:
   - Статические методы
   - Динамические методы
   - Методы реализуемых путей
3. Генерация тестов, соответствующих тестовым путям

Методы построения множества тестов

- **Статические методы.** Построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная вершина.
  Недостатки – не учитывается возможная нереализуемость построенных путей тестирования (непредсказуемый процент брака).
  - Трудоемкость (переход от покрывающего множества путей к полной системе тестов осуществляется вручную)
  Достоинство - сравнительно небольшое количество необходимых ресурсов

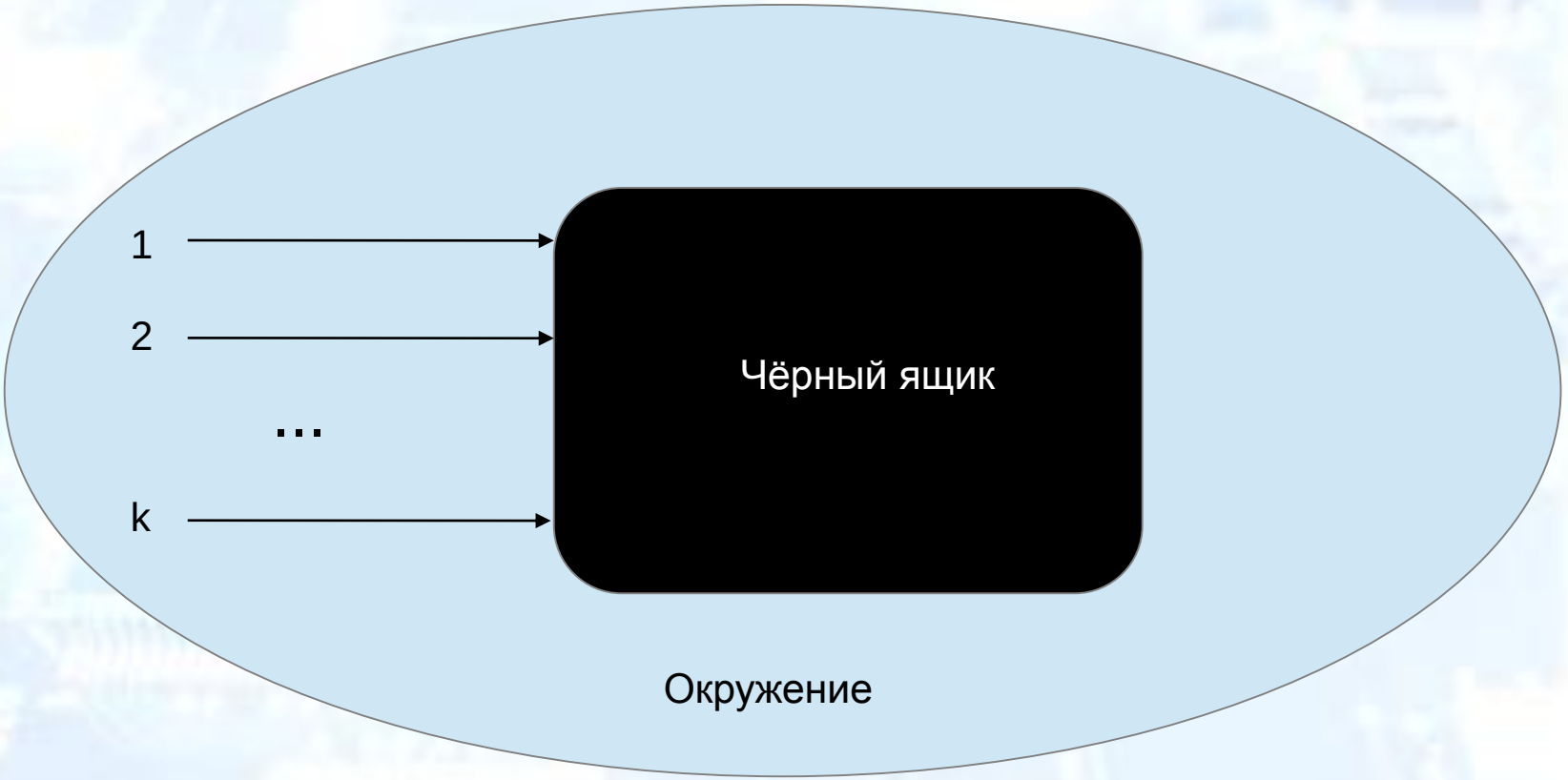
Сравнение методов

- Достоинство *статических методов* состоит в сравнительно небольшом количестве необходимых ресурсов. Однако их реализация может содержать непредсказуемый процент брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего множества путей к полной системе тестов пользователь должен осуществить вручную (трудоемко).

