Министерство цифрового развития

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет
телекоммуникаций и информатики» (СибГУТИ)

Кафедра прикладной математики и кибернетики

Расчетно-графическая работа
по дисциплине “Защита информации”
по теме «Доказательство с нулевым знанием»
Вариант: 2. Гамильтонов цикл

Выполнил:
студент группы ИП-216
Русецкий А.С.

Работу проверил:
Ассистент
Истомина Анна Сергеевна

Новосибирск 2025 г.

Содержание

Постановка задачи

Необходимо написать программу, реализующую протокол доказательства с нулевым знанием для задачи «Гамильтонов цикл».

Задача является NP-полными и не имеет быстрых методов для решения, поэтому для тестирования необходимо будет генерировать правильные решения при помощи дополнительно разработанных программ.

Информацию о графах считывать из файла. В файле описание графа будет определяться следующим образом:

1. в первой строке файла содержатся два числа n<1001 и m <=n^2, количество вершин графа и количество рёбер соответственно;
2. в последующих m строках содержится информация о рёбрах графа, каждое из которых описывается с помощью двух чисел (номера вершин, соединяемых этим ребром);
3. перечисляются цвета вершин графа;

Программа должна наглядно демонстрировать работу алгоритма, возможно (но не обязательно) в графическом режиме. Текст программы должен содержать исчерпывающие комментарии, тем не менее, следует воздержаться от описания очевидных действий.

Теоретические сведения

Доказательство с нулевым знанием (Zero-Knowledge Proof, ZKP) — это криптографический протокол, позволяющий одной стороне (доказывающему) убедить другую (проверяющего) в истинности утверждения, не раскрывая никакой дополнительной информации, кроме самого факта его истинности.

Ключевые свойства ZKP:

- полнота - истина всегда доказывается;
- корректность - ложь почти никогда не проходит;
- нулевое разглашение - проверяющий не узнает ничего лишнего.

Это используется для аутентификации, подтверждения личности и верификации транзакций в блокчейне (например, zk-SNARKs в Zcash), сохраняя конфиденциальность данных.

Результат работы программы

```
PS C:\porogiii\4_course\secure_information\rgr_v2> python .\rgr_v2.py test_graph.txt --cycle .\test_cycle.txt

Граф G (матрица смежности):
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 0
Гамильтонов цикл в G (1-indexed): 1 2 3 4

--- РАУНД 1 ---
Bob запросил: показать изоморфизм (2)
Перестановка:  3, 4, 2, 1
Bob принимает ответ: True

--- РАУНД 2 ---
Bob запросил: показать цикл (1)
Гамильтонов цикл в H: 4 1 3 2
Рёбра цикла H: (4,1), (1,3), (3,2), (2,4)
Bob принимает ответ: True

--- РАУНД 3 ---
Bob запросил: показать цикл (1)
Гамильтонов цикл в H: 3 2 1 4
Рёбра цикла H: (3,2), (2,1), (1,4), (4,3)
Bob принимает ответ: True

--- РАУНД 4 ---
Bob запросил: показать изоморфизм (2)
Перестановка:  1, 3, 2, 4
Bob принимает ответ: True

Результат: Боб принял 4 из 4 раундов.
PS C:\porogiii\4_course\secure_information\rgr_v2>
```

```
PS C:\porogiii\4_course\secure_information\rgr_v2> python .\rgr_v2.py test2_graph.txt --cycle .\test2_cycle.txt

Граф G (матрица смежности):
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0
Гамильтонов цикл в G (1-indexed): 1 2 3

--- РАУНД 1 ---
Bob запросил: показать цикл (1)
Гамильтонов цикл в H: 2 3 1
Рёбра цикла H: (2,3), (3,1), (1,2)
Bob принимает ответ: False

--- РАУНД 2 ---
Bob запросил: показать цикл (1)
Гамильтонов цикл в H: 2 3 1
Рёбра цикла H: (2,3), (3,1), (1,2)
Bob принимает ответ: False

--- РАУНД 3 ---
Bob запросил: показать цикл (1)
Гамильтонов цикл в H: 1 2 3
Рёбра цикла H: (1,2), (2,3), (3,1)
Bob принимает ответ: False

--- РАУНД 4 ---
Bob запросил: показать цикл (1)
Гамильтонов цикл в H: 3 2 4
Рёбра цикла H: (3,2), (2,4), (4,3)
Bob принимает ответ: False

Результат: Боб принял 0 из 4 раундов.
PS C:\porogiii\4_course\secure_information\rgr_v2>
```

