ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

Кафедра прикладной математики и кибернетики

РАСЧЁТНО-ГРАФИЧЕСКАЯ РАБОТА

по дисциплине «Защита информации»

на тему

Доказательство с нулевым знанием для задачи «Гамильтонов цикл»

Выполнил студент Кулик Павел Евгеньевич

Ф.И.О.

Группы ИС-241

Работу принял _____ ассистент кафедры ПМиК Истомина А.С.

Подпись

Новосибирск – 2025

СОДЕРЖАНИЕ

Постановка задачи

Целью данной работы является написание программы, реализующей протокол доказательства с нулевым знанием для задачи «Гамильтонов цикл». Так как рассматривая задача является NP-полной и не имеет быстрого решения, необходимо также реализовать вспомогательную программу, генерирующую граф с заведомо известным решением и сохраняющую его в файл.

Программа, реализующая протокол должна считывать граф из файла, в котором он записан в следующем виде:

1) в первой строчке файла содержатся 2 числа $n < 1001$ и $m \leq n^2$, 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) в последней строке содержится последовательность номеров вершин, образующих гамильтонов цикл.

Реализованная программа должна наглядно демонстрировать работу алгоритма в текстовом формате. По итогам работы должен быть оформлен отчёт.

Теоретическая часть

1. Доказательство с нулевым знанием

В криптографических приложениях возникает задача доказательства знания некоторой информации без раскрытия её содержания. Пусть имеется два участника — Алиса и Боб. Алиса знает решение вычислительно сложной задачи и должна убедить Боба в наличии у неё этого решения, не сообщая при этом никаких сведений о самом решении. В результате Боб убеждается в факте знания, не получая при этом дополнительной информации, что соответствует понятию доказательства с нулевым знанием.

Практическая значимость подобных задач проявляется в системах аутентификации и компьютерных сетях. Сервер (Боб) принимает решение о предоставлении пользователю (Алисе) доступа к защищённым ресурсам, при этом секретная информация пользователя, например пароль, не раскрывается ни серверу, ни возможному нарушителю, осуществляющему перехват данных. Таким образом, сервер получает нулевое знание о секрете пользователя, но уверен в его наличии.

2. Задача о нахождении гамильтонова цикла в графе

Задача нахождения гамильтонова цикла в графе имеет важное теоретическое значение в криптографии, поскольку используется при построении протоколов доказательства с нулевым знанием. М. Блюм показал, что любое математическое утверждение может быть сведено к задаче существования гамильтонова цикла в некотором графе. Следовательно, наличие протокола доказательства с нулевым знанием для гамильтонова цикла позволяет строить такие протоколы для произвольных утверждений.

Гамильтоновым циклом в графе называется замкнутый путь, проходящий через каждую вершину графа ровно один раз.

Если в графе с n вершинами существует гамильтонов цикл, то при некоторой нумерации вершин он будет проходить через вершины с номерами $1,2,\ldots,n$. Теоретически задачу можно решить перебором всех возможных перестановок вершин, количество которых равно $n!$, однако при достаточно больших значениях n данный подход становится вычислительно неосуществимым. Доказано, что задача нахождения гамильтонова цикла относится к классу NP-полных, что неформально означает отсутствие известных эффективных алгоритмов её решения.

В рамках данной работы рассматривается криптографический протокол, с помощью которого Алиса доказывает Бобу знание гамильтонова цикла в заданном графе G, не раскрывая сам цикл. Такое доказательство является доказательством с нулевым знанием, поскольку после выполнения протокола Боб не получает никакой дополнительной информации о цикле, кроме факта его существования.

Будем обозначать графы символами G, H, F, отождествляя их с соответствующими матрицами смежности, где элемент $H_{ij} = 1$ обозначает наличие ребра между вершинами i и j, а $H_{ij} = 0$ – его отсутствие. Для работы протокола используется криптосистема с открытым ключом; в данной работе для определённости предполагается применение схемы RSA, параметры которой известны всем участникам, при этом расшифровка сообщений возможна только владельцем закрытого ключа – Алисой.