Методы отладки

- Результат выполнения теста ничего не говорит о том, где была допущена ошибка.
- Процедура исправления ошибки заключается в анализе протокола промежуточных вычислений с помощью следующих *методов*:
  - "Выполнение программы в уме" (deskchecking).
  - Вставка операторов протоколирования промежуточных результатов (logging).

Предметная область



Каждый из входов принимает n_1, n_2, …, n_k значений

Рабочая площадка

- Для объяснения принципов работы алгоритмов смоделируем такую ситуацию:
- Будем тестировать пользовательский интерфейс системы WebMoney
- Самая частая операция-перевод средств с одного кошелька на другой
- Выберем самые «важные» факторы, которые влияют на этот сценарий: объём передаваемой суммы; необходимость конвертировать валюту; тип кошелька, с которого идёт платёж, способ аутентификации пользователя в системе

Рабочая площадка

Возможные значения параметров:

Передаваемая сумма	Нужна ли конвертация валюты	Тип кошелька, с которого идет платеж	Браузер	Способ аутентификации	Операционная система
< 100 руб.	Не нужна	WMR рубли	Internet Explorer	Сертификат X.509	Windows XP
$100 - 10000$ руб.	Нужна	WMZ доллары	Mozilla Firefox	Enum-авторизация	Windows Vista
> 10000 руб.		WME евро	Opera	Логин и пароль	Debian Ubuntu
		WMU гривны	Google Chrome		Linux SUSE
					Linux RedHat

Рабочая площадка

- Нетрудно подсчитать, что полный перебор потребует 3*2*4*4*3*5 = 1440 тестов

Рабочая площадка

- Это немного, но, например, выполнение их всех вручную потребует значительных затрат

Примерная оценка:
Время выполнения одного теста ~5 минут(убедиться, что деньги дошли)
Переключить параметры WebMoney(конвертация, кошелёк и т. д.) - мгновенно
Переключить браузер — 10 секунд
Перезагрузить системы — 2 минуты
Самый худший случай(при каждом тесте перезагружаем ОС) ~10080 минут = 168 часов = 7 дней
Самый лучший случай сократит это время примерно в 3 раза — до 2х дней
И это только наш «упрощённый» пример

Hint: как видно из примера, немаловажную роль играет **порядок**, в котором тестируются параметры

Рабочая площадка

- Значит, нам нужен алгоритм, который бы помог сократить пространство перебора

- *и не забывать про «правильный» порядок тестирования

Рабочая площадка

- Классический случай: проверить простоту числа N.
- Решение «в лоб»: перебрать все числа до N/2.
- Математики же подсказывают нам, что достаточно перебрать все числа до sqrt(N)(и, более того, пропускать все чётные). И, к тому же, в этой задаче важно направление обхода(с какого конца отрезка начинать обход).
- Подобной оптимизации мы хотим добиться

- Определения

Основные определения

- *Покрывающий набор глубины t*
- Матрица из k столбцов
- В каждом столбце стоят значения соответствующего параметра
- Любая комбинация любых t факторов есть в этой матрице

Основные определения

- *Конфигурация покрывающего набора*
- Набор чисел (t; k, n_1,...., n_k)
- Множество наборов с такой конфигурацией обозначим так:
- CA(t; k, n_1,...,n_k)