Исходный код

```
import sys
import argparse
import random
import math
from typing import List, Tuple, Optional


# ---
# RSA
# ---

def is_probable_prime(n: int, k: int = 12) -> bool:
    """
    Тест простоты с k итерациями.
    Использует малые простые числа для быстрой фильтрации,
    затем проверяет n-1 = 2^s * d через случайные основания a.
    """
    if n < 2: return False
    small_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
    for p in small_primes:
        if n % p == 0: return n == p

    d, s = n - 1, 0
    while d % 2 == 0:
        d //= 2
        s += 1

    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, d, n)  # a^d mod n
        if x == 1 or x == n - 1: continue

        for __ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1: break
        else:
            return False  # Составное число
    return True


def gen_prime(bits: int) -> int:
    """
    Генерирует простое число заданной битовой длины.
    Формирует нечетное число с установленным старшим битом,
    проверяет простоту.
    """
    while True:
        # | (1<<(bits-1)) | 1: четное число нужной длины
        p = random.getrandbits(bits) | (1 << (bits - 1)) | 1
```

```
        if is_probable_prime(p): return p


def egcd(a, b):
    """
    Расширенный алгоритм Евклида для нахождения НОД.
    Рекурсивная реализация возвращает (g,x,y) где g=ax+by.
    """
    if b == 0: return a, 1, 0
    g, x1, y1 = egcd(b, a % b)
    return g, y1, x1 - (a // b) * y1


def modinv(a, m):
    """Модульная инверсия через расширенный алгоритм Евклида."""
    g, x, _ = egcd(a, m)
    if g != 1: raise ValueError("No modular inverse")
    return x % m


def generate_rsa_keypair(bits: int = 256) -> Tuple[int, int, int]:
    """
    Генерация RSA ключей (e,d,N).
    Использует стандартное e=65537, повторяет генерацию пока gcd(e,ϕ(N))=1.
    """
    while True:
        p, q = gen_prime(bits), gen_prime(bits)
        if p == q: continue  # p ≠ q
        N = p * q
        phi = (p - 1) * (q - 1)
        e = 65537
        if math.gcd(e, phi) == 1:
            d = modinv(e, phi)
            return e, d, N


# ---------------
# Работа с графом
# ---------------

def read_graph_file(path: str) -> Tuple[int, List[Tuple[int, int]],
Optional[List[int]]]:
    """
    Парсит файл
    Игнорирует пустые строки и комментарии (#).
    Вершины нумеруются с 1 в файле, ребра неориентированные.
    """
    with open(path, 'r', encoding='utf-8') as f:
        tokens = []
        for line in f:
            line = line.strip()
```

```
            if not line or line.startswith('#'): continue
            tokens.extend(line.split())
    n, m = int(tokens[0]), int(tokens[1])
    edges = [(int(tokens[i]), int(tokens[i + 1])) for i in range(2, 2 + 2 * m,
2)]
    cycle = None
    if len(tokens) > 2 + 2 * m and len(tokens[2 + 2 * m:]) >= n:
        cycle = [int(x) for x in tokens[2 + 2 * m:2 + 2 * m + n]]
    return n, edges, cycle


def build_adj_matrix(n: int, edges: List[Tuple[int, int]]) -> List[List[int]]:
    """
    Строит симметричную матрицу смежности (0/1).
    Индексы в матрице 0-based, вершины в edges 1-based.
    """
    A = [[0] * n for _ in range(n)]
    for a, b in edges:
        A[a - 1][b - 1] = 1
        A[b - 1][a - 1] = 1  # Неориентированный граф
    return A


def find_hamiltonian_cycle_bruteforce(adj: List[List[int]]) ->
Optional[List[int]]:
