КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ

УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Кафедра системного аналізу та теорії прийняття рішень

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

за спеціальністю 124 «Системний аналіз»

на тему:

ЗАСТОСУВАННЯ ТЕОРІЇ ГРАФІВ ДЛЯ ШИФРУВАННЯ ДАННИХ

Студента 4 курсу

Афоніна Дмитра Ігоровича

Науковий керівник:

професор, доктор фіз.-мат. наук

Івохін Є.В.

Робота заслухана на засіданні кафедри системного аналізу та теорії прийняття рішень та рекомендована до захисту в ДЕК, протокол № __10__від_07.06.2022 р.

Завідувач кафедри системного аналізу

професор, доктор фіз.-мат. наук Наконечний О Г.

Київ – 2022

Зміст

Анотації **2**

Умовні позначення та скорочення 2

Реферат 3

Вступ **4**

Розділ 1 Правильне розфарбування графів у протоколах нульового розголосу **6**

1.1. Теоретичні відомості 6

1.2. Реалізація 12

1.3. Проміжні висновки 12

Розділ 2 Пошук гамільтонового циклу в графі **17**

1.1. Теоретичні відомості 17

2.2. Реалізація 20

2.3. Проміжні висновки 21

Висновки **25**

Список використаних джерел **27**

Додатки **29**

Анотації

Дипломна робота складається зі вступу, 2 розділів, висновків, списку використаних джерел (20 найменувань). Загальний обсяг роботи становить 48 сторінок, основний текст роботи викладено на 23 сторінці.

Ключові слова: алгоритм шифрування, RSA, теорія графів, задача правильного розфарбування графа у протоколах з нульовим рішенням, задача пошуку гамільтонового циклу.

Умовні позначення та скорочення

- G – позначення граф
- V – множина вершин графа G
- v – вершина з множини вершин V
- E – множина ребер графа G
- k – кількість кольорів використаних для розфарбування графа G
- R – червоний колір
- Gr – зелений колір
- B – синій колір
- RSA – криптографічний алгоритм з відкритим ключем, що базується на обчисленій складності задачі факторизації великих цілих чисел
- p – просте число, використовується для RSA
- q – просте число, використовується для RSA
- n – число, використовується для RSA, входить як у відкриту так і у секретну частини ключа
- e (encryption) – відкрита експонента ключа для RSA
- d (decryption) – секретна експонента ключа для RSA
- mod - залишок від ділення
- a – довільний параметр
- N – кількість ітерацій алгоритму (також використовується для повноти)

- P – імовірність обходу методу шифрування (також використовується для коректності)
- G' – граф, ізоморфний графу G
- H – матриця графу, що ізоморфний графу G
- F – матриця, що є зашифрованою для матриці H
- i – рядок матриці
- j – стовпець матриці
- m – повідомлення для шифрування
- m_s – послідовність, яка складає повідомлення для шифрування
- c – криптотекст
- c_s – послідовність, яка складає криптотекст

Реферат

В роботі було поєднано метод шифрування RSA з двома задачами теорії графів: розфарбування графу та пошуку гамільтонового циклу, – виведено алгоритми для цих двох варіантів поєднань та реалізовано ці алгоритми. Реалізація була здійснена мовою програмування Python. На основі написаних програм та показників отриманих в результаті їх виконання було проведено аналіз цих двох рішень та їх порівняння.

Вступ

У сучасному світі важко недооцінити цінність інформації. Завдяки розвитку технологій стало можливим оброблювати великі об’єми інформації, яка потім використовується у відповідних сферах життя. З усвідомленням важливості інформації постає задача її збереження та захисту. Останнє і буде об’єктом цієї роботи.

Здавна люди намагались використати для розв'язання завдання захисту інформації найрізноманітніші методи, і одним з них була тайнопис - вміння складати повідомлення таким чином, щоб його зміст був недоступний нікому крім посвячених у таємницю. Є свідчення того, що мистецтво тайнопису зародилося ще в до античні часи. Протягом всієї своєї багатовікової історії, аж до зовсім недавнього часу, це мистецтво служило небагатьом, в основному верхівці суспільства, не виходячи за межі резиденцій глав держав, посольств і - звичайно ж - розвідувальних місій. І лише кілька десятиліть тому все змінилося докорінно - інформація набула самостійну комерційну цінність і стала широко поширеним, майже звичайним товаром. Її виробляють, зберігають, транспортують, продають і купують, а значить - крадуть і підробляють - і, отже, її необхідно захищати. Сучасне суспільство все більше стає інформаційно-обумовленим, успіх будь-якого виду діяльності все сильніше залежить від володіння певними відомостями і від відсутності їх у конкурентів. І чим сильніше проявляється зазначений ефект, тим більше потенційні збитки від зловживань в інформаційній сфері, і тим більше потреба в захисті інформації.

Серед усього спектру методів захисту даних від несанкціонованого доступу особливе місце займають методи шифрування даних. На відміну від інших методів вони спираються лише на властивості самої інформації і не використовують властивості її матеріальних носіїв, особливості вузлів її обробки, передачі та зберігання. Шифрування даних – оборотне перетворення даних, з метою приховання інформації.

Широке застосування комп'ютерних технологій та постійне збільшення обсягу інформаційних потоків викликає постійне зростання інтересу до криптографії. Останнім часом збільшується роль програмних засобів захисту інформації, просто модернізованих не потребують великих фінансових витрат в порівнянні з апаратними криптосистемами.

Однією із основних задач, що виникають при шифруванні/розшифруванні інформації є проведення операцій з множинами (даних, ключів тощо). Саме тому вбачаю можливим розглянути методології теорії графів для вирішення проблеми шифрування даних. Теорія графів - один із основних інструментів у математиці, що використовується для представлення множин та зв'язків між ними. Тому у криптографії графи мають достатньо широкі способи застосування. У своїй роботі я розглянув задачі теорії графів, що лежать в основі криптографічного протоколу RSA.

У даній роботі розглянуто способи імплементації теорії графів для шифрування даних. Було розглянуто дві задачі з теорії графів, а саме: правильне розфарбування графів у протоколах нульового розголошення та пошук гамільтонового циклу в графі – та об’єднано з методом шифрування RSA. Зроблено висновки що до прийнятності та ефективності отриманих алгоритмів та зроблено їх порівняння.

Розділ 1

Правильне розфарбування графів у протоколах нульового розголосу

1.1. Теоретичні відомості

Граф – це сукупність об’єктів зі зв’язками між ними.

Планарний граф – граф, що може бути зображений на площині без перетину ребер.

Регулярний граф – граф, степені всіх вершин якого рівні.

Хроматичне число графу – мінімальна кількість кольорів, в які можна розфарбувати вершини графу таким чином, щоб кінці будь-якого ребра мали різні кольори.

Задача правильного розфарбування графу є одною із найвідоміших проблем розмітки графів.

NP-повнота – властивість задачі, яка належить до класу NP (клас складності, до якого належать задачі, що можна розв’язати недетермінованими алгоритмами за поліноміальний час).

1.1.1. Постановка задачі та проблематика розв’язання

Дано граф $G = (V, E)$ і кількість хроматичних кольорів k (хроматичне число). Чи існує таке розфарбування вершин графа G, за якого жодні дві вершини, з'єднані ребром, не були б розфарбовані одним кольором?

Пошук хроматичного числа, при якому дане завдання можна розв'язати, входить до класу NP-повних.

Теорема 1.1. k-розфарбованість є NP-повною для $k > 2$. Ця проблема, визначення чи даний граф є k-розфарбованим для даного k, є NP-повною. [3, ст. 2]

Теорема 1.2. Неповторне розфарбування – NP-повне. [3, ст. 2]

Теорема 1.3. 3-розфарбованість планарного графу з кількістю вузлів не більше 4 є NP-повною. [3, ст. 3]

Теорема 1.4. 3-розфарбованість 4-регулярного графу є NP-повною. [3, ст. 3]

Зважаючи на вищенаведені теореми маємо: при $k = 3$ у разі планарного 4-регулярного графа довести розфарбовуваність або її відсутність також можна не

швидше за поліноміальний час. Тому навіть 3-розфарбовуваність графу має великий інтерес у криптографії і має практичне застосування, наприклад, протоколах нульового розголошення.

1.1.2. Алгоритм

Запропонуємо алгоритм нульового розголошення на основі 3-розмальовування з використанням поширеного протоколу RSA.

Шифрування – алгоритмічне перетворення даних, яке виконується з метою одержання шифрованого тексту; обернене перетворення даних, з метою приховання інформації. Зашифровувач – виконує шифрування.

Розшифровування – процес санкціонованого перетворення зашифрованих даних у придатні для читання. Розшифровувач – виконує розшифровування.

Дешифрування – процес несанкціонованого отримання інформації з зашифрованих даних, при цьому ключ дешифрування невідомий.

Нехай сторона, що доводить, далі зашифровувач, знає правильне забарвлення в три кольори $\{R, Gr, B\}$ у фіксованого планарного 4-регулярного графа G. Вона хоче довести контролюючій стороні, далі розшифровувач, що знає правильний розв'язок (структура графа G йому відома).

Крок 1 (зашифровувач). Зашифровувач вибирає випадкову перестановку кольорів (наприклад: $\{R, Gr, B\} \rightarrow \{ Gr, B, R \}$) і таким чином перефарбовує граф. Вочевидь, що правильність розфарбування не порушиться.

Крок 2 (зашифровувач). Для кожної вершини зашифровувач формує випадкове число r і замінює в ньому два молодші біти зважаючи на колір вершини, для якої створено r (наприклад: $00(R), 10\ (Gr), 01\ (B)$).

Крок 3 (зашифровувач). Для кожної вершини зашифровувач формує параметри для RSA:

$$\forall v \in V \quad \exists p_v, q_v, \quad n_v = p_v q_v, \quad e_v, d_v \qquad (1.1)$$

і обчислює

$$z_v = {r_v}^{e_v} mod n_v \quad (1.2)$$

для кожної вершини посилає розшифровувачу значення

$$n_v, d_v, z_v. \qquad (1.3)$$

Крок 4 (розшифровувач). Тепер розшифровувач випадково вибирає у графі G одне ребро

$$\exists(v_1, v_2) \in E \qquad (1.4)$$

і повідомляє зашифровувачу дані про це ребро.

