Министерство науки и высшего образования РФ
Пензенский государственный университет
Кафедра “Вычислительная техника”

Отчёт

по лабораторной работе №8
по курсу “Логика и основы алгоритмизации в инженерных задачах”
на тему “Обход графа в ширину”

Выполнил студент гр. 22ВВВ3:
Кулахметов С.И.

Приняли:
к.т.н., доцент Юрова О.В.
к.э.н., доцент Акифьев И.В.

Пенза 2023

Цель работы

Реализовать алгоритм обхода в ширину графа при помощи матрицы смежности и списков смежности. Выполнить лабораторное задание.

Лабораторное задание

Задание 1

1. Сгенерировать (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Вывести матрицу на экран.

2. Для сгенерированного графа осуществить процедуру обхода в ширину, реализованную в соответствии с приведенным в методических указаниях алгоритмом. При реализации алгоритма в качестве очереди использовать класс *queue* из стандартной библиотеки C++.

3. Реализовать процедуру обхода в ширину для графа, представленного списками смежности.

Задание 2

1. Для матричной формы представления графов реализовать алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», самостоятельно созданной в лабораторной работе № 3.

2. Оценить время работы двух реализаций алгоритмов обхода в ширину (использующего стандартный класс *queue* и использующего очередь, реализованную самостоятельно) для графов разных порядков.

Пояснительный текст к прорамме

Программа разделена на 2 файла, соответствующих заданиям лабораторной работы.

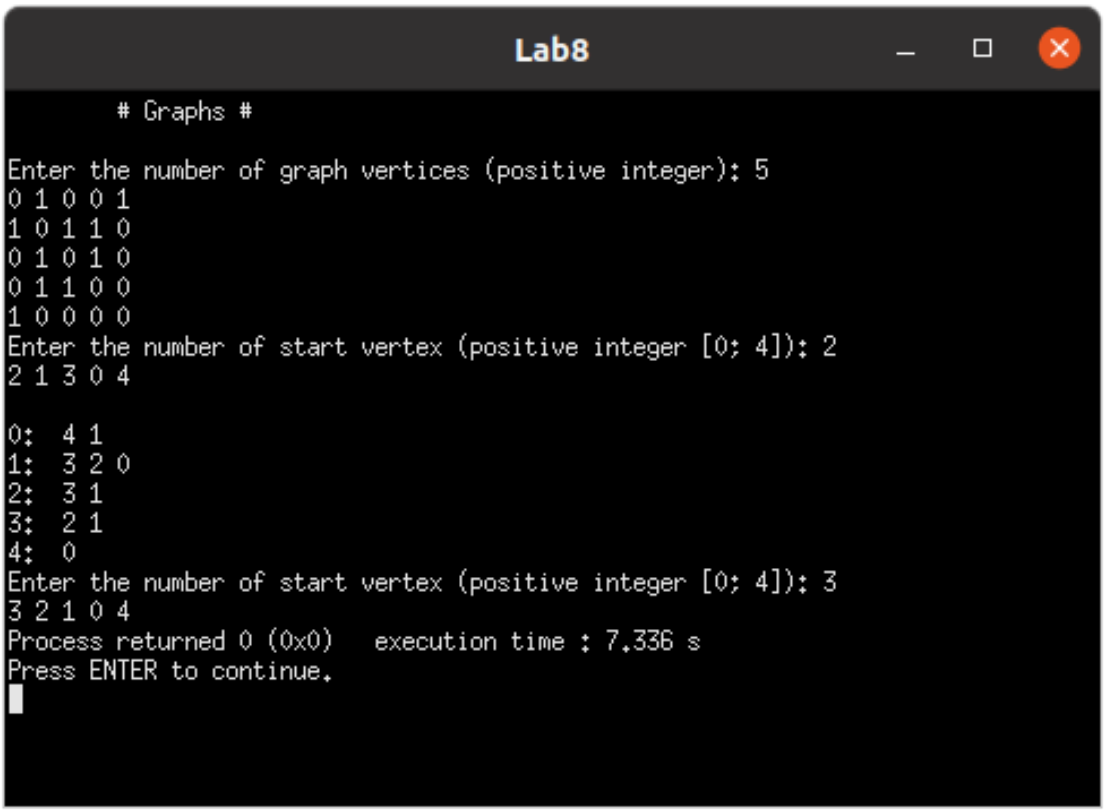
В первой программе инициализирован двумерный массив *graph[m][n]*, отвечающий за хранение матрицы смежности неориентированного графа. А также структуры *node* и *Graph*, отвечающие за реализацию списков смежности заданного графа. Функции обхода в глубину B*FS* и *bfs*, согласно реализации алгоритма, используют стандартный класс *queue*.