    """
    Полный перебор для поиска гамильтонова цикла в маленьких графах.
    Возвращает 1-based цикл или None.
    """
    n = len(adj)
    path = [0];
    used = [False] * n;
    used[0] = True

    def dfs():
        if len(path) == n: return adj[path[-1]][path[0]] == 1  # Замыкание
        u = path[-1]
        for v in range(n):
            if not used[v] and adj[u][v] == 1:
                used[v] = True;
                path.append(v)
                if dfs(): return True
                path.pop();
                used[v] = False
        return False

    if dfs(): return [v + 1 for v in path]
    return None


# ------------
```

```python
# Перестановка
# ------------

def permute_graph(adj: List[List[int]]) -> Tuple[List[List[int]], List[int]]:
    """
    Создает случайную перестановку вершин π и изоморфный граф H=π(G).
    H[i][j] = G[π(i)][π(j)], индексы 0-based.
    """
    n = len(adj)
    perm = list(range(n))
    random.shuffle(perm)
    H = [[adj[perm[i]][perm[j]] for j in range(n)] for i in range(n)]
    return H, perm


def commit_matrix(H: List[List[int]], e: int, N: int) -> Tuple[List[List[int]],
List[List[int]], List[List[int]]]:
    """
    RSA-коммитмент матрицы H: F[i][j] = (H[i][j] + 2*r)^e mod N.
    Случайное r делает коммитмент скрывающим и связывающим.
    Возвращает закоммиченную F, s=H+2r и r матрицы.
    """
    n = len(H)
    F, s_mat, r_mat = [[0] * n for _ in range(n)], [[0] * n for _ in range(n)],
[[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            r = random.randrange(1, max(2, N - 1))  # r ∈ [1, N-2]
            s = H[i][j] + 2 * r  # Парное кодирование 0=>even, 1=>odd
            F[i][j] = pow(s, e, N)
            s_mat[i][j] = s
            r_mat[i][j] = r
    return F, s_mat, r_mat


def apply_permutation_to_cycle(cycle: List[int], perm: List[int]) -> List[int]:
    """
    Преобразует гамильтонов цикл G в цикл H=π(G).
    Строит обратную перестановку pos_in_H[π(i)]=i.
    """
    n = len(perm)
    pos_in_H = [-1] * n
    for h_idx, g_idx in enumerate(perm): pos_in_H[g_idx] = h_idx
    return [pos_in_H[v - 1] + 1 for v in cycle]  # 1-based вывод


# --------------
# Проверка Бобом
# --------------

def verify_ciphertext(s: int, e: int, N: int, c_expected: int) -> bool:
```

```
    """Проверяет корректность RSA коммитмента."""
    return pow(s, e, N) == c_expected


def verify_revealed_cycle_on_H(edges: List[Tuple[int, int]], n: int) -> bool:
    """
    Проверяет, что раскрытые ребра образуют гамильтонов цикл в H:
    1. Ровно n ребер
    2. Каждая вершина степени 2
    3. Граф связный
    """
    if len(edges) != n: return False
    deg = [[0] * n for _ in range(n)];
    d = [0] * n
    for u, v in edges:
        d[u] += 1;
        d[v] += 1;
        deg[u][v] = deg[v][u] = 1

    if any(x != 2 for x in d): return False  # Степени ≠ 2

    visited = [False] * n;
    stack = [0];
    visited[0] = True
    while stack:
        x = stack.pop()
        for y in range(n):
            if deg[x][y] and not visited[y]:
                visited[y] = True;
                stack.append(y)
    return all(visited)


# ------------
# Демонстрация
# ------------

def print_graph(adj: List[List[int]], name="G"):
    """Вывод матрицы смежности в читаемом виде."""
    print(f"\nГраф {name} (матрица смежности):")
    for row in adj: print(" ".join(str(x) for x in row))