3. Описание протокола

Протокол доказательства состоит из следующих четырех шагов, повторяющихся t раз:

1) Алиса строит граф H, являющийся копией исходного графа G, где у всех вершин новые, случайно выбранные номера. На языке теории графов говорят, что H изоморфен G. Иными словами, H получается

путем некоторой перестановки вершин в графе G (с сохранением связей между вершинами). Алиса кодирует матрицу H , приписывая к первоначально содержащимся в ней нулям и единицам случайные числа r_{ij} по схеме $\tilde{H}_i = r_{ij} \cap H_{ij}$. Затем она шифрует элементы матрицы $\tilde{H}$, получая зашифрованную матрицу F , $F_i = \tilde{H}_i^d mod N$. Матрицу F Алиса передает Бобу.

2) Боб, получивший зашифрованный граф F, задаёт Алисе один из двух вопросов.
    a. Каков гамильтонов цикл для графа H?
    b. Действительно ли граф H изоморфен G?
3) Алиса отвечает на соответствующий вопрос Боба.
    a. Она расшифровывает в F рёбра, образующие гамильтонов цикл.
    b. Она расшифровывает F полностью (фактически передаёт Бобу граф H) и предъявляет перестановки, с помощью которых граф H был получен из графа G.
4) Получив ответ, Боб проверяет правильность расшифровки путем повторного шифрования и сравнения с F и убеждается либо в том, что показанные ребра действительно образуют гамильтонов цикл, либо в том, что предъявленные перестановки действительно переводят граф G в граф H.

Практическая часть

В соответствии с постановкой задачи были реализованы 2 программы. Первая программа отвечает за генерацию графа с гамильтоновым циклом. Программа может принимать на вход следующие параметры:

1. – *extra* – количество дополнительных вершин, помимо гамильтонового цикла;
2. – *name* – имя выходного файла;
3. – *size* – размер графа.

Результат работы программы при $size = 6$ и $extra = 0$. Представлен на рис. 1.