Крок 5 (зашифровувач). У відповідь зашифровувач надсилає два числа:

$$e_{v_1}, e_{v_2}. \quad (1.5)$$

Крок 6 (розшифровувач). На основі яких розшифровувач обчислює кольори вершин, які з'єднані обраним ребром:

$$z'_{v_1} = {z_{v_1}}^{d_{v1}} mod n_{v_1} = r_{v_1}, \quad (1.6)$$

$$z'_{v_2} = {z_{v_2}}^{d_{v2}} mod n_{v_2} = r_{v_2}. \quad (1.7)$$

Крок 7 (розшифровувач). Якщо два молодших біти обчислених $r_{v_i}, i = \overline{1,2}$ збігаються, то кольори вершин, з'єднаних ребром, однакові - це означає обман, зв'язок переривається. У протилежному випадку – імовірність брехні зменшується на деяке невелике число і алгоритм переходить до кроку 1 і повторюється до тих пір, поки імовірність брехні не зменшиться до вагомо малого значення.

1.1.3. Імовірність обходу алгоритму

Число ітерацій N можна встановити через число ребер і довільний параметр:

$$N = a|E|, \qquad a > 0, \qquad G = (V, E) \qquad (1.8)$$

Отже, цей алгоритм задовольняє умову повноти.

Імовірність того, що зашифровувача не викрито, становить:

$$P = \left(1 - \frac{1}{|E|}\right)^N = \left(1 - \frac{1}{|E|}\right)^{a|E|} \leq \left(\exp\left(-\frac{1}{|E|}\right)\right)^{a|E|} = \exp(-a) \xrightarrow[a\to+\infty]{} 0. \qquad (1.9)$$

1.1.4. Умови нульового розголошення

Алгоритм з нульовим розголошенням має три основні умови:

1. Повнота.
2. Коректність.
3. Нульове розголошення.

Важливо, що цей алгоритм задовольняє умові нульового розголошення. Через те, що:

1. Формула (1.8).
2. Формула (1.9).
3. На кожній ітерації колірна палітра змінюється, розшифровувач, маючи розмальовку кожного ребра, не зможе відтворити правильне забарвлення всього графу.

1.1.5. RSA

Алгоритм RSA (за першими літерами прізвищ його творців Rivest-Shamir-Adleman) заснований на властивостях простих чисел (причому дуже великих). Є прикладом асиметричного методу шифрування даних.

1.1.5.1. Опис алгоритму

Для початку виберемо два дуже великих простих числа (великі вихідні числа потрібні для побудови великих криптостійких ключів. Визначимо параметр n як результат перемноження p і q. Виберемо велике випадкове число і назвемо його e, причому воно повинно бути взаємно простим з результатом множення

$$\varphi(\mathrm{n}) = (\mathrm{p}-1)(q-1). \qquad (1.10)$$

Відшукаємо таке число d, для якого правильне співвідношення

$$(ed)mod\big((\mathrm{p}-1)(q-1)\big) = 1 \qquad (1.11)$$

або

$$ed \equiv 1 \ \big(mod\ \varphi(\mathrm{n})\big) \qquad (1.12)$$

(mod - залишок від ділення, тобто якщо e, помножене на d, поділити на $\varphi(\mathrm{n})$, то у залишку отримаємо 1).

Відкритим ключем є пара чисел (e, n) , а закритим – (d, n). При шифруванні вихідний текст розглядається як числовий ряд, і над кожним його числом ми здійснюємо операцію

$$c_s = (m_s{}^e)mod(n). \qquad (1.13)$$

У результаті виходить послідовність c_s, яка і складе криптотекст.

Декодування інформації відбувається за формулою

$$m_s = (c_s{}^d) mod(n). \qquad (1.14)$$

Імовірність розшифрування криптотексту без знання ключа:

$$P = c^{-\frac{1}{e}} \qquad (1.15)$$

де c – розмір криптотексту, а e – розмір відкритої експоненти ключа.

1.1.5.2. Теорія об'єднання алгоритму та методу шифрування

Оскільки, алгоритм перевіряє коректність зашифровувача інформації, то логічно буде використати його на початку.

За умовами RSA для того, щоб розшифровувач міг відправити свої секретні повідомлення, зашифровувач передає свій відкритий ключ (n, e) розшифровувачу через надійний, але не обов'язково секретний маршрут. Секретний ключ d ніколи не розповсюджується.

Отже, зважаючи на алгоритм розфарбування графу пропонуються наступні зміни до нього для об'єднання з RSA:

I. Крок 3 (зашифровувач). Замість надсилання для кожної вершини значення n_v, d_v, z_v, зашифровувач посилає розшифровувачу тільки n_v, z_v; оскільки для зменшення часу на розрахунки використаємо ці ж ключі для шифрування, отже розголошення d_v є неприпустимим зважаючи на вимоги RSA. Більш того не відправлення d_v не вплине вагомо на алгоритм розфарбування графу.

II. Крок 5 (зашифровувач). Разом з e_{v_1}, e_{v_2} зашифровувач надсилає й повідомлення, зашифроване двома вищезгаданими ключами.

III. Крок 6 (розшифровувач). Розшифровувач не приймає повідомлення у разі завеликої імовірність брехні, інакше – отримує.

IV. Крок 7 (розшифровувач). У випадку отримання криптотексту – розшифровує його, потім продовжує виконання алгоритму.

Звідси маємо алгоритм для реалізації:

Нехай сторона, що доводить, далі зашифровувач, знає правильне забарвлення в три кольори $\{R, Gr, B\}$ у фіксованого планарного 4-регулярного графа G. Вона хоче

довести контролюючій стороні, далі розшифровувач, що знає правильний розв'язок (структура графа G йому відома).

Крок 1 (зашифровувач). Зашифровувач вибирає випадкову перестановку кольорів (наприклад: $\{R, Gr, B\} \rightarrow \{ Gr, B, R \}$) і таким чином перефарбовує граф. Вочевидь, що правильність розфарбування не порушиться.

Крок 2 (зашифровувач). Для кожної вершини зашифровувач формує випадкове число r і замінює в ньому два молодші біти зважаючи на колір вершини, для якої створено r (наприклад: $00(R), 10\ (Gr), 01\ (B)$).

Крок 3 (зашифровувач). Для кожної вершини зашифровувач формує параметри для RSA формула (1.1) і обчислює формулу (1.2) для кожної вершини посилає розшифровувачу значення

$$n_v, z_v. \qquad (1.16)$$

Крок 4 (розшифровувач). Тепер розшифровувач випадково вибирає у графі G одне ребро (1.4) і повідомляє зашифровувачу дані про це ребро.

Крок 5 (зашифровувач). У відповідь зашифровувач шифрує повідомлення m (1.13) надсилає два числа та криптотекст:

$$e_{v_1}, e_{v_2}, c. \qquad (1.17)$$

Крок 6 (розшифровувач). Розшифровувач не приймає повідомлення у разі завеликої імовірність брехні, інакше – отримує, а на основі надісланих двох чисел розшифровувач обчислює кольори вершин, які з'єднані обраним ребром формули (1.6), (1.7).

Крок 7 (розшифровувач). Якщо два молодших біти обчислених $r_{v_i}, i = \overline{1,2}$ збігаються, то кольори вершин, з'єднаних ребром, однакові - це означає обман, зв'язок переривається. У протилежному випадку – імовірність брехні зменшується на деяке невелике число і алгоритм переходить до кроку 1 і повторюється до тих пір, поки імовірність брехні не зменшиться до вагомо малого значення. У випадку отримання криптотексту – розшифровує його, потім продовжує виконання алгоритму (продовжуємо зменшувати імовірність брехні поки йде обмін

інформацією; у випадку закінчення інформації – завершуємо алгоритм, якщо виявлено обман – інформація не розшифровується, а програма завершується).

1.2. Реалізація

Розглянемо проблеми реалізації вищенаведеного алгоритму.

Для реалізації роботи буде використана мова програмування Python.

Способами реалізації графу є:

1. Списками вершин та ребер
2. Списками суміжності
3. Матрицею суміжності
4. Матрицею інцидентності.

Зважаючи на особливості алгоритму, буде використано перший варіант подання графу. Окрім того двомірним масивом буде подано розфарбування всіх вершин графу та списком буде подано перелік всіх (3-х) кольорів, що будуть використані. Повідомлення для шифрування подається рядком. Усе вищенаведене є вхідними даними програми.

Випадкові числа будуть вибиратися у наступних межах:

- $p_v \in (100, 200)$
- $q_v \in (100, 200)$
- $e_v \in (1, \varphi(\mathrm{n}))$
- $r_v \in (100, 900)$

Для оцінки роботи програми буде виміряно час на виконання програми.

1.3. Проміжні висновки

Результатом даного розділу є створення програми на основі алгоритму, запропонованого вище, що об'єднав метод шифрування RSA та алгоритм правильного розфарбування графів у протоколах нульового розголосу.

Код програми та результати виконання наведено у додатках А, В, Ґ та Д відповідно.

Імовірність дешифрування цього алгоритму буде становити:

$$P = \exp(-a) \cdot c^{-\frac{1}{e}} \qquad (1.18)$$

Програма була запущена для 3-х різних графів; отримані результати коротко викладено нижче:

- Кількість вершин – 6 (див. Рис. 1.1)

  Загальна кількість ітерацій – 6

  Час виконання кожної ітерації:

  1) 0.04041314125061035 секунди
  2) 0.14381766319274902 секунди
  3) 0.034966230392456055 секунди
  4) 0.035981178283691406 секунди
  5) 0.04498028755187988 секунди
  6) 0.6423666477203369 секунди