def print_permutation(perm: List[int]):
    """Вывод перестановки в 1-based нотации."""
    print("Перестановка: ", ", ".join(str(x + 1) for x in perm))


# --------
# Протокол
# --------
```

```
def run_protocol_once(adj_G: List[List[int]], cycle_G: List[int], e: int, d: int,
N: int) -> bool:
    """
    Один раунд протокола.
    Алиса: H=π(G), коммитмент F=(H+2r)^e.
    Боб: случайно выбирает challenge 1=покажи цикл или 2=покажи π.
    """
    H, perm = permute_graph(adj_G)
    F, s_mat, r_mat = commit_matrix(H, e, N)
    challenge = random.choice([1, 2])

    if challenge == 1:
        # Раскрытие цикла: ребра + s для них
        cycle_in_H = apply_permutation_to_cycle(cycle_G, perm)
        edges = [(cycle_in_H[i] - 1, cycle_in_H[(i + 1) % len(cycle_in_H)] - 1)
                 for i in range(len(cycle_in_H))]
        print("Bob запросил: показать цикл (1)")
        print("Гамильтонов цикл в H:", " ".join(str(x) for x in cycle_in_H))
        print("Рёбра цикла H:", ", ".join(f"({u + 1},{v + 1})" for u, v in
edges))

        ok = True
        # Проверка коммитментов ребер цикла
        for u, v in edges:
            if not verify_ciphertext(s_mat[u][v], e, N, F[u][v]):
                ok = False;
                break
        if ok and not verify_revealed_cycle_on_H(edges, len(H)): ok = False

    else:
        # Раскрытие изоморфизма
        print("Bob запросил: показать изоморфизм (2)")
        print_permutation(perm)
        ok = True
        # Проверка H[i][j] == G[π(i)][π(j)] для всех i,j
        for i in range(len(H)):
            for j in range(len(H)):
                if H[i][j] != adj_G[perm[i]][perm[j]]:
                    ok = False;
                    break
            if not ok: break

    print("Bob принимает ответ:", ok)
    return ok


# --------------
# Главная логика
# --------------
```

```
def protocol_demo(graph_file: str, cycle_file: Optional[str], rounds: int = 8,
rsa_bits: int = 256):
    """
    Демонстрация протокола доказательства с нулевым знанием для задачи
"гамильтонов цикл".
    Многократные раунды для статистической достоверности.
    """
    n, edges, cycle_in_file = read_graph_file(graph_file)
    if cycle_file:
        with open(cycle_file, 'r', encoding='utf-8') as f:
            tokens = [x for line in f if line.strip() and not
line.startswith('#')
                      for x in line.split()]
        cycle = [int(x) for x in tokens[:n]]
    else:
        cycle = cycle_in_file

    A = build_adj_matrix(n, edges)
    print_graph(A, "G")
    print("Гамильтонов цикл в G (1-indexed):", " ".join(str(x) for x in cycle))

    e, d, N = generate_rsa_keypair(bits=rsa_bits)

    successes = 0
    for r in range(1, rounds + 1):
        print(f"\n--- РАУНД {r} ---")
        ok = run_protocol_once(A, cycle, e, d, N)
        if ok: successes += 1
    print(f"\nРезультат: Боб принял {successes} из {rounds} раундов.")


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("graph", help="файл с графом")
    parser.add_argument("--cycle", default=None, help="отдельный файл с циклом")
    parser.add_argument("--rounds", type=int, default=8, help="количество
раундов")
    parser.add_argument("--rsa-bits", type=int, default=256, help="разрядность
RSA")
    args = parser.parse_args()
    protocol_demo(args.graph, args.cycle, args.rounds, args.rsa_bits)


if __name__ == "__main__":
    main()
```