Основные определения

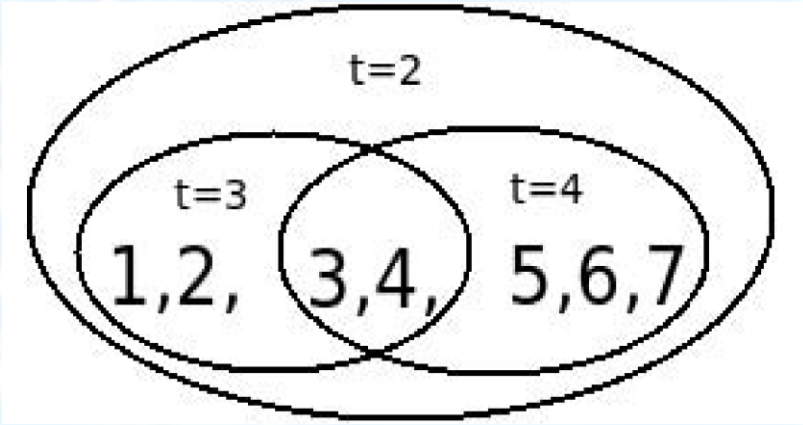
- *Размер набора*
- Это количество строк в матрице 

Основные определения

- CAN(t; k, n_1,...n_k) — размер минимального набора для данной конфигурации.
- Если $n_1=n_2=...=n_k$, то набор называется однородным

Основные определения

- *Набор переменной длины*
- Если в матрице есть комбинации размера t, t-1,...,2. Причём комбинации могут быть не полностью покрывающими. Например, если мы имеем 7 факторов:
- Пары всех факторов
- Тройки 1-4 факторов
- Четвёрки 3-7 факторов



Классификация алгоритмов

- Комбинаторные(прямые)
- Рекурсивные
- Оптимизационные

Классификация алгоритмов

- Какие параметры алгоритма особенно важны?
- Класс конфигураций наборов, которые можно получить с помощью рассматриваемого алгоритма
- Класс алгоритма
- Временная сложность
- Объем требуемой памяти

Обзор алгоритмов

- Булевский алгоритм

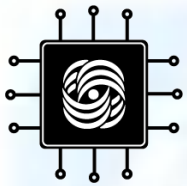
Обзор алгоритмов: булевский

- Тип: комбинаторный
- Получаемые наборы: CA(2; k, 2)
- Время: O(k)
- Память: $O(k*\log_2 k)$

1. Выберем наименьшее N такое, что выполнено $k \leq C_{N-1}^{\lceil N/2 \rceil}$. Здесь $\lceil x \rceil$ – наименьшее целое число, большее или равное x, C_q^r – биномиальный коэффициент. Это число N равно размеру итогового набора.

2. Первую строку набора сделаем состоящей целиком из 0.

3. Оставшиеся $N-1$ строк строятся по столбцам, в качестве этих столбцов берутся все возможные последовательности из $\lceil N/2 \rceil$ единиц и $\lceil N/2 \rceil - 1$ нулей.



Обзор алгоритмов: булевский

- Как он будет работать для нашего примера?

Обзор алгоритмов: булевский

- Итак, идём строго по алгоритму:
- k=6(количество параметров)
- Далее, подбираем N для этого неравенства

$$k \leq C_{N-1}^{\lceil N/2 \rceil}$$

В данном случае min N=6, т. е. в итоговой матрице будет 6 строк и 6 столбцов

Обзор алгоритмов: булевский

- Т.к. N = 6, то таблицу мы заполняем все возможными наборами из 3х единиц и 2х нулей

Обзор алгоритмов: булевский

0	0	0	0	0	0
0	1	1	1	0	1
0	0	0	1	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	0	0

Обзор алгоритмов: булевский

- К сожалению, данный алгоритм работает только с бинарными параметрами(т.е. которые принимают всего 2 значения), поэтому для нашего примера его можно применить только если отбросить часть значений.

Обзор алгоритмов

- Мультипликативный алгоритм

Обзор алгоритмов: мультипликативный

- Тип: рекурсивный
- Получаемые наборы: CA(t; k, $n_1 * n_2$)
- Время и память: O(k(1 + N_1 + N_2)), где N_1 и N_2 число строк в первом и втором наборах, соответственно

1. Пусть есть два покрывающих набора A из CA(t; k, n_1) и B из CA(t; k, n_2).

2. Обозначим элементы двух этих наборов через a_{ij} и b_{lj}, – у этих наборов одинаковое число столбцов, и, возможно, разное число строк.