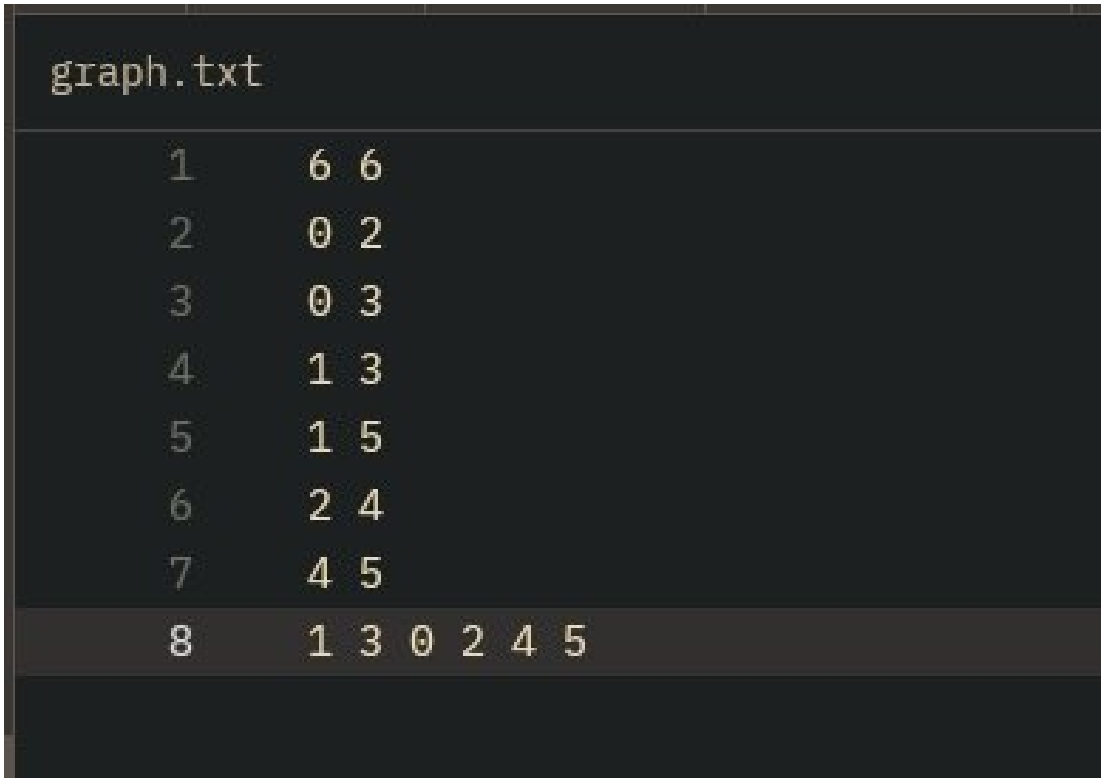

Рисунок 1. Граф, записанный в файл

Графическое изображение соответствующего графа представлено на рис. 2.

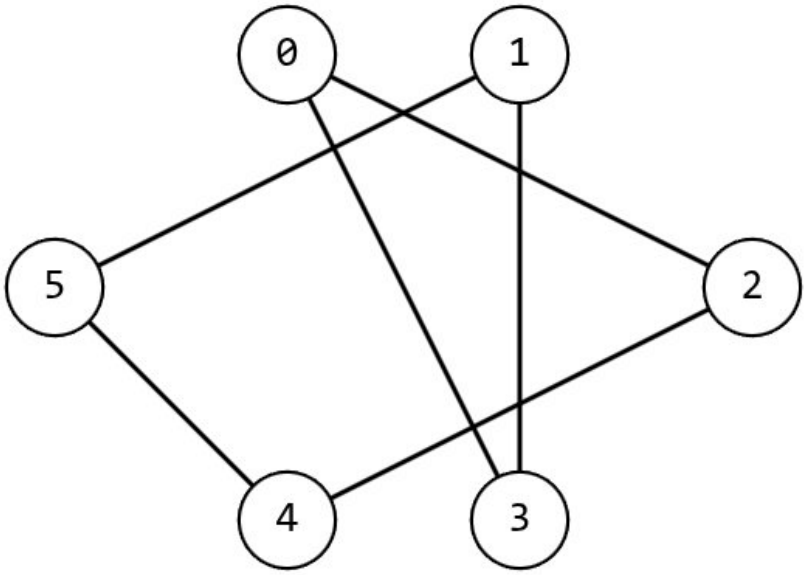

Рисунок 2. Изображение графа

Вторая программа реализует сам протокол и выводит полезные сообщения по ходу выполнения для демонстрации работы протокола. Программа может принимать на вход следующие параметры:

1. – *t* – количество раундов доказательства;
2. – *name* – имя входного файла.

Результат работы программы с представленным выше графом на входе и количеством раундов равным 5 представлен на рис. 3.

```
pahansan@pahansan-pc:~/code/infosec-rgr$ ./zkp -t 5
1: Боб задаёт вопрос: Каков гамильтонов цикл для графа H?
Алиса расшифровывает рёбра цикла: Боб убедился в наличии цикла. Боб шифрует цикл снова: граф снова равен F. Проверка успешна.
Вероятность обмана: 0.5

2: Боб задаёт вопрос: Каков гамильтонов цикл для графа H?
Алиса расшифровывает рёбра цикла: Боб убедился в наличии цикла. Боб шифрует цикл снова: граф снова равен F. Проверка успешна.
Вероятность обмана: 0.25

3: Боб задаёт вопрос: Действительно ли граф H изоморфен G?
Алиса демонстрирует перестановки: G и H совпадают. Боб повторно шифрует H: H == F. Проверка успешна.
Вероятность обмана: 0.0625

4: Боб задаёт вопрос: Каков гамильтонов цикл для графа H?
Алиса расшифровывает рёбра цикла: Боб убедился в наличии цикла. Боб шифрует цикл снова: граф снова равен F. Проверка успешна.
Вероятность обмана: 0.00390625

5: Боб задаёт вопрос: Действительно ли граф H изоморфен G?
Алиса демонстрирует перестановки: G и H совпадают. Боб повторно шифрует H: H == F. Проверка успешна.
Вероятность обмана: 1.52587890625e-05
```

Рисунок 3. Результат работы программы, реализующей протокол

Заключение

В ходе данной работы была реализована программа, реализующая протокол доказательства с нулевым знанием для задачи «Гамильтонов цикл». Программа соответствует всем требованиям, указанным при постановке задачи. Таким образом, цель была выполнена.