  Середній час виконання однієї ітерації: 0.09425251483917237 секунди

  Загальний час виконання програми: 0.9425251483917236 секунди

  Для даного випадку запуску програми (зважаючи на згенеровані ключі, зашифроване повідомлення та вибраний граф):

$$P = 0{,}425007924$$

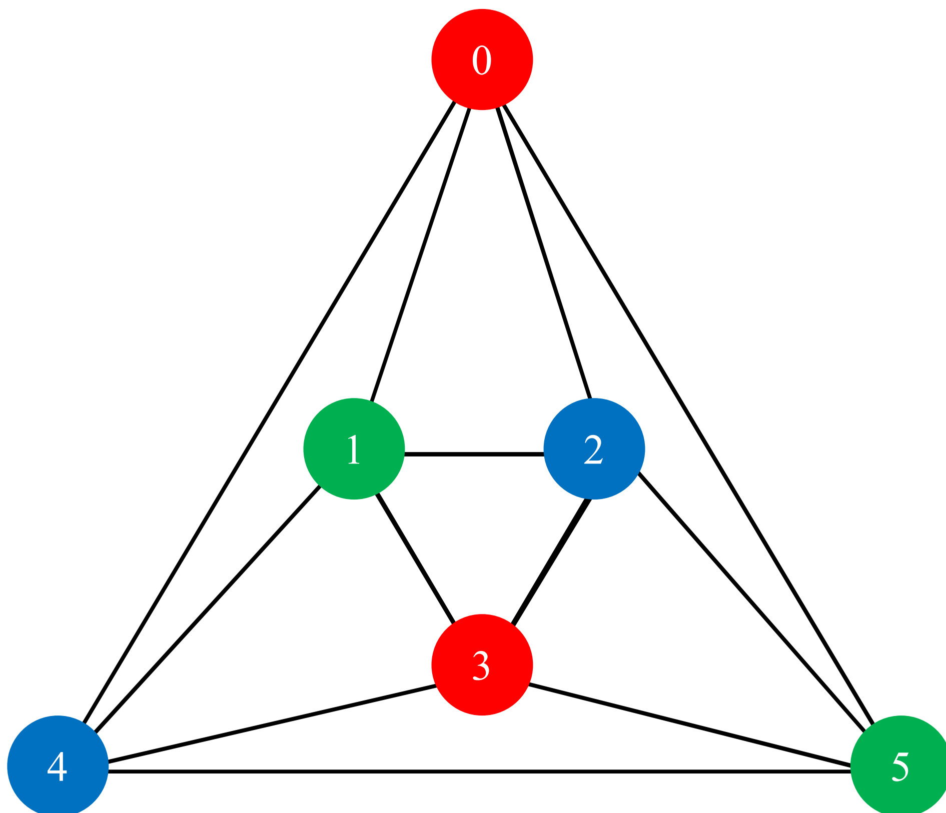

Рис. 1.1. Граф з 6-ма вершинами, використаний у програмі

- Кількість вершин – 8 (див. Рис. 1.2)

  Загальна кількість ітерацій – 6

  Час виконання кожної ітерації:

  1) 0.029007434844970703 секунди
  2) 0.0360407829284668 секунди
  3) 0.02095961570739746 секунди
  4) 0.024099111557006836 секунди
  5) 0.10998916625976562 секунди
  6) 0.38501501083374023 секунди

  Середній час виконання однієї ітерації: 0. 06051112213134766 секунди

  Загальний час виконання програми: 0. 6051111221313477 секунди

Для даного випадку запуску програми (зважаючи на згенеровані ключі, зашифроване повідомлення та вибраний граф):

$$P = 0{,}48773986$$

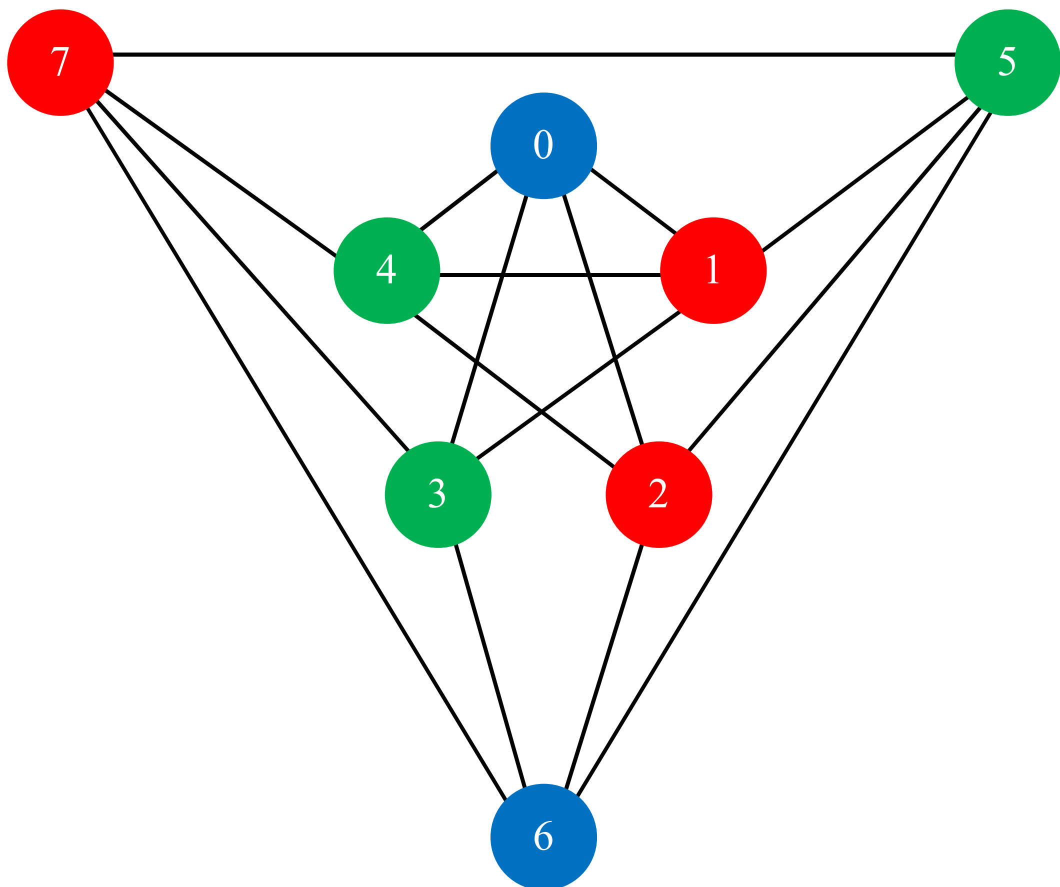

Рис. 1.2. Граф з 8-ма вершинами, використаний у програмі

- Кількість вершин – 10 (див. Рис. 1.3)

  Загальна кількість ітерацій – 6

  Час виконання кожної ітерації:

  1) 0.0892333984375 секунди
  2) 0.05406498908996582 секунди
  3) 0.17218708992004395 секунди
  4) 0.2868931293487549 секунди
  5) 0.15293502807617188 секунди
  6) 0.4285304546356201 секунди

  Середній час виконання однієї ітерації: 0. 11838440895080567 секунди

  Загальний час виконання програми: 1.1838440895080566 секунди

Для даного випадку запуску програми (зважаючи на згенеровані ключі, зашифроване повідомлення та вибраний граф):

$$P = 0{,}506948701$$

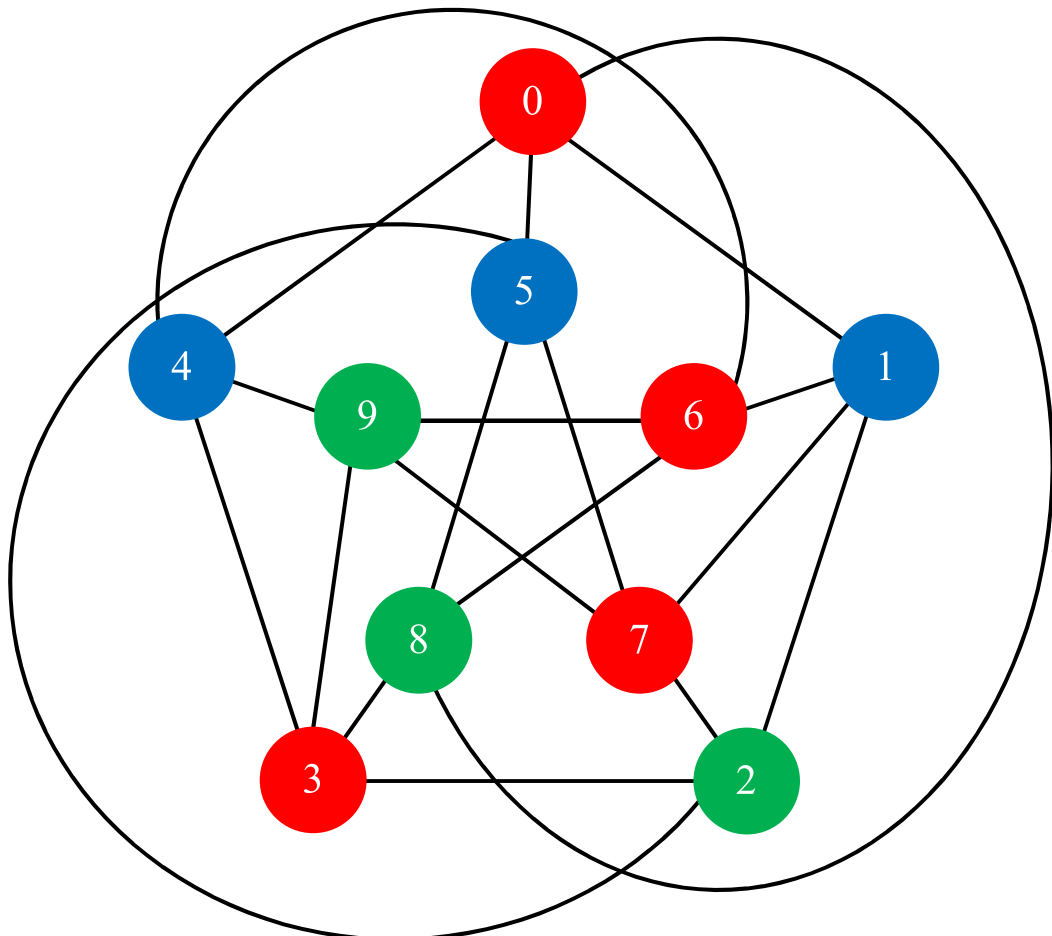

Рис. 1.3. Граф з 10-ма вершинами, використаний у програмі

У висновку бачимо, що хоча і спостерігається зменшення надійності шифру зі збільшенням кількості ребер; це пояснюється тим, що $a = \frac{N}{|E|}$. Тобто для компенсації збільшення ребер потрібно збільшувати кількість ітерації (зменшувати крок зменшення параметру недостовірності). Зважаючи на незначне зростання часу виконання програми такий крок може вважатися повністю обґрунтованим та вигідним для збільшення надійності шифрування.

Розділ 2

Пошук гамільтонового циклу в графі

1.1. Теоретичні відомості

Відповідно до визначення, гамільтонів цикл - замкнутий шлях у графі, що проходить по кожному його ребру рівно один раз. Граф, який має такий цикл, називають гамільтоновим.

Завдання пошуку гамільтонового циклу є NP-повним, оскільки існує недетермінований алгоритм, що може розв'язати дану проблему за поліноміальний час.

Якщо розмальовка графа була прикладом протоколу нульового розголошення, завдання пошуку гамільтонового циклу лежить в основі його сучасної реалізації.

Ізоморфізмом двох графів є бієкція між множинами вершин цих двох графів. Ізоморфні графи – графи між якими присутній ізоморфізм.

2.1.1. Алгоритм

Нехай зашифровувач знає гамільтонів цикл у графі G і хоче довести це розшифровувачу (граф G йому відомий).