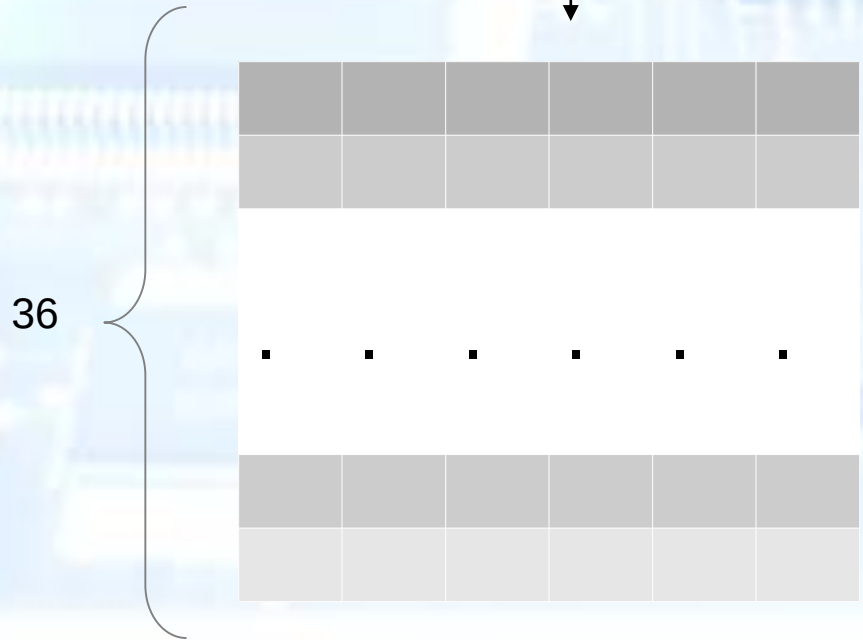
3. Любое число от 0 до $(n_1 \cdot n_2 - 1)$ можно однозначно представить в виде $n_2 \cdot i + m$, где i лежит от 0 до $(n_1 - 1)$, m – от 0 до $(n_2 - 1)$. Таким образом, можно однозначно установить соответствие между индексами строк таблицы получаемого набора и парами индексов строк двух исходных таблиц – (i, m), где i – индекс строк первого набора, m – индекс строк второго набора.

4. Элементы этой таблицы могут быть построены по формуле $x_{(i,m)j} = n_2 \cdot a_{ij} + b_{mj}$.

Обзор алгоритмов: мультипликативный

0	0	0	0	0	0
1	1	1	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	0	0	1
0	1	0	0	1	0

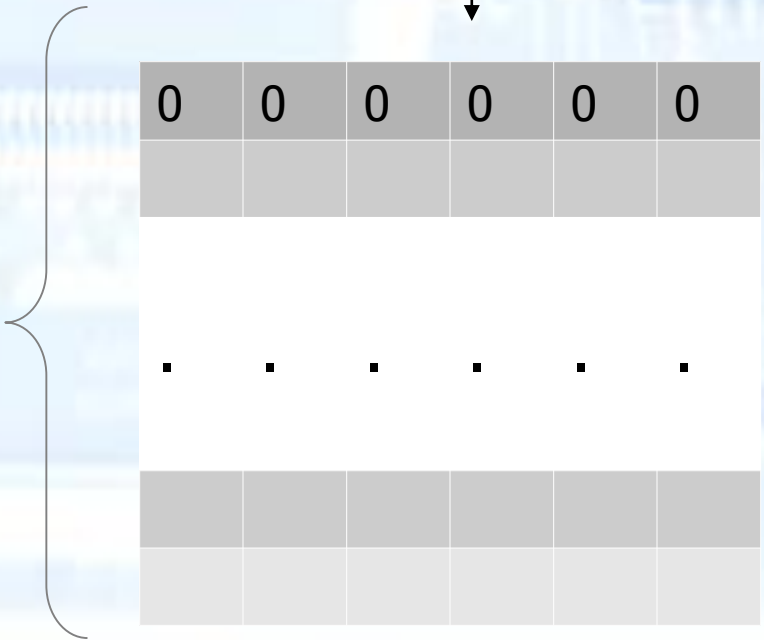
0	0	0	0	0	0
0	0	0	0	1	1
0	1	1	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1



Обзор алгоритмов: мультипликативный

0	0	0	0	0	0
1	1	1	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	0	0	1
0	1	0	0	1	0