Приложение

```
==> cmd/demo/main.go <==
package main

import (
    "flag"
    "fmt"
    "infosec-rgr/internal/gamilton"
    "infosec-rgr/internal/rsa"
    "log"
    "math/big"
    "math/rand"
)

func main() {
    fileName := flag.String("name", "graph.txt", "<path/to/file>")
    t := flag.Int("t", 10, "<count of quetions>")
    flag.Parse()
    g, cycle, err := gamilton.NewGraphFromFile(*fileName)
    if err != nil {
        log.Fatal(err)
    }
    keys, err := rsa.GenerateKeys()
    if err != nil {
        log.Fatal(err)
    }

    lie := big.NewFloat(0.5)
    for i := range *t {
        fmt.Print(i+1, ": ")
        status := gamilton.Protocol(g, cycle, rand.Intn(2), keys)
        if status == true {
            fmt.Println("Проверка успешна.")
        } else {
            fmt.Println("Проверка не прошла. Обман раскрыт.")
            return
        }
        fmt.Print("Вероятность обмана: ", lie, "\n\n")
        lie.Mul(lie, lie)
    }
}

==> cmd/generate-graph/main.go <==
package main

import (
    "flag"
    "fmt"
    "infosec-rgr/internal/gamilton"
    "log"
    "os"
)

func main() {
    fileName := flag.String("name", "graph.txt", "<path/to/file>")
    size := flag.Int("size", 10, "<number of vertexes>")
    nExtraEdges := flag.Int("extra", 0, "<number of edges except cycle>")
    flag.Parse()
    nEdges := *size + *nExtraEdges
    maxEdges := *size * (*size - 1) / 2

    if nEdges > maxEdges {
        log.Fatalf("Sum of vertexes in cycle and extra vertexes can't be bigger
than square size")
```

```
    }

    g, cycle := gamilton.NewGraphWithCycle(*size)
    g.AddNRandEdges(*nExtraEdges)

    file, err := os.Create(*fileName)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    fmt.Fprintf(file, "%d %d\n", *size, nEdges)

    for i := range *size {
        for j := i + 1; j < *size; j++ {
            if g.EdgeExists(i, j) {
                fmt.Fprintf(file, "%d %d\n", i, j)
            }
        }
    }

    for i := range cycle {
        fmt.Fprintf(file, "%d ", cycle[i])
    }
}

==> internal/filereader/filereader.go <==
package filereader

import (
    "bufio"
    "fmt"
    "strconv"
    "strings"
)

func Read2Numbers(scanner *bufio.Scanner) (int, int, error) {
    if !scanner.Scan() {
        return 0, 0, fmt.Errorf("bad file format")
    }
    line := scanner.Text()
    var n1, n2 int
    n, err := fmt.Sscanf(line, "%d %d", &n1, &n2)
    if err != nil {
        return 0, 0, err
    }
    if n != 2 {
        return 0, 0, fmt.Errorf("bad file format")
    }
    return n1, n2, nil
}

func ReadSlice(scanner *bufio.Scanner, size int) ([]int, error) {
    if !scanner.Scan() {
        return nil, fmt.Errorf("bad file format")
    }

    line := scanner.Text()
    fields := strings.Fields(line)

    if len(fields) != size {
        return nil, fmt.Errorf("bad file format")
    }

    array := make([]int, size)
```

```
    for i, v := range fields {
        num, err := strconv.Atoi(v)
        if err != nil {
            return nil, fmt.Errorf("bad file format")
        }
        array[i] = num
    }

    return array, nil
}

==> internal/gamilton/gamilton.go <==
package gamilton

import (
    "fmt"
    "infosec-rgr/internal/rsa"
    "math/rand"
)

func GamiltonCycle(size int) []int {
    cycle := make([]int, size)
    for i := range size {
        cycle[i] = i
    }
    rand.Shuffle(size, func(i, j int) {