Крок 1 (зашифровувач). Зашифровувач будує граф G', ізоморфний графу G. Отриманий граф є матрицею суміжності H.

Крок 2 (зашифровувач). Матриця H шифрується в F за тим самим алгоритмом RSA:

зашифровувач формує параметри для RSA:

$$\exists p, q, \qquad n = pq, \qquad e, d \qquad (2.1)$$

і обчислює

$$F_{ij} = \left(r_{ij} || H_{ij}\right)^e mod n, \qquad (2.2)$$

де r – випадкове число

і пересилає розшифровувачу.

Крок 3 (розшифровувач). Отримавши матрицю F, розшифровувач ставить зашифровувачу одне з двох питань:

1) Який гамільтонів цикл у графа G?

2) Чи дійсно G' ізоморфний G?

Крок 4 (зашифровувач). На запитання (1) зашифровувач розшифровує в F ребра, відповідні гамільтонову циклу в G'. На питання (2) зашифровувач розшифровує граф G' повністю, а також надає перестановки, за допомогою яких з G вийшов G'.

Крок 5 (розшифровувач). Отримавши відповідь, розшифровувач перевіряє правильність розшифровки шляхом повторного шифрування G і порівняння з F. Таким чином, він переконується або в коректності гамільтонова циклу G', або в ізоморфності графів G і G'. Далі, якщо обману зафіксовано не було, збільшити ступінь довіри і перейти до кроку 1.

2.1.2. Імовірність обходу алгоритму

Число ітерацій N можна встановити через число ребер і довільний параметр:

$$N = a|E|,\ a > 0,\ G = (V, E) \qquad (2.3)$$

Отже, цей алгоритм задовольняє умову повноти.

Імовірність обману на N ітераціях не перевищує:

$$P = 2^{-N} \to 0. \qquad (2.4)$$

2.1.3. Умови нульового розголошення

Зазначимо важливі аспекти цього протоколу:

По-перше, навіщо зашифровувачу будувати ізоморфний граф?

Відповідь: без цього розшифровувач, отримавши відповідь на запитання (1), дізнався б про гамільтонів цикл у вихідному графі G, що суперечить умові нульового розголошення.

По-друге, навіщо зашифровувач кодує матрицю суміжності графа G конкатенаціями довільних чисел $(r_{ij}||H_{ij})$?

Відповідь: шифр RSA не дозволяє закодувати 0 або 1, але їх заміна на два довільні інші числа - теж ненадійно (можна легко провести бієкцію між ними і нулем з одиницею). Конкатенація з випадковими r вирішить цю проблему. Більш того за властивістю RSA (він приховує парність вихідного числа) інформація про

молодші біти коефіцієнтів матриці суміжності H буде важко розшифрувати. Вони свідчать про наявність або відсутність ребра між вершинами.

По-третє, навіщо розшифровувач ставити два питання, а не обмежитися лише одним із них?

Відповідь: у такому разі зашифровувач змогла б згенерувати свій власний граф з уже наявним гамільтоновим циклом і, очевидно, розшифровувач не виявить жодних протиріч у тому, що цикл існує і коректний.

По-четверте, чому розшифровувач не може поставити одночасно два запитання?

Відповідь: наданої інформації буде достатньо, щоб розшифровувач зміг сам відновити гамільтонів цикл у вихідному графі, а це суперечить властивості нульового розголошення.

Цей алгоритм задовольняє умові нульового розголошення:

1. Формула (2.3) (повнота).
2. Формула (2.4) (коректність).
3. Зважаючи на вище розглянуті пункти та те, що пара питань утворює бінарне дерево результатів, отримуємо нульове розголошення.

2.1.4. Теорія об’єднання алгоритму та методу шифрування RSA

Оскільки, алгоритм перевіряє коректність зашифровувача інформації, то логічно буде використати його на початку.

Зважаючи на алгоритм пошуку гамільтонового циклу пропонуються наступні зміни до нього для об’єднання з RSA:

I. Крок 4 (зашифровувач). Зашифровувач надсилає повідомлення, зашифроване вищезгаданим ключем.

II. Крок 5 (розшифровувач). Розшифровувач не приймає повідомлення у разі завеликої імовірності брехні, інакше – отримує та розшифровує.

Звідси маємо алгоритм для реалізації:

Нехай зашифровувач знає гамільтонів цикл у графі G і хоче довести це розшифровувачу (граф G йому відомий).

Крок 1 (зашифровувач). Зашифровувач будує граф G', ізоморфний графу G. Отриманий граф є матрицею суміжності H.

Крок 2 (зашифровувач). Матриця H шифрується в F за тим самим алгоритмом RSA:
зашифровувач формує параметри для RSA формула (2.1) і обчислює формулу (2.2) і пересилає розшифровувачу.

Крок 3 (розшифровувач). Отримавши матрицю F, розшифровувач ставить зашифровувачу одне з двох питань:

1) Який гамільтонів цикл у графа G?
2) Чи дійсно G' ізоморфний G?

Крок 4 (зашифровувач). На запитання (1) зашифровувач розшифровує в F ребра, відповідні гамільтонову циклу в G'. На питання (2) зашифровувач розшифровує граф G' повністю, а також надає перестановки, за допомогою яких з G вийшов G'. Зашифровувач вище використаним ключем шифрує та надсилає повідомлення.

Крок 5 (розшифровувач). Отримавши відповідь, розшифровувач перевіряє правильність розшифровки шляхом повторного шифрування G і порівняння з F. Таким чином, він переконується або в коректності гамільтонова циклу G', або в ізоморфності графів G і G'. Розшифровувач не приймає зашифровувача повідомлення у разі завеликої імовірності брехні, інакше – отримує та розшифровує. Далі, якщо обману зафіксовано не було, збільшити ступінь довіри і перейти до кроку 1. Продовжуємо зменшувати імовірність брехні поки йде обмін інформацією; у випадку закінчення інформації – завершуємо алгоритм, якщо виявлено обман – інформація не розшифровується, а програма завершується.

2.2. Реалізація

Зважаючи на особливості алгоритму, подання графу буде виконано за допомогою матриці суміжності. Окрім того матрицею суміжності буде подано гамільтонів шлях(граф) графу. Також списком буде дано перелік вершин графу.

Повідомлення для шифрування подається рядком. Усе вищенаведене є вхідними даними програми.

Реалізацією пошуку ізоморфного графу буде перенумерування вершин, зберігаючи їх первинні зв'язки.

Випадкові числа будуть вибиратися у наступних межах:

- $p_v \in (100, 200)$
- $q_v \in (100, 200)$
- $e_v \in (1, \varphi(\mathrm{n}))$
- $r_v \in (100, 900)$

Для оцінки роботи програми буде виміряно час на виконання програми.

2.3. Проміжні висновки

Результатом даного розділу є створення програми на основі алгоритму, запропонованого вище, що об’єднав метод шифрування RSA та алгоритм пошуку гамільтонового циклу у протоколах нульового розголосу.

Код програми та результати виконання для різних графів наведено у додатках Б, Е, Ж та З відповідно.

Імовірність обходження цього алгоритму буде становити:

$$P = 2^{-N} \cdot c^{-\frac{1}{e}} \qquad (2.5)$$

Програма була запущена для 3 різних графів (використані ті самі графи, що і для програми у розділі 1), отримані результати коротко викладено нижче:

- Кількість вершин – 6 (див. Рис. 2.1)

  Загальна кількість ітерацій – 6

  Час виконання кожної ітерації:

  1) 0.20483994483947754 секунди
  2) 0.11593079566955566 секунди
  3) 0.1712498664855957 секунди
  4) 0.05589485168457031 секунди
  5) 0.14416885375976562 секунди
  6) 0.8728587627410889 секунди

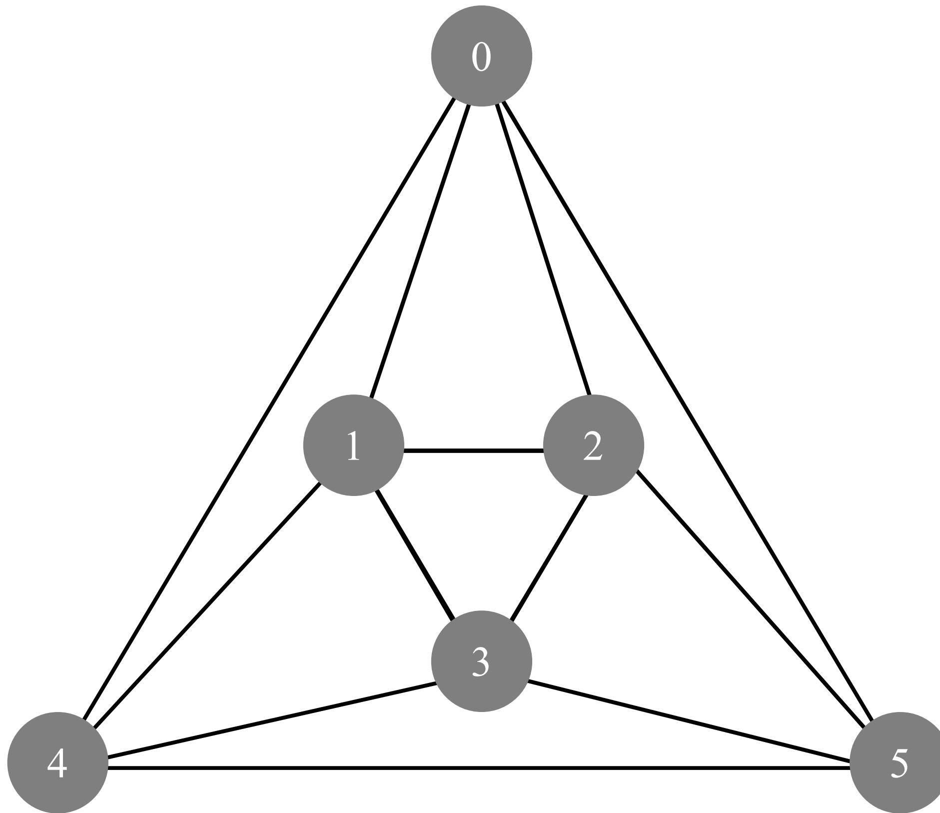

Рис. 2.1. Граф з 6-ма вершинами, використаний у програмі

Середній час виконання однієї ітерації: 0.15659425258636475 секунди

Загальний час виконання програми: 1.5659425258636475 секунди

Для даного випадку запуску програми (зважаючи на згенеровані ключі, зашифроване повідомлення та вибраний граф):

$$P = 0{,}0209604018$$

- Кількість вершин – 8 (див. Рис. 2.2)

  Загальна кількість ітерацій – 6

  Час виконання кожної ітерації:

  1) 0. 11034893989562988 секунди
  2) 0. 2627255916595459 секунди
  3) 0. 1378617286682129 секунди
  4) 0. 27170324325561523 секунди
  5) 0. 5913295745849609 секунди
  6) 0. 189100980758667 секунди

  Середній час виконання однієї ітерації: 0.1563070058826318 секунди

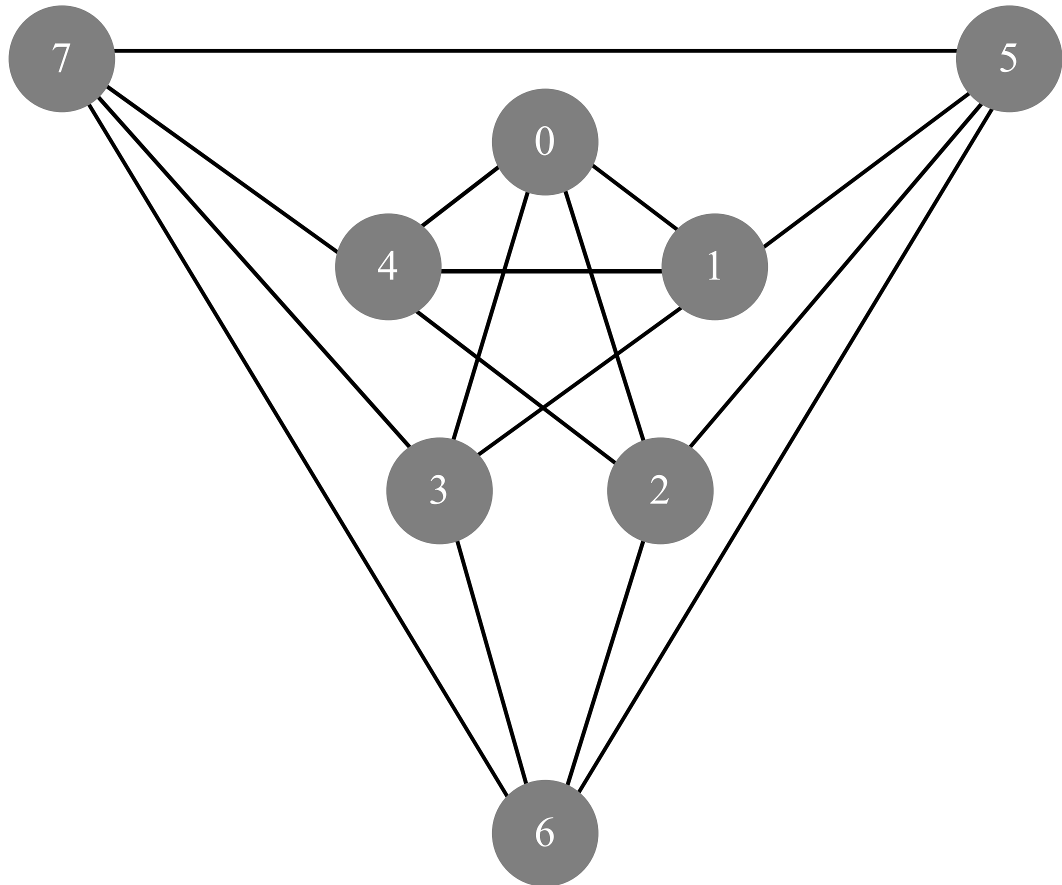

Рис. 2.2. Граф з 8-ма вершинами, використаний у програмі

Загальний час виконання програми: 1. 5630700588226318 секунди

Для даного випадку запуску програми (зважаючи на згенеровані ключі, зашифроване повідомлення та вибраний граф):

$$P = 0{,}015745547$$

- Кількість вершин – 10 (див. Рис. 2.3)

Загальна кількість ітерацій – 6

Час виконання кожної ітерації:

1) 0.004001140594482422 секунди
2) 0.7387039661407471 секунди
3) 0.02500176429748535 секунди
4) 0.5810754299163818 секунди
5) 0.07103800773620605 секунди
6) 0.9926769733428955 секунди

Середній час виконання однієї ітерації: 0.2412497282028198 секунди

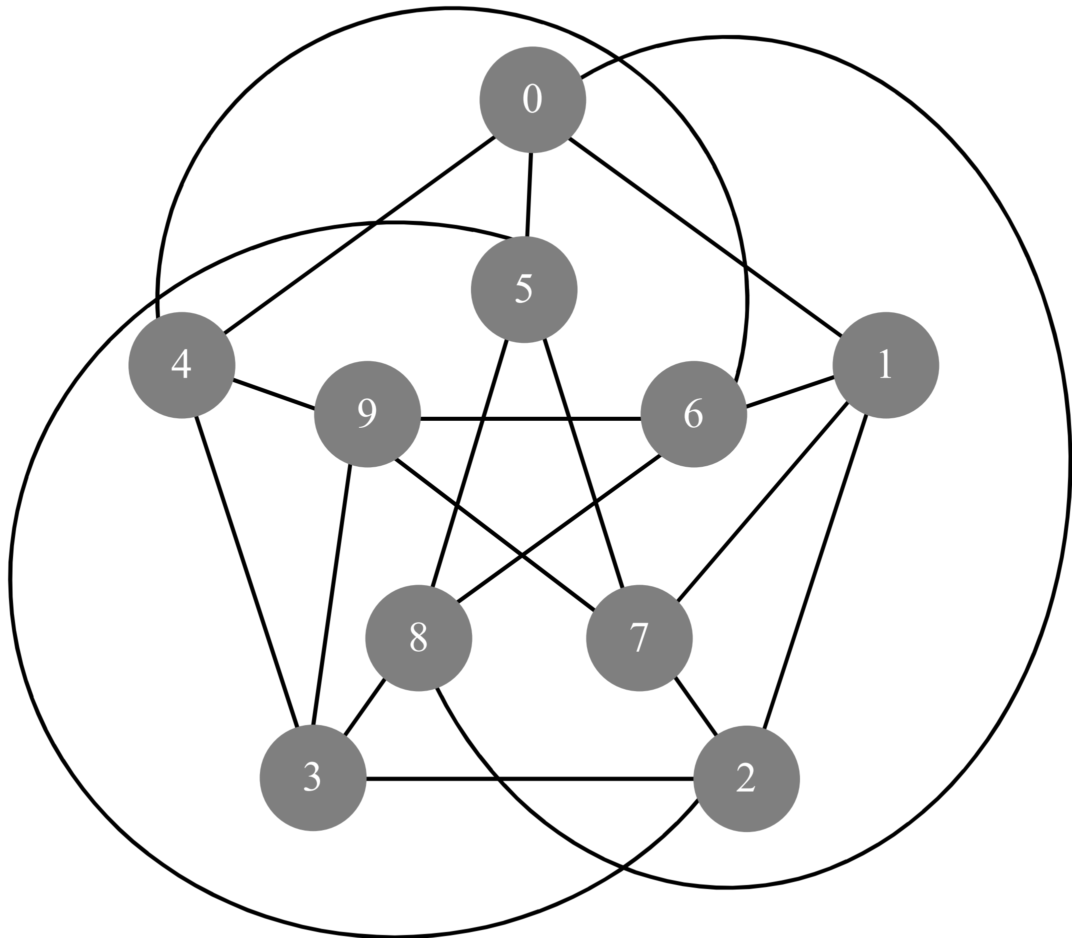

Рис. 2.3. Граф з 10-ма вершинами, використаний у програмі

Загальний час виконання програми: 2.4124972820281982 секунди

Для даного випадку запуску програми (зважаючи на згенеровані ключі, зашифроване повідомлення та вибраний граф):

$$P = 0{,}0106633413$$

У висновку фіксуємо значне зниження ймовірності дешифрування з використанням графів з більшою кількістю вершин, але аналогічно зростає і час виконання програми. Тому в рамках цього алгоритму є важливим визначення оптимальної кількості вершин графу для кожного конкретного випадку шифрування (залежить від потужностей пристрою та завдань у рамках яких вирішується завдання шифрування даних).

Висновки

В результаті було реалізовано алгоритми зазначені в розділах 1 та 2 (код програм надано в додатках). Та отримано показники що до часу виконання та якості кожного методу шифрування.

Зробимо наступні порівняння двох програм (табл. 1).

На основі отриманих даних можемо зробити висновок, що поєднання RSA та гамільтонового циклу є більш вдалим ніж поєднання RSA та розфарбування графу, оскільки перша програма дає більш надійне рішення ніж друге (зі збільшенням кількості ребер і зі сталою кількістю ітерацій надійність методу з розфарбуванням буде погіршуватись, а у випадку з гамільтоновим циклом – ні), незважаючи на більш довгий час виконання програми.

Тим не менш обидві програми є самодостатніми та можуть бути використані для шифрування даних та мають прикладне використання.

Таблиця 1

Порівняльна таблиця реалізованих методів шифрування

Метод шифрування	Граф	Час на одну ітерацію (секунд)	Загальний час виконання (секунд)	P (формула)	P (значення)
RSA + розфарбу-вання графів	Рис. 1.1	0.094250	0.94253	$\exp\left(-\frac{N}{\lvert E\rvert}\right) \cdot c^{-\frac{1}{e}}$	0,42500
	Рис. 1.2	0.060511	0.60511		0,48770
	Рис. 1.3	0.118384	1.18384		0,50695
RSA + гамільтонів цикл	Рис. 1.1	0.156594	1.56594	$2^{-N} \cdot c^{-\frac{1}{e}}$	0,020960
	Рис. 1.2	0.156307	1.56307		0,015746
	Рис. 1.3	0.241249	2.41249		0,010663

Потрібно відмітити, що в обох випадках імовірність обходження алгоритму є прямо пропорційна до довжини даних, що шифруються. Саме тому для більш надійного шифрування інформації рекомендується розбивати завеликі обсяги інформації на менші блоки і шифрувати кожен із них окремо різними ключами. Це гарантує збільшення захищеності даних.

Також варте уваги те, що, незважаючи на погане шифрування великих об’ємів даних, обидва методи взагалі не дозволяють шифрувати числа 0 та 1 та достатньо погано будуть шифруватись будь-які набори чисел від 0 до 10 (тут мається на увазі спроба зашифрувати безпосередньо деяку послідовність таких чисел; у випадку їх поодинокої появи у тексті або наборі чисел ця проблема не буде актуальною). На жаль, нема простого методу обійти дану проблему, а для усунення недоліку потрібно внести значні специфічні зміни у код, що зроблять надскладним (або навіть неможливим) шифрування будь-яких інших даних. Тому даний метод не рекомендується для шифрування таких даних.

