Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра «Вычислительная техника»

Дисциплина «Информационная безопасность»

**Лабораторная работа 1**

**Тестирование псевдослучайных последовательностей**

Выполнил

Студент группы ИВТАСбд-41

Ведин В.А.

Проверил:

Старший преподаватель кафедры «ВТ»

Мартынов А.И.

Ульяновск

2025

# Задание

Реализовать приложение, позволяющее выполнять следующие действия:

1. Задавать длину генерируемой последовательности в битах
2. Генерировать псевдослучайную последовательность 0 и 1 с помощью стандартного алгоритма генерации случайных чисел
3. Загружать последовательность из текстового файла
4. Сохранять полученную последовательность в файл и выводить ее на экран приложения
5. Проверять полученную последовательность с помощью реализованных тестов. Результат проверки должен отображаться в приложении

# .

# Ход работы

# 1. Частотный тест

# Данный тест оценивает пропорцию нулей и единиц в проверяемой последовательности. Тест определяет, является ли количество нулей и единиц в последовательности приблизительно таким же, как должно быть в истинно случайно последовательности.

# 1. Входная последовательность, состоящая из 0 и 1 (будем обозначать ее ε), преобразовывается в последовательность -1 и 1 (будем обозначать ее X) соответственно:

# 2. Вычисляется сумма , где n – количество элементов проверяемой последовательности.

# 3. Вычисляется статистика

# 4. Если , то тест считается успешно пройденным, иначе делается вывод о том, что последовательность является неслучайной.

# Код, реализующий частотный тест представлен в листинге 1.

# Листинг 1. Частотный тест

|  |
| --- |
| def generate\_bit\_seq(seq\_len: Optional[int] = 10000,input\_file\_path: Optional[str] = "",output\_file\_path: Optional[str] = "",seed: int = 42) -> list[int]:"""Генерирует или загружает последовательность битов (0 и 1).Args:seq\_len: Длина генерируемой последовательности (по умолчанию 10000 бит)input\_file\_path: Путь к файлу для загрузки последовательностиoutput\_file\_path: Путь для сохранения последовательностиseed: Seed для генератора случайных чиселReturns:list[int]: последовательность битов (0 и 1).Note:Если указан input\_file\_path, последовательность загружается из файла.Иначе генерируется случайная последовательность заданной длины."""bit\_seq = Nonerandom.seed(seed)if input\_file\_path != "":with open(input\_file\_path, "r", encoding="utf-8") as f:bit\_seq = list(f.read())bit\_seq = [int(bit) for bit in bit\_seq]bit\_seq = [0 if random.random() < 0.5 else 1 for \_ in range(seq\_len)] if bit\_seq is None else bit\_seqif output\_file\_path != "":with open(output\_file\_path, "w", encoding="utf-8") as f:f.write("".join(map(str, bit\_seq)))return bit\_seq |

2. Тест на последовательность одинаковых бит.

Этот тест анализирует количество цепочек в проверяемой последовательности, где цепочка – это непрерывная последовательность одинаковых бит. Под цепочкой длиной k понимается цепочка, состоящая из ровно k бит и ограниченная до и после битами с противоположным значением. Тест определяет, является ли количество цепочек из нулей и единиц различной длины в последовательности приблизительно таким же, как должно быть в истинно случайной последовательности.

Шаги алгоритма:

1. Вычисляется частота, с которой в проверяемой последовательности встречаются единицы:

2. Вычисляется значение , где , если и иначе

3. Вычисляется статистика

4. Если , то тест считается успешно пройденным, иначе делается вывод о том, что последовательность является не случайной

Код реализующий данный тест представлен в листинге 2.

Листинг 2. Тест на последовательность одинаковых бит

|  |
| --- |
| def r(bit\_seq: list[int], k: int):  """  Вспомогательная функция для теста на последовательность одинаковых бит.  Определяет, является ли k-тый бит началом новой цепочки.  Args:  bit\_seq: Последовательность битов  k: Индекс текущего бита    Returns:  int: 1 если бит k и k+1 разные (начало новой цепочки) 0 если одинаковые  """  if bit\_seq[k] == bit\_seq[k + 1]:  return 0  return 1  def identical\_bit\_seq\_test(bit\_seq: list[int]) -> bool:  """  Тест на последовательность одинаковых бит (Rusn Test).  Анализирует количество цепочек (непрерывных последовательностей одинаковых бит)  в проверяемой последовательности  Args:  bit\_seq: Последовательность битов (0 и 1)    Returns:  bool: True если тест пройден, False в противном случае    Algorithm:  1. Вычисление частоты единиц (pi)  2. Вычисление количества переходов V\_n  3. Вычисление статистики S  4. Сравнение с критическим значением CONST  """  pi = 1 / (len(bit\_seq)) \* sum(bit\_seq)  V\_n = sum([r(bit\_seq, k) for k in range(len(bit\_seq) - 1)]) + 1  S = abs(V\_n - 2 \* len(bit\_seq) \* pi \* (1 - pi))  S /= 2 \* math.sqrt(2 \* len(bit\_seq) \* pi \* (1 - pi))  if S <= CONST:  return True  return False |