        cycle[i], cycle[j] = cycle[j], cycle[i]
    })
    return cycle
}

func NewGraphWithCycle(size int) (*Graph, []int) {
    cycle := GamiltonCycle(size)
    graph := NewGraph(size)
    for i := 1; i < size; i++ {
        graph.AddEdge(cycle[i-1], cycle[i])
    }
    graph.AddEdge(cycle[0], cycle[size-1])
    return graph, cycle
}

func transformCycle(cycle, permutations []int) []int {
    newCycle := make([]int, len(cycle))
    for i := range len(cycle) {
        newCycle[i] = permutations[cycle[i]]
    }
    return newCycle
}

func Protocol(g *Graph, cycle []int, q int, keys *rsa.Keys) bool {
    h, perms := g.IsomorphicCopy()
    hTilda := h.AddPadding()
    f := hTilda.EncryptRSA(keys)
    switch q {
    case 0:
        fmt.Println("Боб задаёт вопрос: Каков гамильтонов цикл для графа H?")
        fmt.Print("Алиса расшифровывает рёбра цикла: ")
        newCycle := transformCycle(cycle, perms)
        fDecrypted := f.DecryptCycleRSA(keys, newCycle)
        if !fDecrypted.CheckCycle(newCycle) {
            return false
        }
        fmt.Print("Боб убедился в наличии цикла. ")
        fmt.Print("Боб шифрует цикл снова: ")
        fEncrypted := fDecrypted.EncryptCycleRSA(keys, newCycle)
```

```
            if f.Equals(fEncrypted) {
                fmt.Print("граф снова равен F. ")
                return true
            } else {
                return false
            }
        case 1:
            fmt.Println("Боб задаёт вопрос: Действительно ли граф H изоморфен G?")
            fmt.Print("Алиса демонстрирует перестановки: ")
            fDecrypted := f.DecryptRSA(keys)
            original := fDecrypted.IsomorphicOriginal(perms)
            if !original.SameAs(g) {
                return false
            }
            fmt.Print("G и H совпадают. ")
            fmt.Print("Боб повторно шифрует H: ")
            fEncrypted := fDecrypted.EncryptRSA(keys)
            if !fEncrypted.Equals(f) {
                return false
            }
            fmt.Print("H == F. ")
            return true
        default:
            return false
        }
    }

    ==> internal/gamilton/graph.go <==
    package gamilton

    import (
        "bufio"
        "infosec-rgr/internal/filereader"
        "infosec-rgr/internal/rsa"
        "math/big"
        "math/rand"
        "os"
    )

    type Graph struct {
        v [][]*big.Int
    }

    func NewGraph(size int) *Graph {
        g := &Graph{v: make([][]*big.Int, size)}
        for i := range size {
            g.v[i] = make([]*big.Int, size)
            for j := range size {
                g.v[i][j] = new(big.Int)
            }
        }
        return g
    }

    func NewGraphFromFile(filename string) (*Graph, []int, error) {
        file, err := os.Open(filename)
        if err != nil {
            return nil, nil, err
        }
        defer file.Close()

        scanner := bufio.NewScanner(file)
        size, nEdges, err := filereader.Read2Numbers(scanner)
        if err != nil {
            return nil, nil, err
```

```
    }

    g := NewGraph(size)
    for range nEdges {
        r, c, err := filereader.Read2Numbers(scanner)
        if err != nil {
            return nil, nil, err
        }
        g.AddEdge(r, c)
    }

    cycle, err := filereader.ReadSlice(scanner, size)
    if err != nil {
        return nil, nil, err
    }

    return g, cycle, nil
