САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Отчет по лабораторной работе №3

по дисциплине «Системное программное обеспечение»

Выполнили: студенты гр. R3335

Ал-Наим Рами

Костровская Ольга

Носкова Полина

Преподаватель:

Капитонов А.А.

Санкт-Петербург

2018 г.

Цель работы: закрепление навыков программирования на Python, повторение ранее изученного материала.

Задание 1. «Лабиринт»

Используя лекционные материалы, создайте игру «лабиринт». Желательный размер лабиринта – 50 х 50 клеток (можно выбрать свой размер, но не менее 20х20), пользователь может перемещаться вверх, вниз, вправо и влево. Каждый шаг смещает пользователя на одну клетку, если не мешает стена. Вход и выход из лабиринта генерируются случайно при каждом запуске игры. При прохождении лабиринта пользователю выдается информация о затраченном количестве шагов, времени прохождения, предложение завершить игру и начать заново.

Решение:

Мы воспользовались алгоритмом, основанным на бэктрекинге (поиске с возвратом), который позволяет создавать лабиринты без циклов, имеющие единственный путь между двумя точками. Алгоритм не самый быстрый, но очень прост в реализации и позволяет создавать ветвистые лабиринты с очень длинными тупиковыми ответвлениями. Предполагается, что изначально у каждой клетки есть стенки со всех четырех сторон, которые отделяют ее от соседних клеток. Опишем алгоритм поэтапно.

1. Создаем начальную клетку текущей и отмечаем ее как посещенную;

2. Пока есть не посещённые клетки:

    1. Если текущая клетка имеет не посещённых «соседей»:

        1. Проталкиваем текущую клетку в стек;

        2. Выбираем случайную клетку из соседних;

        3. Убираем стенку между текущей клеткой и выбранной;

        4. Делаем выбранную клетку текущей и отмечаем ее как посещенную.

    2. Иначе, если стек не пуст:

        1. Вытаскиваем клетку из стека;

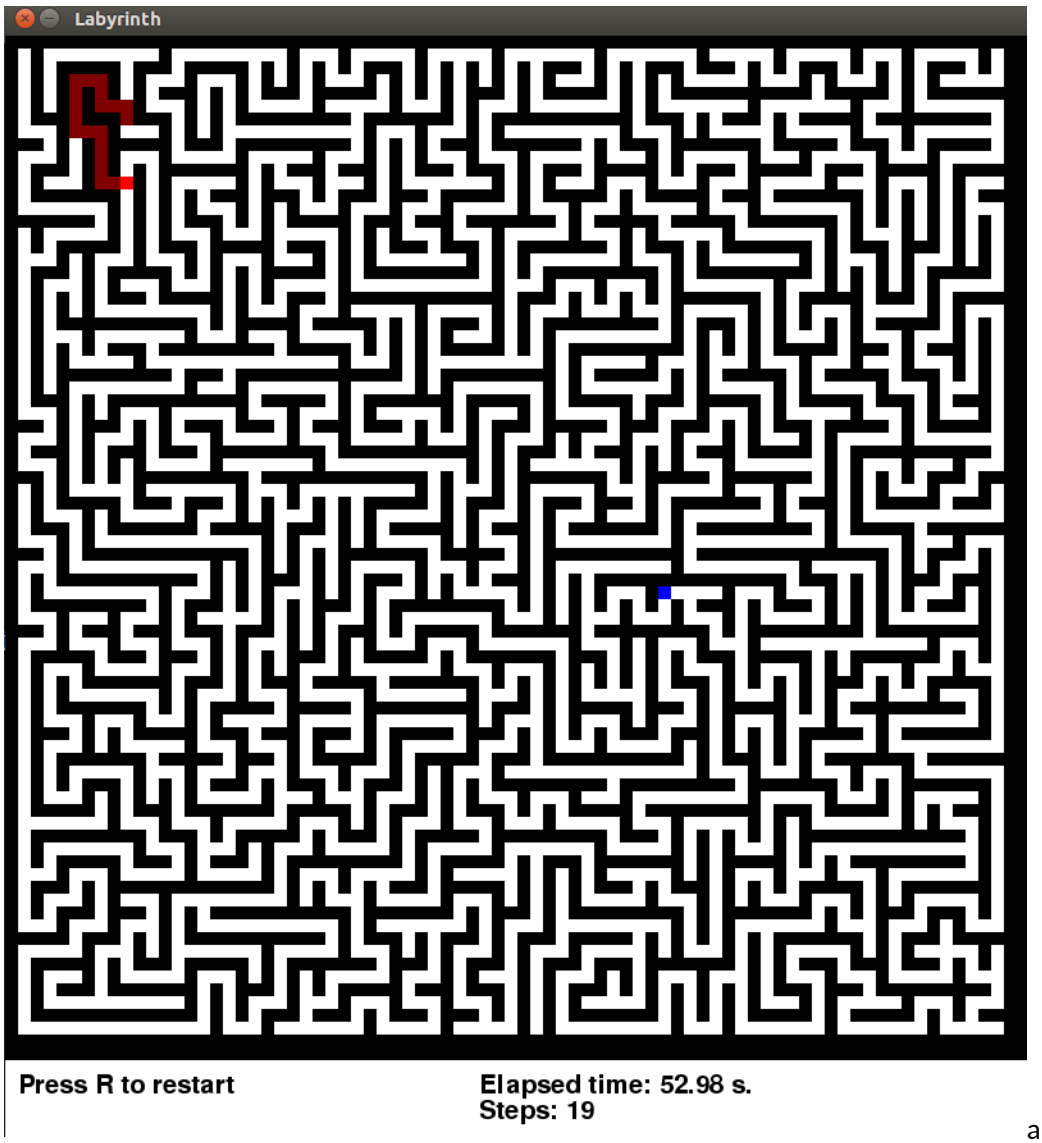
        2. Делаем ее текущей.