Во второй программе функции обхода в ширину *BFS* и *bfs* используют очередь *queue*, реализованную через стандартный класс C++, и очередь *Queue*, написанную в лабораторной работе №3 — соответственно.

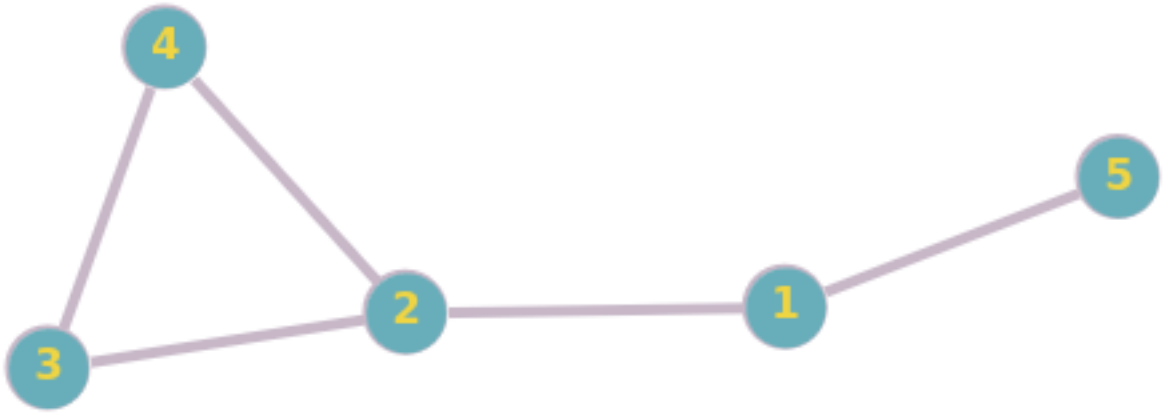
Результаты работы программы

Задание 1

Построение матрицы и списков смежности. Выполнение обхода в ширину.

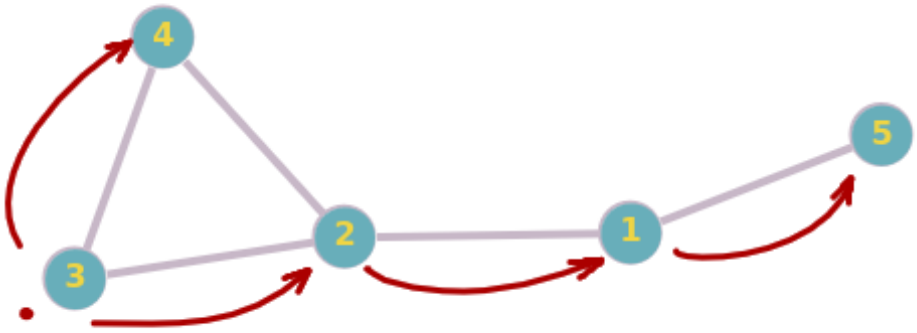


Граф, сгенерированный по матрице смежности.