}

func (g *Graph) Copy() *Graph {
    size := g.Size()
    newGraph := NewGraph(size)
    for i := range size {
        for j := range size {
            newGraph.v[i][j].Set(g.v[i][j])
        }
    }
    return newGraph
}

func (g *Graph) IsomorphicCopy() (*Graph, []int) {
    size := g.Size()

    permutation := make([]int, size)
    for i := range size {
        permutation[i] = i
    }

    rand.Shuffle(size, func(i, j int) {
        permutation[i], permutation[j] = permutation[j], permutation[i]
    })

    newGraph := NewGraph(size)

    for i := range size {
        for j := range size {
            newI := permutation[i]
            newJ := permutation[j]
            newGraph.v[newI][newJ].Set(g.v[i][j])
        }
    }
    return newGraph, permutation
}

func (g *Graph) IsomorphicOriginal(permutation []int) *Graph {
    size := g.Size()

    inverse := make([]int, size)
    for i, p := range permutation {
        inverse[p] = i
    }

    original := NewGraph(size)

    for i := range size {
```

```
        for j := range size {
            origI := inverse[i]
            origJ := inverse[j]
            original.v[origI][origJ].Set(g.v[i][j])
        }
    }

    return original
}

func (g *Graph) Size() int {
    return len(g.v)
}

func (g *Graph) AddEdge(i, j int) {
    g.v[i][j].SetInt64(1)
    g.v[j][i].SetInt64(1)
}

func (g *Graph) RemoveEdge(i, j int) {
    g.v[i][j].SetInt64(0)
    g.v[j][i].SetInt64(0)
}

func (g *Graph) EdgeExists(i, j int) bool {
    size := g.Size()
    if i >= size || j >= size {
        return false
    }
    return g.v[i][j].Bit(0) == 1
}

func (g *Graph) SetEdge(i, j int, value *big.Int) {
    g.v[i][j].Set(value)
    g.v[j][i].Set(value)
}

func (g *Graph) GetEdge(i, j int) *big.Int {
    return new(big.Int).Set(g.v[i][j])
}

func (g *Graph) AddNRandEdges(n int) {
    i := 0
    size := g.Size()
    for i < n {
        r := rand.Intn(size)
        c := rand.Intn(size)
        if r == c {
            continue
        }
        if g.EdgeExists(r, c) {
            continue
        }
        g.AddEdge(r, c)
        i++
    }
}

func (g *Graph) padWithRandomness(i, j int) *big.Int {
    randomPart := big.NewInt(rand.Int63())
    randomPart.Lsh(randomPart, 1)

    if g.EdgeExists(i, j) {
        randomPart.Or(randomPart, big.NewInt(1))
    }
```

```
    return randomPart
}

func (g *Graph) AddPadding() *Graph {
    size := g.Size()
    newG := NewGraph(size)
    for i := range size {
        for j := i + 1; j < size; j++ {
            newValue := g.padWithRandomness(i, j)
            newG.SetEdge(i, j, newValue)
        }
    }
    return newG
}

func (g *Graph) RemovePadding() *Graph {
    size := g.Size()
    newG := NewGraph(size)
    for i := range size {
        for j := i + 1; j < size; j++ {
            if g.EdgeExists(i, j) {
                newG.AddEdge(i, j)
            } else {
                newG.RemoveEdge(i, j)
            }
        }
    }
    return newG
}

func (g *Graph) encryptEdgeRSA(i, j int, keys *rsa.Keys) {
    plain := g.GetEdge(i, j)
    cipher, _ := rsa.Encrypt(plain, keys.D, keys.N)
    g.SetEdge(i, j, cipher)
}

func (g *Graph) decryptEdgeRSA(i, j int, keys *rsa.Keys) {
    cipher := g.GetEdge(i, j)
    plain, _ := rsa.Decrypt(cipher, keys.C, keys.N)
    g.SetEdge(i, j, plain)
}

func (g *Graph) EncryptRSA(keys *rsa.Keys) *Graph {