    3. Иначе:

        1. Выбираем случайную не посещённую клетку, делаем ее текущей и отмечаем, как посещенную.

Выходной точкой, как и стартовой, выступает любая точка лабиринта, не являющаяся стенкой. Выход должен быть «прижат» к одной из стенок, но, по сути, может находиться где угодно.

Критерий нахождения «выхода» очень прост: достаточно сравнить координаты текущей точки и координаты «выхода»: если они равны, путь между стартовой и выходной точками найден. Точка «вход» - красного цвета, точка «выход» - синего цвета.



a

Рисунок 1. Лабиринт

Задание 2. «Поиск выхода»

Напишите программу, которая будет автоматически передвигать пользователя из задания 1 и искать выход из лабиринта. При прохождении лабиринта выдается информация о количестве шагов.

Решение:

Теперь рассмотрим алгоритм поиска выхода из лабиринта.

Алгоритм поиска пути бэктрекингом:

1. Сделаем начальную клетку текущей и отметим ее как посещенную;
2. Пока не найден выход:
    1. Если текущая клетка имеет не посещенных «соседей»:
        1. Проталкиваем текущую клетку в стек;
        2. Выбираем случайную клетку из соседних;

3. Сделаем выбранную клетку текущей и отметим ее как посещенную.

2. Иначе если стек не пуст:

    1. Вытаскиваем клетку из стека;

    2. Сделаем ее текущей.

3. Иначе выхода нет.