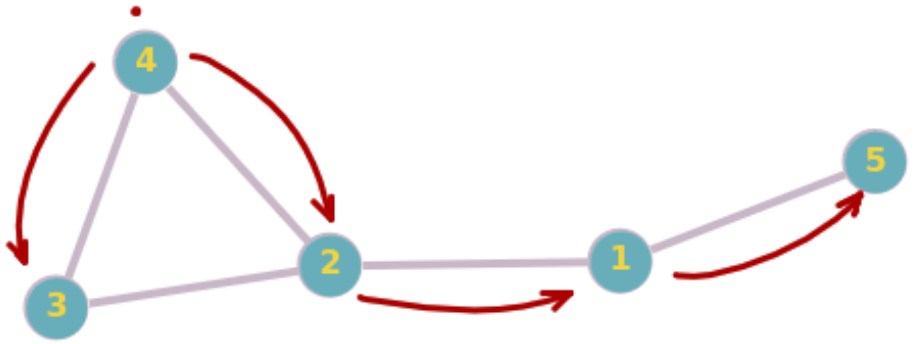
Обход графа в ширину по матрице инцидентности.

```
Enter the number of graph vertices (positive integer): 5
0 1 0 0 1
1 0 1 1 0
0 1 0 1 0
0 1 1 0 0
1 0 0 0 0
Enter the number of start vertex (positive integer [0; 4]): 2
2 1 3 0 4
```



Обход графа в ширину по спискам инцидентности.

```
0:  4 1
1:  3 2 0
2:  3 1
3:  2 1
4:  0
Enter the number of start vertex (positive integer [0; 4]): 3
3 2 1 0 4
```



Задание 2

Измерение времени при обходе графов в ширину.

Количество вершин	Время обхода	
	queue (C++ class)	Queue (from Lab3)
5	0.063 с	0.019 с
10	0.073 с	0.023 с
15	0.098 с	0.046 с

(Замеры см. Приложение Б)

Вывод

В ходе выполнения данной лабораторной работы были получены навыки реализации на языке C++ алгоритма поиска в ширину в неориентированном графе, представленном матрицей и списками смежности. Проведено сравнение скорости работы алгоритма при использовании очереди из стандартного класса C++ и написанной в лабораторной работе №3: в результате можно сделать вывод о том, что самописная очередь работает в 2-3 раза быстрее стандартной.