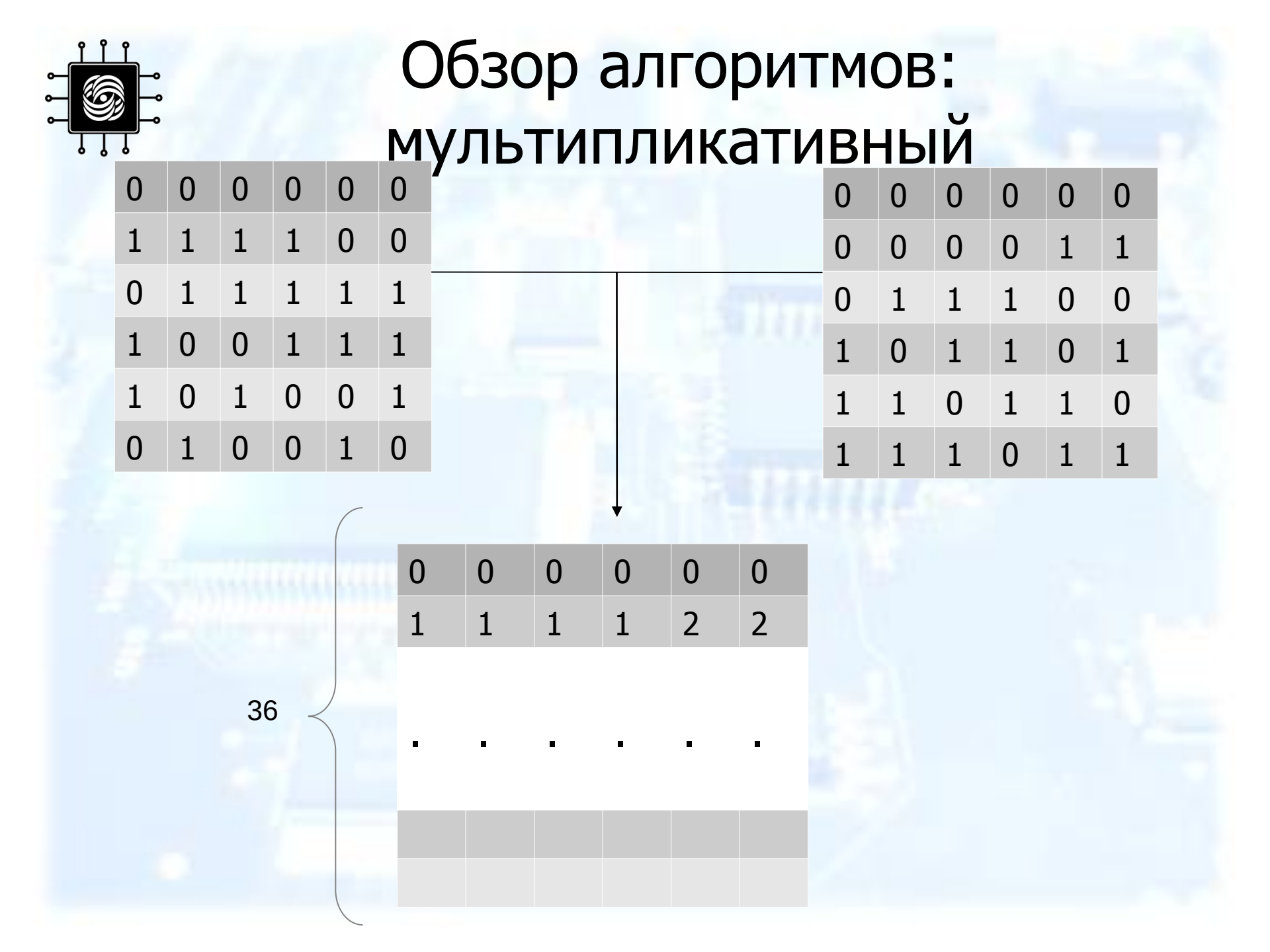
0	0	0	0	0	0
0	0	0	0	1	1
0	1	1	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1



Обзор алгоритмов: мультипликативный

0	0	0	0	0	0
1	1	1	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	0	0	1
0	1	0	0	1	0

0	0	0	0	0	0
0	0	0	0	1	1
0	1	1	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1



0	0	0	0	0	0
1	1	1	1	2	2
.	.	.	.	.	.