3. Расширенный тест на произвольные отклонения

Этот тест оценивает общее число посещений определенного состояния при произвольном обходе кумулятивной суммы. Цель этого теста – определить отклонения от ожидаемого числа посещений различных состояний при произвольном обходе. Фактически данный тест состоит из 18 тестов, по одному для каждого состояния: −9, −8, … − 1, 1, 2, … , 9.

Шаги алгоритма:

1. Входная последовательность, состоящая из 0 и 1 (будем обозначать ее ), преобразовывается в последовательность -1 и 1 (будем обозначать ее X) соответственно:

2. Вычисляются суммы последовательно удлиняющихся подпоследовательностей, начинающихся с

...

3. Формируется новая последовательность

4. Вычисляется где k – количество нулей в полученной последовательности

5. Для каждого из 18 состояний вычисляется которое показывает, сколько раз состояние j встречалось в последовательности . Здесь j = -9, -8, ..., -1, 1, 2, ..., 9.

6. Вычисляются 18 статистик для каждого состояния j = -9, -8, ..., -1, 1, 2, ..., 9.

7. Если все статистики то тест считается успешно пройденным, если же хотя бы для одной статистики это условие не выполнилось, то делается вывод о том, что последовательность является неслучайной.

Код данного теста представлен в листинге 3.

Листинг 3. Расширенный тест на произвольные отклонения

|  |
| --- |
| def extended\_random\_deviation\_test(bit\_seq: list[int]) -> bool:  """  Расширенный тест на произвольные отклонения.  Оценивает общее число посещений определенного состояния при произвольном обходе  кумулятивный суммы. Состоит из 18 подтестов для состояний из -9 до 9 (кроме 0).  Args:  bit\_seq: Последовательность битов (0 и 1)    Returns:  bool: True если все 18 тестов пройдены, False если хотя бы один не пройден    Algorithm:  1. Преобразование битов в -1 и 1  2. Вычисление кумулятивных сумм  3. Фомирование расширенной последовательности S'  4. Подсчёт количества нулей L  5. Вычисление количества посещений каждого состояния  6. Вычисление статистик Y\_j для каждого состояния  7. Проверка всех статистик на превышение CONST  """  states = [str(i) for i in range(-9, 10) if i != 0]  fixed\_bit\_seq = [2 \* bit - 1 for bit in bit\_seq]  S = [0, fixed\_bit\_seq[0]]  fixed\_bit\_seq = fixed\_bit\_seq[1:]  S.extend([S.append(S[-1] + bit) for bit in fixed\_bit\_seq])  S.append(0)  L = S.count(0)  theta = {state: S.count(int(state)) for state in states}  Y = {  j: (abs(theta[j]) - L) / math.sqrt(2 \* L \* (4 \* abs(int(j)) - 2))  for j in states  }  if all([y < CONST for y in list(Y.values())]):  return True  return False |

# Вывод

В результате выполнения лабораторной работы был успешно реализован и протестирован генетический алгоритм для оптимизации сельскохозяйственного планирования, который продемонстрировал эффективность в поиске компромиссных решений между максимизацией урожая и минимизацией затрат. Экспериментальное сравнение различных операторов скрещивания и мутации показало существенное влияние выбора генетических операторов на качество и скорость сходимости алгоритма, при этом наилучшие результаты были достигнуты при комбинациях, обеспечивающих баланс между исследованием пространства решений и использованием найденных хороших решений. Полученные результаты подтверждают практическую применимость разработанного подхода для решения задач оптимизации в сельском хозяйстве..

# Приложение.

# Ссылка на репозиторий с кодом лабораторной работы: <https://github.com/Tsaranchik/Artifical_Intelligence_Systems/tree/master/lab1>