Ссылка на *GitHub* репозиторий с лабораторной работой

https://github.com/KulakhmetovS/Lab8

Приложение А

Листинг программы

Файл Lab8_Task1/main.c

```
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 40

using namespace std;

int** Creategraph(int **, int); //Создание графа
void BFS(int **, int *, int, int);  //Обход в ширину

int size;   //Размер графа


struct Queue
{
   int items[SIZE];
   int Front;
   int rear;
};

struct Queue* createQueue();
void enqueue(struct Queue* q, int);
int dequeue(struct Queue* q);
int isEmpty(struct Queue* q);

struct node
{
   int vertex;
   struct node* next;
};

struct node* createNode(int);

struct Graph
{
   int numVertices;
   struct node** adjLists;
   int* visited;
};

struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);


int main()
{
   int i, j, v, num;
   int **graph = NULL, *visited = NULL, *List = NULL;
```

```
    printf("\t# Graphs #\n\n");
    printf("Enter the number of graph vertices (positive integer): ");
    scanf("%d", &size);
    num = size - 1;

    // Creating the graph
    graph = Creategraph(graph, size);
    visited = (int *)(malloc(sizeof(int *) * size));  // Array for visited vertices
    List = (int *)(malloc(sizeof(int *) * size));

    // Printing the matrix
    for(i = 0; i < size; i++)
    {
       for(j = 0; j < size; j++)
          printf("%d ", graph[i][j]);
       printf("\n");
    }

    printf("Enter the number of start vertex (positive integer [0; %d]): ", num);
    scanf("%d", &v);

    // <---------- ! ---------->
    BFS(graph, visited, size, v);
    // <---------- ! ---------->
    printf("\n\n");


    struct Graph* Graph = createGraph(size);

    for(i = 0; i < size; i++)
    {
       for(j = i; j < size; j++)
       {
          if(graph[i][j] == 1)
          {
             addEdge(Graph, i, j);
          }
       }
    }

    printGraph(Graph);

    printf("Enter the number of start vertex (positive integer [0; %d]): ", num);
    scanf("%d", &v);

    // <---------- ! ---------->
    bfs(Graph, v);
    // <---------- ! ---------->



    return 0;
}

void BFS(int **graph, int *visited, int size, int v)
{
    queue<int> q;
    q.push(v);
    visited[v] = 1;
```

```
    while(!q.empty())
    {
        v = q.front();
        printf("%d ", v);
        q.pop();

        for(int i = 0; i < size; i++)
        {
            if((graph[v][i] == 1) && (visited[i] != 1))
            {
                q.push(i);
                visited[i] = 1;
            }
        }
    }

}

int** Creategraph(int **graph, int size)
{
    srand(time(NULL));

    int i = 0, j = 0;

    // Memory allocation
    graph = (int **)(malloc(sizeof(int *) * size));
    for(i = 0; i < size; i++)
        graph[i] = (int *)(malloc(sizeof(int *) * size));

    // Filling the matrix
    for(i = 0; i < size; i++)
        for(j = i; j < size; j++)
        {
            graph[i][j] = rand() % 2;
            graph[j][i] = graph[i][j];
            if(i == j) graph[i][j] = 0;
            if(graph[i][j] == 1);
        }

    return graph;
}


//-------------------------------------------------------


    void bfs(struct Graph* graph, int startVertex)
    {
        struct Queue* q = createQueue();

        graph->visited[startVertex] = 1;
        enqueue(q, startVertex);

        while(!isEmpty(q))
        {
            int currentVertex = dequeue(q);
            printf("%d ", currentVertex);

           struct node* temp = graph->adjLists[currentVertex];
```

```
      while(temp)
      {
          int adjVertex = temp->vertex;
          if(graph->visited[adjVertex] == 0)
          {
             graph->visited[adjVertex] = 1;
             enqueue(q, adjVertex);
          }
          temp = temp->next;
      }
   }
}


struct node* createNode(int v)
{
   struct node* newNode = new struct node;
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

struct Graph* createGraph(int vertices)
{
   struct Graph* graph = new struct Graph;
   graph->numVertices = vertices;

   graph->adjLists = new struct node*[vertices];
   graph->visited = new int[vertices];


   int i;
   for (i = 0; i < vertices; i++)
   {
      graph->adjLists[i] = NULL;
      graph->visited[i] = 0;
   }

   return graph;
}

void addEdge(struct Graph* graph, int src, int dest)
{
   // Add edge from src to dest
   struct node* newNode = createNode(dest);
   newNode->next = graph->adjLists[src];
   graph->adjLists[src] = newNode;

   // Add edge from dest to src
   newNode = createNode(src);
   newNode->next = graph->adjLists[dest];
   graph->adjLists[dest] = newNode;
}
struct Queue* createQueue()
{
   struct Queue* q = new struct Queue;
   q->Front = -1;
   q->rear = -1;
   return q;
```

```
    }
    int isEmpty(struct Queue* q)
    {
        if(q->rear == -1)
            return 1;
        else
            return 0;
    }
    void enqueue(struct Queue* q, int value)
    {
        if(q->rear == SIZE-1)
            printf("\nQueue is Full!!");
        else {
            if(q->Front == -1)
                q->Front = 0;
            q->rear++;
            q->items[q->rear] = value;
        }
    }
    int dequeue(struct Queue* q)
    {
        int item;
        if(isEmpty(q))
        {
            printf("Queue is empty");
            item = -1;
        }
        else{
            item = q->items[q->Front];
            q->Front++;
            if(q->Front > q->rear)
            {
                //printf("Resetting queue");
                q->Front = q->rear = -1;
            }
        }
        return item;
    }

    void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node* temp = graph->adjLists[v];
        printf("%d:  ", v);
        while (temp)
        {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

Файл Lab8_Task2/main.c