На додачу потрібно відмітити, що хоча виведені алгоритми удосконалюють RSA, вони не вирішують вразливості цього методу шифрування. Саме тому потрібно враховувати, що інформація може бути дешифрована тими самими методами, що і алгоритм RSA, хоча і за довший час. Для більш конкретних результатів потрібно робити додаткове дослідження.

Список використаних джерел

1. Шнайер Б «Прикладная криптография» 2-е издание «Протоколы, алгоритмы, исходные тексты на языке Си»
2. Сергей Панасенко «Алгоритмы шифрования»
3. David P. Dailey Note «Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete» 8 February 1979 (https://www.sciencedirect.com/science/article/pii/0012365X80902368?via%3Dihub)
4. Algorithms. Construction and analysis Stein Clifford, Rivest Ronald L.
5. Cormen Thomas H.; Leiserson Charles E.; Rivest Ronald L. «Introduction to Algorithms» (first ed.) 1990
6. Ю.А. Тарнавський «Технології захисту інформації»
7. Calderbank, Michael "The RSA Cryptosystem: History, Algorithm, Primes" (2007-08-20)
8. Jim Sauerberg. "From Private to Public Key Ciphers in Three Easy Steps"
9. Margaret Cozzens and Steven J. Miller. "The Mathematics of Encryption: An Elementary Introduction"
10. DeLeon Melissa "A study of sufficient conditions for Hamiltonian cycles" (2000)
11. Ore Øystein "Note on Hamilton circuits" The American Mathematical Monthly (1960)
12. Pósa L. "A theorem concerning Hamilton lines" (1962)
13. Thomason A. G. "Hamiltonian cycles and uniquely edge colourable graphs" (1978)
14. Babai László "Graph isomorphism in quasipolynomial time [extended abstract]" (2016)
15. Babai László "Group, graphs, algorithms: the graph isomorphism problem" (2018)
16. WikiPedia,Інтернет-енциклопедія. RSA (https://en.wikipedia.org/wiki/RSA_(cryptosystem))
17. WikiPedia,Інтернет-енциклопедія. Graph coloring (https://en.wikipedia.org/wiki/Graph_coloring#Chromatic_number)
18. WikiPedia,Інтернет-енциклопедія. Hamiltonian path (https://en.wikipedia.org/wiki/Hamiltonian_path)

19. WikiPedia,Інтернет-енциклопедія. Hamiltonian path problem (https://en.wikipedia.org/wiki/Hamiltonian_path_problem)
20. WikiPedia,Інтернет-енциклопедія. Isomorphism (https://en.wikipedia.org/wiki/Graph_isomorphism)

Додатки

Додаток А.

Код програми RSA + розфарбування графів

```
import numpy as np
import random
import sympy
import time


# change chosen bit to 1
def bit_1(value, bit):
    return value | (1 << bit)


# change chosen bit to 0
def bit_0(value, bit):
    return value & ~(1 << bit)


# reminder of the division for RSA
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


# find privat exponent for RSA while having public exponent and phi
def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
        temp1 = temp_phi // e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2 - temp1 * x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi


# generate keys and exponents for RSA
def generate_keypair():
    p = sympy.randprime(100, 200)
    q = sympy.randprime(100, 200)
    n = p * q
    phi = ((p - 1) * (q - 1))
```

```
    e = random.randrange(1, phi)

    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)

    d = multiplicative_inverse(e, phi)

    return (e, n), (d, n), n, e, d


# RSA encryption
def encrypt(pk, plaintext):
    key, n = pk
    cipher = [(ord(char) ** key) % n for char in plaintext]
    return cipher


# RSA decryption
def decrypt(pk, ciphertext):
    key, n = pk
    plain = [chr((char ** key) % n) for char in ciphertext]
    return ''.join(plain)


# 1-3 steps| encryptor
def sender_first(coloration, color):

    # step 1
    color_replacement = np.random.choice(color, size=len(color), replace=False)

    send_arr = []
    send_pack = []

    for vertex1 in coloration:
        if vertex1[1] == "R":
            send_arr.append([vertex1[0], color_replacement[0]])
        elif vertex1[1] == "G":
            send_arr.append([vertex1[0], color_replacement[1]])
        elif vertex1[1] == "B":
            send_arr.append([vertex1[0], color_replacement[2]])
        else:
            print("Color: ERROR")

    # step 2
    for vertex1 in send_arr:

        r = random.randint(100, 900)

        if vertex1[1] == "R":
            r = bit_0(r, 0)
            r = bit_0(r, 1)
        elif vertex1[1] == "G":
            r = bit_0(r, 0)
            r = bit_1(r, 1)
        elif vertex1[1] == "B":
            r = bit_1(r, 0)
            r = bit_0(r, 1)
        else:
            print("Bits: ERROR")
        vertex1.append(r)
```

```
        # step 3
        # generate RSA components
        # public, private, n, e, d = generate_keypair()
        vertex1.append(generate_keypair())
        # count Z
        Z = (vertex1[2] ** (int(vertex1[3][4]))) % (int(vertex1[3][2]))
        vertex1.append(Z)
        # form send array
        send_pack.append([vertex1[0], vertex1[3][2], Z])

    return send_arr, send_pack


# step 4| decrypter
def recipient_first(send_pack):
    chosen_line = random.choice(E)
    point_1 = chosen_line[0]
    point_2 = chosen_line[1]
    return send_pack, point_1, point_2


# step 5| encryptor
def sender_second(send_arr):
    send_e_1 = 0
    send_e_2 = 0
    d_1 = 0
    d_2 = 0
    n_1 = 0
    n_2 = 0
    for vertex in send_arr:
        if vertex[0] == point1:
            send_e_1 = vertex[3][3]
            d_1 = vertex[3][4]
            n_1 = vertex[3][2]
        if vertex[0] == point2:
            send_e_2 = vertex[3][3]
            d_2 = vertex[3][4]
            n_2 = vertex[3][2]

    return send_e_1, send_e_2, d_1, d_2, n_1, n_2


# step 6| decrypter
def recipient_second(send_pack1, send_e_1, send_e_2, point_1, point_2):
    Z_check1 = (send_pack1[(int(point_1))][2] ** send_e_1) %
send_pack1[(int(point_1))][1]
    Z_check2 = (send_pack1[(int(point_2))][2] ** send_e_2) %
send_pack1[(int(point_2))][1]

    if (bin(Z_check1)[-2] + bin(Z_check1)[-1]) == (bin(Z_check2)[-2] +
bin(Z_check2)[-1]):
        # print("Sender is unreliable")
        return 1
    else:
        # print("All Ok")
        return -2


# input variants
def inpt(tp):
    if int(tp) == 6:
        # Input 1
```

```
        # list of vertices of the graph
        in_V = ["0", "1", "2", "3", "4", "5"]
        # list of edges of the graph
        in_E = ["01", "02", "04", "05", "12", "13", "14", "23", "25", "34", "35",
"45"]
        # list of coloring of vertices of the graph
        in_coloring = [["0", "R"], ["1", "G"], ["2", "B"], ["3", "R"], ["4", "B"],
["5", "G"]]

    elif int(tp) == 8:
        # Input 2
        # list of vertices of the graph
        in_V = ["0", "1", "2", "3", "4", "5", "6", "7"]
        # list of edges of the graph
        in_E = ["01", "02", "03", "04", "13", "14", "15", "24", "25", "26", "36",
"37", "47", "56", "57", "67"]
        # list of coloring of vertices of the graph
        in_coloring = [["0", "B"], ["1", "R"], ["2", "R"], ["3", "G"], ["4", "G"],
["5", "G"], ["6", "B"], ["7", "R"]]

    elif int(tp) == 10:
        # Input 2
        # list of vertices of the graph
        in_V = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
        # list of edges of the graph
        in_E = ["01", "04", "05", "08", "12", "16", "17", "23", "25", "27", "34",
"38", "39", "46", "49", "57", "58",
                "68", "69", "79"]
        # list of coloring of vertices of the graph
        in_coloring = [["0", "R"], ["1", "B"], ["2", "G"], ["3", "R"], ["4", "B"],
["5", "B"], ["6", "R"], ["7", "R"],
                       ["8", "G"], ["9", "G"]]

    else:
        print("Input ERROR\nAutomatically chosen 6")
        # Input 1
        # list of vertices of the graph
        in_V = ["0", "1", "2", "3", "4", "5"]
        # list of edges of the graph
        in_E = ["01", "02", "04", "05", "12", "13", "14", "23", "25", "34", "35",
"45"]
        # list of coloring of vertices of the graph
        in_coloring = [["0", "R"], ["1", "G"], ["2", "B"], ["3", "R"], ["4", "B"],
["5", "G"]]

    # list of colors of the graph
    in_colors = ["R", "G", "B"]
    # message to encrypt
    in_encryption_data = "Realisation of combination of RSA and coloring graph
methods is successful"

    return in_V, in_E, in_coloring, in_colors, in_encryption_data


# input for user
V, E, coloring, colors, encryption_data1 = inpt(input("What graph? (6, 8 or 10
vertices)\n"))

start_time = time.time()
msg_encr = 0
iteration = 0
test = 10
```

```
# body of encryption-decryption algorithm
while msg_encr == 0:

    # show new iteration start
    iteration += 1

    iteration_start_time = time.time()

    sender_arr, sender_pack = sender_first(coloring, colors)

    recipient_pack, point1, point2 = recipient_first(sender_pack)

    send_e1, send_e2, decryption_1, decryption_2, num1, num2 =
sender_second(sender_arr)

    test += recipient_second(sender_pack, send_e1, send_e2, point1, point2)

    # encryption is done by sender
    encrypted_data1 = encrypt((send_e1, num1), encryption_data1)

    # if sender is confirmed, data will be decrypted
    if test < 0:
        print(f"\nLength of privat exponent : {len(bin(send_e1))}")
        c_size = 0
        c_size += np.array(encrypted_data1).itemsize * len(encrypted_data1)
        print(f"Length of encrypted message : {c_size}")
        print(f"Encrypted message is : {encrypted_data1}")
        print(f"Decrypted message is : {decrypt((decryption_1, num1),
encrypted_data1)}")
        msg_encr += 1

    # show how long iteration has taken
    print("\nIteration ", iteration, " time ", "%s seconds" % (time.time() -
iteration_start_time))

print("\nOverall time", "---%s seconds ---" % (time.time() - start_time))
print("Average iteration time", "---%s seconds ---" % ((time.time() - start_time) /
10))
```

Додаток Б.