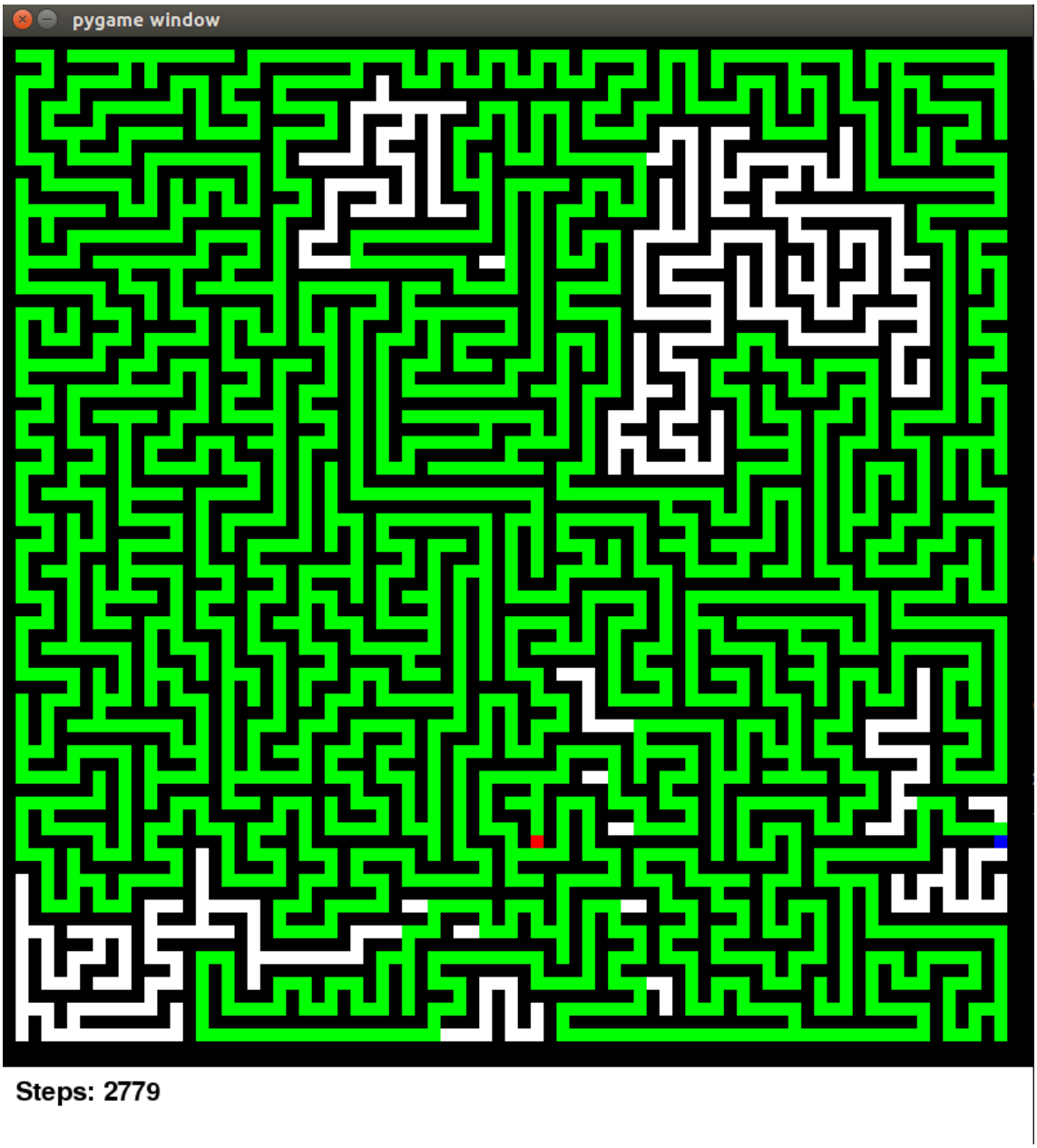


Рисунок 2. Поиск выхода

Вывод:

В этих заданиях мы изучили библиотеку для написания игровых приложений на языке Python - pygame. Научились с ней работать, успешно написали лабиринт. Так же в процессе выполнения лабораторной работы мы освоили новый алгоритм – метод поиска в глубину по графу.

Задание 3. «ТАУ»

Напишите свои алгоритмы для вычисления управляемости, наблюдаемости и ранга матрицы. Сравните быстродействие полученных функций с аналогами NumPy.

Решение:

Для вычисления управляемости, наблюдаемости и ранга матрицы нам понадобились вспомогательные функции: умножение матриц, возведение матрицы в степень и «проверка», которая удаляет из матрицы столбцы и строки содержащие нули.

def check(matrix):

Удаляет из матрицы столбцы и строки, содержащие одни нули.

```
def  check(matrix):
    list=[]
    for i in range (0,rows(matrix)):
        flag = 0
        for j in range(0, columns(matrix)):
            if matrix[i][j] != 0 : flag =1
        if flag ==0:
            list.append(i)
    step = 0
    for i in list:
        del matrix[i-step]
        step += 1
    list = []
    for j in range(0,len(matrix[0])):
        flag = 0
        for i in range (0, rows(matrix)):
            if matrix [i] [j] != 0 : flag = 1
        if flag == 0:
            list.append(j)
    step = 0
    for j in list:
        for i in range(0,rows(matrix)):
            del matrix[i][j-step]
        step += 1
    return matrix
```

Возведение матрицы в степень N (N – целое число).