36

Обзор алгоритмов

- Это были одни из самых простых алгоритмов. В остальные сложно вникнуть за короткое время, и очень часто в них применяются зубодробительные теория групп\комбинаторика\математический анализ, поэтому далее будет приведена небольшая сводная таблица.

Алгоритм	Область применения	Класс алгоритма	Временная сложность и затраты по памяти
Булевский	(2; k, 2)	Прямой	$O(k)$, Память: $k\log_2 k$
Аффинный	(t; n+1, n), n – степень простого числа; (3; 2k+2, 2k),	Прямой	$n^t + (n+1)\log_2(n+1) + O(\log_p^4 n)$ Память: $O(n) + O(\log_p^3 n)$
Действия групп	(3; 2k, n+1), n – степень простого числа	Прямой	$O(2k((2k-1)(n^3-n)+n+1)+n^3)$ Память: $O(2k(2k-1)+n^4-n^2)$
	(2; 4, n), где $n \equiv 2 \pmod 4$		$O(4(n+1)n)$, Память: $O(4(n+1)+n^2)$,
	(2; l, g), (g, l)-исходный вектор покрытия, (2; l+1, g), особый (g, l)– исходный вектор покрытия		$O(l(l(g-1)+1))$ Память: $O(l+(g-1)g)$
Умножение	(t; k, n_1n_2) из (t; k, n_1) и (t; k, n_2), N_1, N_2 – число строк в исходных наборах	Рекурсивный	$O(k(1+N_1+N_2))$ Память: $O(k(1+N_1+N_2))$
Однородный, рекурсивный, t=2	(2; $k_1n^r + rk_1n^{r-1} + k_2n^r$, n), r ≥ 1, n – степень простого числа	Рекурсивный	$O(\sum_{i=1}^{r}((N+i(n^2-n))(k_1n^i + ik_1n^i + k_2n^i)))$ Память:$O((N+(r-1)(n^2-n))(k_1n^{r-1} + (r-1)k_1n^{r-2} + k_2n^{r-1}) + (n+1)n^2)$
	(2; $n(k_1+k_2)D_{r+1,n} + k_1D_{r,n}$, n), r ≥ 1, n – степень простого числа, $D_{r,t} = \sum_{i=1}^{\lfloor (r+1)/2 \rfloor} C_{i-1}^{r-i}t^{r-i}$		$O(\sum_{i=1}^{r}((N+i(n^2-n))(n(k_1+k_2)D_{i+1,n} + k_1D_{i,n})) + \sum_{i=1}^{r-1}(n^2(n(k_1+k_2)D_{i,n} + k_1D_{i-1,n})))$ Память:$O((N+(r-1)(n^2-n))(n(k_1+k_2)D_{r,n} + k_1D_{r-1,n}) + (n+1)n^2)$
Ordered design	(3; n+1, n+1) из OD(3, n+1, n+1) и (3; n+1, 2) размера N, n – степень простого числа	Рекурсивный	$O((n^3 - n + Nn(n+1)/2(n^2-1))(n+1))$ Память: $O((n^3-n+N)(n+1))$
Семейство совершенных хэш-функций	(t; k^{2p}, n) из (t; k, n) размера N, НОД((t–1)t/2, k)=1 (3; $(2v-1)^{2\wedge j}$, v), $v \equiv 0,\ 1 \bmod 3$, (3; $(2v-3)^{2\wedge j}$, v), $v \equiv 2 \bmod 3$, v>2, q ≥ v − 1–степень простого числа, j–любое целое	Рекурсивный	$O(k^p((t-1)t/2+1) + k^{2p}((t-1)t/2+1)^pN)$ Память: $O(k^p((t-1)t/2+1+N))$
	(t; k, n) из PHF $(N_1, t; k, m)$ и (t; m, n) размера N_2		$O(N_1N_2k)$ Память: $O(N_1k + N_2m)$
Теорема Roux	(3; 2k, n) из (3; k, n) и (2; k, n)(t; 2k, n) из (t; k, n),..., (t–2; k, n), t ≥ 4	Рекурсивный	$O(2N_{tmax}k(n+N_{tmax}(t-3)))$ Память: $O(2kN_{tmax}^2)$, N_{tmax} – число строк в наибольшем исходном наборе
IPO для наборов	Любые конфигурации, для которых нет более эффективных методов. Дополнение (t; k, n) до (t; k+p, n), где p>0	Жадный	$O(n^{2t}\log^2(n^tC_t^k)/n^{t-1}C_t^k + n^{2t}\log(n^tC_t^k)C_t^k)$ Память: $O(n^{t+1}\log(n^tC_t^k) + n^tC_{t-1}^{k-1}$