    size := g.Size()
    encrypted := g.Copy()

    for i := range size {
        for j := i + 1; j < size; j++ {
            encrypted.encryptEdgeRSA(i, j, keys)
        }
    }
    return encrypted
}

func (g *Graph) DecryptRSA(keys *rsa.Keys) *Graph {
    size := g.Size()
    decrypted := g.Copy()

    for i := range size {
        for j := i + 1; j < size; j++ {
            decrypted.decryptEdgeRSA(i, j, keys)
        }
    }
    return decrypted
```

```
}

func (g *Graph) EncryptCycleRSA(keys *rsa.Keys, cycle []int) *Graph {
    size := g.Size()
    encrypted := g.Copy()

    for i := 1; i < size; i++ {
        encrypted.encryptEdgeRSA(cycle[i-1], cycle[i], keys)
    }
    encrypted.encryptEdgeRSA(cycle[0], cycle[size-1], keys)
    return encrypted
}

func (g *Graph) DecryptCycleRSA(keys *rsa.Keys, cycle []int) *Graph {
    size := g.Size()
    decrypted := g.Copy()

    for i := 1; i < size; i++ {
        decrypted.decryptEdgeRSA(cycle[i-1], cycle[i], keys)
    }
    decrypted.decryptEdgeRSA(cycle[0], cycle[size-1], keys)
    return decrypted
}

func (g *Graph) Equals(other *Graph) bool {
    size := g.Size()
    if size != other.Size() {
        return false
    }
    for i := range size {
        for j := i + 1; j < size; j++ {
            if g.v[i][j].Cmp(other.v[i][j]) != 0 {
                return false
            }
        }
    }
    return true
}

func (g *Graph) CheckCycle(cycle []int) bool {
    for i := 1; i < len(cycle); i++ {
        if !g.EdgeExists(cycle[i-1], cycle[i]) {
            return false
        }
    }
    if !g.EdgeExists(cycle[0], cycle[len(cycle)-1]) {
        return false
    }
    return true
}

func (g *Graph) SameAs(other *Graph) bool {
    size := g.Size()
    if size != other.Size() {
        return false
    }
    for i := range size {
        for j := i + 1; j < size; j++ {
            if g.EdgeExists(i, j) != other.EdgeExists(i, j) {
                return false
            }
        }
    }
    return true
}
```

```
==> internal/rsa/rsa.go <==
package rsa

import (
    "crypto/rand"
    "fmt"
    "math/big"
)

type Keys struct {
    C *big.Int // private
    D *big.Int // public
    N *big.Int // public
}

func GenerateKeys() (*Keys, error) {
    P, err := rand.Prime(rand.Reader, 256)
    if err != nil {
        return nil, err
    }
    Q, err := rand.Prime(rand.Reader, 256)
    if err != nil {
        return nil, err
    }
    d, err := rand.Prime(rand.Reader, 256)
    if err != nil {
        return nil, err
    }
    N := new(big.Int).Mul(P, Q)
    phi := new(big.Int).Mul(P.Sub(P, big.NewInt(1)), Q.Sub(Q, big.NewInt(1)))
    gcd := new(big.Int)
    for {
        gcd.GCD(nil, nil, d, phi)
        if gcd.Cmp(big.NewInt(1)) == 0 {
            break
        }
        d, err = rand.Prime(rand.Reader, 256)
        if err != nil {
            return nil, err
        }
    }
    c := new(big.Int).ModInverse(d, phi)

    return &Keys{c, d, N}, nil
}

func Encrypt(m, d, N *big.Int) (*big.Int, error) {
    if m.Cmp(N) != -1 {
        return nil, fmt.Errorf("message size must be less than key N")
    }

    e := new(big.Int).Exp(m, d, N)
    return e, nil
}

func Decrypt(e, c, N *big.Int) (*big.Int, error) {
    if e.Cmp(N) != -1 {
        return nil, fmt.Errorf("message size must be less than key N")
    }

    m := new(big.Int).Exp(e, c, N)
    return m, nil
}
```