Для возведения матрицы в степень N, необходимо умножить матрицу саму на себя N раз, пользуясь правилом умножения матриц. При этом необходимо помнить, что в степень можно возвести только квадратную матрицу и показатель степени должен быть натуральным числом. Поэтому после проверки на «квадратность» if rows(matrix) == columns(matrix): , мы или возводим ее в степень, или выдаем сообщение, что возведение невозможно.

Точно так же, как и в предыдущем задании, мы формируем матрицу, которая представляет собой вложенный список из списков-строк, содержащий элементы, полученные по правилу перемножения матриц. resultPower += matrix[i][k] * matrix[k][j]

Для получения resultPower , мы используем цикл в цикле, ведь нам нужно пройтись по всем элементам наших исходных матриц.

```
def power(matrix,power):
    i=0
    j=0
    k=0
    p=1
    resultPower=0
    if rows(matrix) == columns(matrix):
        matrixInPower = []
        while p < power:
            p+=1
            for i in range (rows(matrix)):
                line = []
                for j in range(columns(matrix)):
                    for k in range (rows(matrix[i])):
                        resultPower += matrix [i][k]*matrix [k][j]
                    line.append(resultPower)
                    resultPower=0
                matrixInPower.append(line)
        else:
            return matrixInPower
    else:
        return 'Matrix shoud be square'
```

Умножение матриц.

Чтобы матрицу K можно было умножить на матрицу L нужно, чтобы число столбцов матрицы K равнялось числу строк матрицы L. Поэтому мы обязательно осуществляем проверку. if rows(matrix2) == columns(matrix1): Умножение элементов происходит по следующей формуле:

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} a_1c_1 + b_1c_2 \\ a_2c_1 + b_2c_2 \end{pmatrix}$$

```
def mult(matrix1,matrix2):
    i=0
    j=0
    k=0
    resultLine = 0
    if rows(matrix2) == columns(matrix1):
        result = []
        for i in range(rows(matrix1)):
            line=[]
            for j in range (len(matrix2[i])):
                for k in range(columns(matrix1)):
                    resultLine += matrix1[i][k] * matrix2[k][j]
                line.append(resultLine)
                resultLine = 0
            result.append(line)
        else:
            return result
    else:
        return "Multiplication is impossible."
```

Вычисление матрицы наблюдаемости.

A, C - матрицы постоянных коэффициентов с размерами соответственно n x n, n x r.

Для осуществления управления необходимо иметь информацию о текущем состоянии системы, то есть о значениях вектора состояния x(t) в каждый момент времени. Однако некоторые из переменных состояния являются абстрактными, не имеют физических аналогов в реальной системе или же не могут быть измерены. Измеряемыми и наблюдаемыми являются физические выходные переменные y(t).

Пусть матрицы A и C постоянны тогда *матрицу наблюдаемости* можно найти как:

$$\begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

```
def observabilityMatrix(matrix1,matrix2):
    n = len(matrix2)*len(matrix2[0])
    result = [0]*n
    for i in range (n):
        result[i] = [0]* len(matrix2[0])
    result[0]= matrix2
    i=0
    while i < (len(matrix2[0])-1):
        i += 1
        result[i]=mult(matrix2,matrix1)
        matrix1 = power(matrix1,i+1)
    result1 = [0]*n
    for i in range(n):
        result1[i] = []
    i = 0
    for j in range(len(matrix2[0])):
        for k in range(len(result[0])):
            result1[i] = result[j][k]
            i +=1
    return result1
```

Проверка системы на наблюдаемость.

Справедлив следующий критерий наблюдаемости: линейный стационарный объект X вполне наблюдаем тогда и только тогда, когда ранг матрицы наблюдаемости равен размерности n пространства состояний объекта.

```
def checkObservability(matrix1,matrix2):
    rangMatrix = rang(observabilityMatrix(matrix1,matrix2),0)
    n = columns(matrix2)
    if n == rangMatrix:
        return "The matrix is observable."
    else:
        return "The matrix is not observable."
```

Вычисление матрицы управляемости.