Алгоритм	Область применения	Класс алгоритма	Временная сложность и затраты по памяти
Выбрасывание лишних значений	Необходимы дальнейшие исследования.	Редуцирующий	Зависит от размеров исходного набора
Двойная проекция	$(2;\ n{+}1{+}t,\ (n-t^{n+1}, s^t)$, n – степень простого числа, $1 \le t \le n$ и $1 \le s \le n-t$	Редуцирующий	$O(N(k{+}1))$ Память:$O(Nk)$
Комбинирование блоков	$(2;\ k,\ n_1\ \ldots\ n_k)$ и $k > \max(n_1\ \ldots\ n_k)$ Необходимы дальнейшие исследования.	Рекурсивный	$O(n^2 + k\log_2 k)$, Память: $O(n^2(n+1) + k(\lceil \log_{n+1} k\rceil (n^2-1)+$ $+1))$, где $n \ge d$ и $n = p^m$, p – простое число, $k \ge 1$
Неоднородный, рекурсивный, t=2	$(2;\ l_1 + \ldots + l_k,\ m_{1,1}, \ldots, m_{1,l1}, \ldots,$ $m_{k,1}, \ldots, m_{k,lk})$, из $(2;\ k,\ v_1, \ldots, v_k)$ с профилем $(d_1,\ \ldots,\ d_k)$, и $(2;\ l_i,\ m_{i,1}, \ldots, m_{i,li})$ с профилем $(f_{i,1}, \ldots, f_{i,li})$, $i \in [1,\ k]$ и $m_{ij} \le v_i,\ j \in [1,\ l_i]$	Рекурсивный	$(N+M)L + \sum_{i=1}^{k}(M_i l_i)$ Память:$(N+M)L + \max_{i=1}^{k}(M_i l_i)$, $M = \max_{i=1}^{k}(M_i - d_i),\ L = \sum_{i=1}^{k} l_i$, где M_i – число строк в исходных наборах
IPO	Дополнение $(t;\ k,\ n_1 \ldots n_k)$ до $(t;\ k+p,\ m_1, \ldots m_{k+p})$, где $n_i \le m_i$, $i \in [1,\ k], \exists j :\ j \in [1,\ k]$, $n_j < m_j$ и/или $p{>}0$	Жадный	$O(tk^6 d^t(d^t + k + d^t k))$ Память: $O(d^t(k\log k + tC_{t-1}^{k-1}))$
Оптимизация исходного набора	Уменьшение строк в исходном наборе $(t;\ k,\ n_1 \ldots n_k)$ размера N	Редуцирующий Оптимизационный	$O(N_e(M(C_t^k tN + Nk + N - 1)+$ $+6k))$, M – число перезапусков для поиска улучшений. Память: $O(Nk{+}d^t)$

Обзор алгоритмов: плюсы и минусы

- Прямые:
- +Дают наборы, близкие к минимальным, за хорошее время
- -Узкая область применения
- -Нет возможности отсеивать строки набора

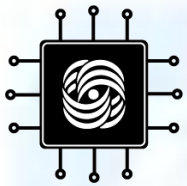
Обзор алгоритмов: плюсы и минусы

- Рекурсивные:
- Для особых классов конфигураций неплохо дополняют прямые, плюсы и минусы те же самые

Обзор алгоритмов: плюсы и минусы

- Жадные:

- +Универсальны, поэтому используются в большинстве инструментов
- +Возможность отсеивать строки наборов по ограничениям на возможные комбинации;
- +Возможность применения их для построения наборов переменной глубины;
- +Возможность расширять уже существующий тестовый набор.
- -Обратная зависимость между близостью результирующего набора к минимальному и временем работы алгоритма



Спасибо за внимание!