Код програми RSA + гамільтонів цикл

```
import numpy as np
import random
import sympy
import time


# change chosen bit to 1
def bit_1(value, bit):
    return value | (1 << bit)


# change chosen bit to 0
def bit_0(value, bit):
    return value & ~(1 << bit)


# reminder of the division for RSA
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
```

```
    return a


# find privat exponent for RSA while having public exponent and phi
def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
        temp1 = temp_phi // e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        = temp2

        x = x2 - temp1 * x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi


# generate keys and exponents for RSA
def generate_keypair():
    p = sympy.randprime(100, 200)
    q = sympy.randprime(100, 200)
    n = p * q
    phi = ((p - 1) * (q - 1))

    e = random.randrange(1, phi)

    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)

    d = multiplicative_inverse(e, phi)

    return (e, n), (d, n), n, e, d


# RSA encryption
def encrypt(pk, plaintext):
    key, n = pk
    cipher = [(ord(char) ** key) % n for char in plaintext]
    return cipher


# RSA decryption
def decrypt(pk, ciphertext):
    key, n = pk
    plain = [chr((char ** key) % n) for char in ciphertext]
    return ''.join(plain)


# encrypt matrix with RSA
```

```
def encrypt_matrix(vertices, matrix, n, e):
    encrypted_matrix = []
    for node1 in vertices:
        encrypted_matrix.append([])
        for p in vertices:
            r = random.randint(100, 900)
            if matrix[node1][p] == 0:
                r = bit_0(r, 0)
                r = bit_0(r, 1)
            elif matrix[node1][p] == 1:
                r = bit_1(r, 0)
                r = bit_0(r, 1)
            f = (r ** e) % n
            encrypted_matrix[node1].append(f)
    return encrypted_matrix


# decrypt matrix with RSA
def decrypt_matrix(vertices, encrypted_matrix, n, d):
    decrypted_matrix = []
    for node1 in vertices:
        decrypted_matrix.append([])
        for p in vertices:
            f = (encrypted_matrix[node1][p] ** d) % n
            decrypted_matrix[node1].append(f)
    return decrypted_matrix


# form binary matrix with last bits
def to_binary(vertices, matrix):
    result = []
    for node3 in vertices:
        result.append([])
        for point3 in vertices:
            result[node3].append(bin(matrix[node3][point3])[-1])
    return result


# find isomorphic graph
def find_iso(matrix, vertices, repl):
    isom_H1 = []
    isom_H2 = []

    for point in repl:
        isom_H2.append(matrix[point])

    for node in vertices:
        isom_H1.append([])
        for point in repl:
            isom_H1[node].append(isom_H2[node][point])
    return isom_H1


# 1-2 steps| encryptor
def sender_first(vertices, matrix):

    # step 1
    # generating random samples without replacement
    = np.random.choice(vertices, size=len(vertices), replace=False)

    # create isomorphism
    H = find_iso(matrix, vertices, replacement)
```

```
    # step 2
    # encrypt matrix
    public, private, n, e, d = generate_keypair()

    F = encrypt_matrix(vertices, H, n, e)

    return replacement, public, private, n, e, d, F


# step 3| decrypter
def recipient_first(F):
    message = random.randint(1, 2)
    return F, message


# step 4| encryptor
def sender_second(message, iso_matrix, vertices, F, n, d, replacement):
    if message == 1:
        G1_repl = find_iso(iso_matrix, vertices, replacement)
        return iso_matrix, 0, 0
    elif message == 2:
        G_decryption = decrypt_matrix(vertices, F, n, d)
        return 0, G_decryption, sender_replacement


# step 5| decrypter
def recipient_second(message, vertices, matrix, G_decryption, replacement, given_G):
    res = -2
    if message == 1:
        for row in vertices:
            for column in vertices:
                if matrix[row][column] == 1 and given_G[row][column] != 1:
                    # print("ERROR for answer 1")
                    res += 1
    elif message == 2:
        check_G = to_binary(vertices, G_decryption)
        check_H = to_binary(vertices, find_iso(given_G, vertices, replacement))
        if check_G != check_H:
            # print("ERROR for answer 2")
            res += 1
    return res


# input variants
def inpt(tp):
    if int(tp) == 6:
        # Input 1
        # list of vertices of the graph
        in_V = [0, 1, 2, 3, 4, 5]
        # adjacency matrix of the graph
        #         0  1  2  3  4  5
        in_G = [[0, 1, 1, 0, 1, 1],  # 0
                [1, 0, 1, 1, 1, 0],  # 1
                [1, 1, 0, 1, 0, 1],  # 2
                [0, 1, 1, 0, 1, 1],  # 3
                [1, 1, 0, 1, 0, 1],  # 4
                [1, 0, 1, 1, 1, 0]]  # 5
        # adjacency matrix of the hamilton path
        #          0  1  2  3  4  5
        in_GH = [[0, 0, 0, 0, 1, 1],  # 0
                 [0, 0, 1, 0, 1, 0],  # 1
                 [0, 1, 0, 1, 0, 0],  # 2
                 [0, 0, 1, 0, 0, 1],  # 3
```

```
               [1, 1, 0, 0, 0, 0],   # 4
               [1, 0, 0, 1, 0, 0]]   # 5

    elif int(tp) == 8:
        # Input 2
        # list of vertices of the graph
        in_V = [0, 1, 2, 3, 4, 5, 6, 7]
        # adjacency matrix of the graph
        #         0  1  2  3  4  5  6  7
        in_G = [[0, 1, 1, 1, 1, 0, 0, 0],   # 0
                [1, 0, 0, 1, 1, 1, 0, 0],   # 1
                [1, 0, 0, 0, 1, 1, 1, 0],   # 2
                [1, 1, 0, 0, 0, 0, 1, 1],   # 3
                [1, 1, 1, 0, 0, 0, 0, 1],   # 4
                [0, 1, 1, 0, 0, 0, 1, 1],   # 5
                [0, 0, 1, 1, 0, 1, 0, 1],   # 6
                [0, 0, 0, 1, 1, 1, 1, 0]]   # 7
        # adjacency matrix of the hamilton path
        #          0  1  2  3  4  5  6  7
        in_GH = [[0, 0, 1, 1, 0, 0, 0, 0],   # 0
                 [0, 0, 0, 1, 0, 1, 0, 0],   # 1
                 [1, 0, 0, 0, 1, 0, 0, 0],   # 2
                 [1, 1, 0, 0, 0, 0, 0, 0],   # 3
                 [0, 0, 1, 0, 0, 0, 0, 1],   # 4
                 [0, 1, 0, 0, 0, 0, 1, 0],   # 5
                 [0, 0, 0, 0, 0, 1, 0, 1],   # 6
                 [0, 0, 0, 0, 1, 0, 1, 0]]   # 7

    elif int(tp) == 10:
        # Input 2
        # list of vertices of the graph
        in_V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        # adjacency matrix of the graph
        #         0  1  2  3  4  5  6  7  8  9
        in_G = [[0, 1, 0, 0, 1, 1, 0, 0, 1, 0],   # 0
                [1, 0, 1, 0, 0, 0, 1, 1, 0, 0],   # 1
                [0, 1, 0, 1, 0, 1, 0, 1, 0, 0],   # 2
                [0, 0, 1, 0, 1, 0, 0, 0, 1, 1],   # 3
                [1, 0, 0, 1, 0, 0, 1, 0, 0, 1],   # 4
                [1, 0, 1, 0, 0, 0, 0, 1, 1, 0],   # 5
                [0, 1, 0, 0, 1, 0, 0, 0, 1, 1],   # 6
                [0, 1, 1, 0, 0, 1, 0, 0, 0, 1],   # 7
                [1, 0, 0, 1, 0, 1, 1, 0, 0, 0],   # 8
                [0, 0, 0, 1, 1, 0, 1, 1, 0, 0]]   # 9
        # adjacency matrix of the hamilton path
        #          0  1  2  3  4  5  6  7  8  9
        in_GH = [[0, 0, 0, 0, 1, 0, 0, 0, 1, 0],   # 0
                 [0, 0, 1, 0, 0, 0, 1, 0, 0, 0],   # 1
                 [0, 1, 0, 1, 0, 0, 0, 0, 0, 0],   # 2
                 [0, 0, 1, 0, 1, 0, 0, 0, 0, 0],   # 3
                 [1, 0, 0, 1, 0, 0, 0, 0, 0, 0],   # 4
                 [0, 0, 0, 0, 0, 0, 0, 1, 1, 0],   # 5
                 [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],   # 6
                 [0, 0, 0, 0, 0, 1, 0, 0, 0, 1],   # 7
                 [1, 0, 0, 0, 0, 1, 0, 0, 0, 0],   # 8
                 [0, 0, 0, 0, 0, 0, 1, 1, 0, 0]]   # 9

    else:
        print("Input ERROR\nAutomatically chosen 6")
        # Input 1
        # list of vertices of the graph
        in_V = [0, 1, 2, 3, 4, 5]
        # adjacency matrix of the graph
```

```
        #         0  1  2  3  4  5
        in_G = [[0, 1, 1, 0, 1, 1],  # 0
                [1, 0, 1, 1, 1, 0],  # 1
                [1, 1, 0, 1, 0, 1],  # 2
                [0, 1, 1, 0, 1, 1],  # 3
                [1, 1, 0, 1, 0, 1],  # 4
                [1, 0, 1, 1, 1, 0]]  # 5
        # adjacency matrix of the hamilton path
        #          0  1  2  3  4  5
        in_GH = [[0, 0, 0, 0, 1, 1],  # 0
                 [0, 0, 1, 0, 1, 0],  # 1
                 [0, 1, 0, 1, 0, 0],  # 2
                 [0, 0, 1, 0, 0, 1],  # 3
                 [1, 1, 0, 0, 0, 0],  # 4
                 [1, 0, 0, 1, 0, 0]]  # 5

    # message to encrypt
    in_encryption_data = "Realisation of combination of RSA and hamiltonian paths
methods is successful"

    return in_V, in_G, in_GH, in_encryption_data


# input for user
V, G, GH, encryption_data = inpt(input("What graph? (6, 8 or 10 vertices)\n"))

start_time = time.time()
msg_encr = 0
iteration = 0
test = 10

# body of encryption-decryption algorithm
while msg_encr == 0:

    # show new iteration start
    iteration += 1

    iteration_start_time = time.time()

    sender_replacement, sender_public, sender_private, sender_n, sender_e, sender_d,
sender_F = sender_first(V, G)

    recipient_F, recipient_message = recipient_first(sender_F)

    sender_reply1, sender_reply2, sender_reply3 = sender_second(recipient_message,
GH, V, sender_F, sender_n, sender_d,
                                                                  sender_replacement)

    test += recipient_second(recipient_message, V, sender_reply1, sender_reply2,
sender_reply3, G)

    # encryption is done by sender
    encrypted_data = encrypt(sender_public, encryption_data)

    # if sender is confirmed, data will be decrypted
    if test < 0:
        print(f"\nLength of privat exponent : {len(bin(sender_e))}")
        c_size = 0
        c_size += np.array(encrypted_data).itemsize * len(encrypted_data)
        print(f"Length of encrypted message : {c_size}")
        print(f"Encrypted message is : {encrypted_data}")
        print(f"Decrypted message is : {decrypt(sender_private, encrypted_data)}")
        msg_encr += 1
```

```
    # show how long iteration has taken
    print("\nIteration ", iteration, " time ", "%s seconds" % (time.time() -
iteration_start_time))

print("\nOverall time", "---%s seconds ---" % (time.time() - start_time))
print("Average iteration time", "---%s seconds ---" % ((time.time() - start_time) /
10))
```

Додаток В.