A, B - матрицы постоянных коэффициентов с размерами соответственно n x n, n x r .

Объект называют полностью управляемым, если его можно с помощью некоторого ограниченного управляющего воздействия U(t) перевести в течение конечного интервала времени tк из любого начального состояния X(0) в заданное конечное состояние X(tк).

Для осуществления такого перевода объекта необходимо, но не достаточно, чтобы каждая из переменных состояния Xi (i=1,...,n) зависела хотя бы от одной из составляющих Uj (j=1,...,r) вектора управлений U(t).

Без доказательства приведём критерий управляемости линейных стационарных объектов.

Пусть матрицы A и B постоянны. Введём так называемую *матрицу управляемости* $Q_y = [B \quad AB \quad A^2B \ldots A^{n-1}B]$, которая состоит из столбцов матрицы B и произведений матриц AB, A^2B,..., $A^{n-1}B$ и имеет размерность (n * nr).

```
def controllabilityMatrix(matrix1,matrix2):
    n = len(matrix2)*len(matrix2[0])
    result = [0]*n
    for i in range (n):
        result[i] = [0] * len(matrix2)
    result[0] = matrix2
    i=0
    while i < (len(matrix2)-1):
        i += 1
        result[i] = mult(matrix1,matrix2)
        matrix1 = power(matrix1,i+1)
    result1= [0] * len(matrix2)
    for i in range(len(matrix2)):
        result1[i] = []
    for i in range(len(matrix2)):
        for j in range (len(matrix2)):
            result1[i] = result1[i] + result[j][i]
    return result1
```

Проверка системы на управляемость.

Справедлив следующий критерий управляемости: линейный стационарный объект X вполне управляем тогда и только тогда, когда ранг матрицы управляемости равен размерности n пространства состояний объекта.

```
def checkControllability(matrix1,matrix2):
    rangMatrix = rang(controllabilityMatrix(matrix1,matrix2),0)
    n = rows(matrix2)
    if n == rangMatrix:
        return "The matrix is controllable."
    else:
        return "The matrix is not controllable."
```

Определение ранга матрицы.

Определение ранга матрицы производится по методу Гаусса. Согласно ему, матрица должна быть приведена к трапециевидной форме, причем на ее главной диагонали не должно находиться нулевых элементов. Так же в процессе приведения к такому виду из матрицы удаляются нулевые строки и столбцы. Получившееся количество строк итоговой матрицы равно ее рангу.

```
def rang(matrix,n):
    ch_matr = check(matrix)
    f_matr, k = swap(ch_matr, n)
    minim = rows(f_matr) if (rows(f_matr)) <= columns (f_matr) else columns(f_matr)
    global r
    r = rows(matrix)
    N = n
    if (n < minim-1) or k != 0:
        for i in range(n+1, rows(f_matr)):
            kek = -f_matr[i][n]/f_matr[n][n]
            for j in range(n, columns(f_matr)):
                f_matr[i][j] = kek* f_matr[n][j]+f_matr[i][j]
        return rang(f_matr, n+1)
    else:
        return r
```

```
rami@rami-K501UX:~/Desktop/labyrinth$ python Lab_3_rang.py
ControllabilityMatrix:
[1, 20, 346]
[2, 28, 440]
[3, 50, 814]

Observability natrix:
[1, 7, 3]
[28, 61, 72]
[700, 937, 1098]

The matrix is controllable.
The matrix is observable.
The rank of the matrix A is:
3
```

Рисунок 3. Вывод без NumPy.

Аналогичным образом был реализован алгоритм с помощью NumPy.

```
rami@rami-K501UX:~/Desktop/labyrinth$ python3 Lab_3_rang2.py
The rank of the matrix A is:
3
The matrix is observable.
The matrix is controllable.
```

Рисунок 4. Вывод с NumPy.

Вывод:

После выполнения лабораторной работы мы научились работа с модулями на языке Python, создавать собственные функции и применять его для решения математических задач. Время работы программы без NumPy = 0.0013, а время работы программы с NumPy = 0.0007. Как мы видим, время работы уменьшилось, значит наши функции написаны не так эффективно, как функции в библиотеки NumPy.