Результат №1 роботи програми з додатку 1.

C:\ProgramData\Anaconda3\python.exe

C:/Users/Владелец/PycharmProjects/RSA+graph/main.py

What graph? (6, 8 or 10 vertices)

6

Iteration 1 time 0.04041314125061035 seconds

Iteration 2 time 0.14381766319274902 seconds

Iteration 3 time 0.034966230392456055 seconds

Iteration 4 time 0.035981178283691406 seconds

Iteration 5 time 0.04498028755187988 seconds

Length of privat exponent : 16

Length of encrypted message : 296

Encrypted message is : [6013, 1570, 2579, 4039, 18114, 15888, 2579, 23315, 18114, 2696, 13114, 4211, 2696, 1415, 4211, 23430, 2696, 6832, 14375, 18114, 13114, 2579, 23315, 18114, 2696, 13114, 4211, 2696, 1415, 4211, 6013, 4913, 2946, 4211, 2579, 13114, 23345, 4211, 23430, 2696, 4039, 2696, 7078, 18114, 13114, 12920, 4211, 12920, 7078, 2579, 12525, 2997, 4211, 6832, 1570, 23315, 2997, 2696, 23345, 15888, 4211, 18114, 15888, 4211, 15888, 16629, 23430, 23430, 1570, 15888, 15888, 1415, 16629, 4039]

Decrypted message is : Realisation of combination of RSA and coloring graph methods is successful

Iteration 6 time 0.6423666477203369 seconds

Overall time ---0.9425251483917236 seconds ---

Average iteration time ---0.09425251483917237 seconds ---

Process finished with exit code 0

Додаток Ґ.

Результат №2 роботи програми з додатку 1.

C:\ProgramData\Anaconda3\python.exe C:/Users/Владелец/PycharmProjects/RSA+graph/main.py

What graph? (6, 8 or 10 vertices)

8

Iteration 1 time 0.029007434844970703 seconds

Iteration 2 time 0.0360407829284668 seconds

Iteration 3 time 0.02095961570739746 seconds

Iteration 4 time 0.024099111557006836 seconds

Iteration 5 time 0.10998916625976562 seconds

Length of privat exponent : 14

Length of encrypted message : 296

Encrypted message is : [30, 2117, 58, 11704, 9422, 9180, 58, 17718, 9422, 18411, 3149, 12383, 18411, 8514, 12383, 2054, 18411, 15390, 15144, 9422, 3149, 58, 17718, 9422, 18411, 3149, 12383, 18411, 8514, 12383, 30, 4448, 10524, 12383, 58, 3149, 15230, 12383, 2054, 18411, 11704, 18411, 15285, 9422, 3149, 4640, 12383, 4640, 15285, 58, 18498, 5418, 12383, 15390, 2117, 17718, 5418, 18411, 15230, 9180, 12383, 9422, 9180, 12383, 9180, 9677, 2054, 2054, 2117, 9180, 9180, 8514, 9677, 11704]

Decrypted message is : Realisation of combination of RSA and coloring graph methods is successful

Iteration 6 time 0.38501501083374023 seconds

Overall time ---0.6051111221313477 seconds ---

Average iteration time ---0.060511112213134766 seconds ---

Process finished with exit code 0

Додаток Д.

Результат №3 роботи програми з додатку 1.

C:\ProgramData\Anaconda3\python.exe

C:/Users/Владелец/PycharmProjects/RSA+graph/main.py

What graph? (6, 8 or 10 vertices)

10

Iteration 1 time 0.0892333984375 seconds

Iteration 2 time 0.05406498908996582 seconds

Iteration 3 time 0.17218708992004395 seconds

Iteration 4 time 0.2868931293487549 seconds

Iteration 5 time 0.15293502807617188 seconds

Length of privat exponent : 15

Length of encrypted message : 296

Encrypted message is : [1871, 594, 7769, 6930, 20174, 12309, 7769, 16577, 20174, 14246, 3876, 8816, 14246, 6857, 8816, 6134, 14246, 6151, 19873, 20174, 3876, 7769,

16577, 20174, 14246, 3876, 8816, 14246, 6857, 8816, 1871, 7013, 19597, 8816, 7769, 3876, 15296, 8816, 6134, 14246, 6930, 14246, 13674, 20174, 3876, 15863, 8816, 15863, 13674, 7769, 15367, 18054, 8816, 6151, 594, 16577, 18054, 14246, 15296, 12309, 8816, 20174, 12309, 8816, 12309, 9330, 6134, 6134, 594, 12309, 12309, 6857, 9330, 6930]

Decrypted message is : Realisation of combination of RSA and coloring graph methods is successful

Iteration 6 time 0.4285304546356201 seconds

Overall time ---1.1838440895080566 seconds ---

Average iteration time ---0.11838440895080567 seconds ---

Process finished with exit code 0

Додаток Е.

Результат №1 роботи програми з додатку 2.

C:\ProgramData\Anaconda3\python.exe "C:/Users/Владелец/PycharmProjects/RSA+Hamiltonian path/main.py"

What graph? (6, 8 or 10 vertices)

6

Iteration 1 time 0.20483994483947754 seconds

Iteration 2 time 0.11593079566955566 seconds

Iteration 3 time 0.1712498664855957 seconds

Iteration 4 time 0.05589485168457031 seconds

Iteration 5 time 0.14416885375976562 seconds

Length of privat exponent : 15

Length of encrypted message : 308

Encrypted message is : [11793, 24477, 13657, 4436, 20816, 8278, 13657, 5897, 20816, 25718, 12934, 19444, 25718, 21452, 19444, 13352, 25718, 21250, 7402, 20816, 12934, 13657, 5897, 20816, 25718, 12934, 19444, 25718, 21452, 19444, 11793, 3395, 16685, 19444, 13657, 12934, 28181, 19444, 1225, 13657, 21250, 20816, 4436, 5897, 25718, 12934, 20816, 13657, 12934, 19444, 10425, 13657, 5897, 1225, 8278, 19444, 21250, 24477, 5897, 1225, 25718, 28181, 8278, 19444, 20816, 8278, 19444, 8278, 12740, 13352, 13352, 24477, 8278, 8278, 21452, 12740, 4436]

Decrypted message is : Realisation of combination of RSA and hamiltonian paths methods is successful

Iteration 6 time 0.8728587627410889 seconds

Overall time ---1.5659425258636475 seconds ---

Average iteration time ---0.15659425258636475 seconds ---

Process finished with exit code 0

Додаток Ж.

Результат №2 роботи програми з додатку 2.

C:\ProgramData\Anaconda3\python.exe "C:/Users/Владелец/PycharmProjects/RSA+Hamiltonian path/main.py"

What graph? (6, 8 or 10 vertices)

8

Iteration 1 time 0.11034893989562988 seconds

Iteration 2 time 0.2627255916595459 seconds

Iteration 3 time 0.1378617286682129 seconds

Iteration 4 time 0.27170324325561523 seconds

Iteration 5 time 0.5913295745849609 seconds

Length of privat exponent : 15

Length of encrypted message : 308

Encrypted message is : [6513, 7638, 3495, 108, 6754, 11285, 3495, 2621, 6754, 3486, 3707, 7380, 3486, 867, 7380, 10222, 3486, 5123, 5618, 6754, 3707, 3495, 2621, 6754, 3486, 3707, 7380, 3486, 867, 7380, 6513, 4575, 7212, 7380, 3495, 3707, 4827, 7380, 1429, 3495, 5123, 6754, 108, 2621, 3486, 3707, 6754, 3495, 3707, 7380, 952, 3495, 2621,

1429, 11285, 7380, 5123, 7638, 2621, 1429, 3486, 4827, 11285, 7380, 6754, 11285, 7380, 11285, 2390, 10222, 10222, 7638, 11285, 11285, 867, 2390, 108]

Decrypted message is : Realisation of combination of RSA and hamiltonian paths methods is successful

Iteration 6 time 0.189100980758667 seconds

Overall time ---1.5630700588226318 seconds ---

Average iteration time ---0.15630700588226318 seconds ---

Process finished with exit code 0

Додаток 3.

Результат №3 роботи програми з додатку 2.

C:\ProgramData\Anaconda3\python.exe "C:/Users/Владелец/PycharmProjects/RSA+Hamiltonian path/main.py"

What graph? (6, 8 or 10 vertices)

10

Iteration 1 time 0.004001140594482422 seconds

Iteration 2 time 0.7387039661407471 seconds

Iteration 3 time 0.02500176429748535 seconds

Iteration 4 time 0.5810754299163818 seconds

Iteration 5 time 0.07103800773620605 seconds

Length of privat exponent : 15

Length of encrypted message : 308

Encrypted message is : [13941, 12445, 2648, 11664, 19673, 9688, 2648, 8659, 19673, 6324, 9157, 10054, 6324, 5926, 10054, 3580, 6324, 7612, 19910, 19673, 9157, 2648, 8659, 19673, 6324, 9157, 10054, 6324, 5926, 10054, 13941, 20372, 14998, 10054, 2648, 9157, 8005, 10054, 2827, 2648, 7612, 19673, 11664, 8659, 6324, 9157, 19673, 2648, 9157, 10054, 19219, 2648, 8659, 2827, 9688, 10054, 7612, 12445, 8659, 2827, 6324, 8005, 9688, 10054, 19673, 9688, 10054, 9688, 12332, 3580, 3580, 12445, 9688, 9688, 5926, 12332, 11664]

Decrypted message is : Realisation of combination of RSA and hamiltonian paths methods is successful

Iteration 6 time 0.9926769733428955 seconds

Overall time ---2.4124972820281982 seconds ---

Average iteration time ---0.2412497282028198 seconds ---

Process finished with exit code 0