```
#include <iostream>
#include <queue>
```

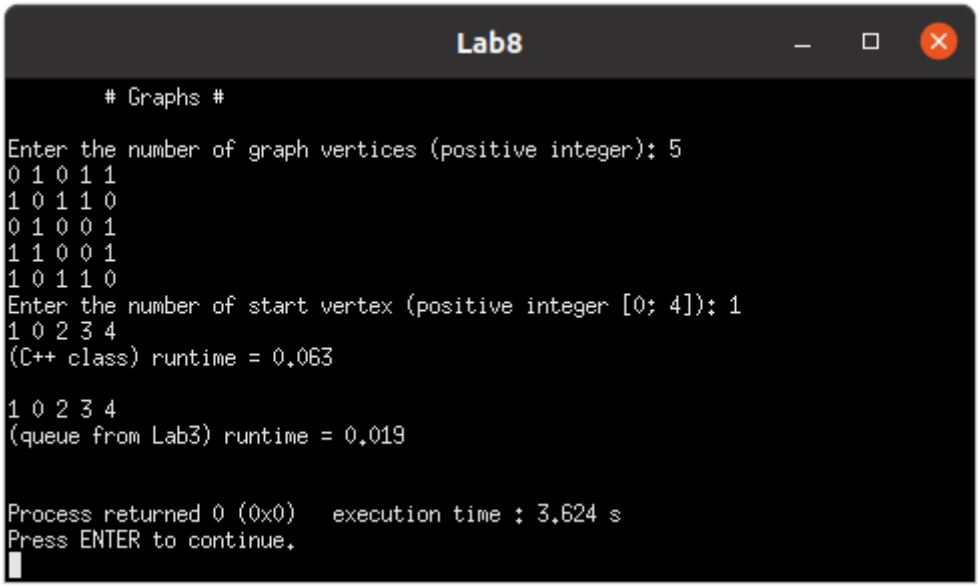
```
#include <cstdlib>
#include <ctime>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

using namespace std;

int** Creategraph(int **, int); //Создание графа
void BFS(int **, int *, int, int);

int size;   //Размер графа


struct Queue;   // Структура, отвечающая за элементы очереди
struct Queue *init(int);    // Инициализация очереди
void Push(struct Queue **, int);    // Добавление элемента в очередь
struct Queue *Pop(struct Queue *);      // Чтение элемента из очереди с последующим удалением

int res = 0;    // Результат извлеения из очереди


void bfs(struct Queue*, int**, int*, int, int);


int main()
{
   int i, j, v, num;
   int **graph = NULL, *visited = NULL, *List = NULL;

   printf("\t# Graphs #\n\n");
   printf("Enter the number of graph vertices (positive integer): ");
   scanf("%d", &size);
   num = size - 1;

   // Creating the graph
   graph = Creategraph(graph, size);
   visited = (int *)(malloc(sizeof(int *) * size));  // Array for visited vertices
   List = (int *)(malloc(sizeof(int *) * size));

   // Printing the matrix
   for(i = 0; i < size; i++)
   {
      for(j = 0; j < size; j++)
         printf("%d ", graph[i][j]);
      printf("\n");
   }

   printf("Enter the number of start vertex (positive integer [0; %d]): ", num);
   scanf("%d", &v);

   unsigned int start_time = clock();
   BFS(graph, visited, size, v);   //Native queue
   unsigned int end_time = clock();
   unsigned int search_time = end_time - start_time;
   cout << "\n(C++ class) runtime = " << search_time / 1000.0;
   printf("\n\n");

   struct Queue *list = NULL;
```

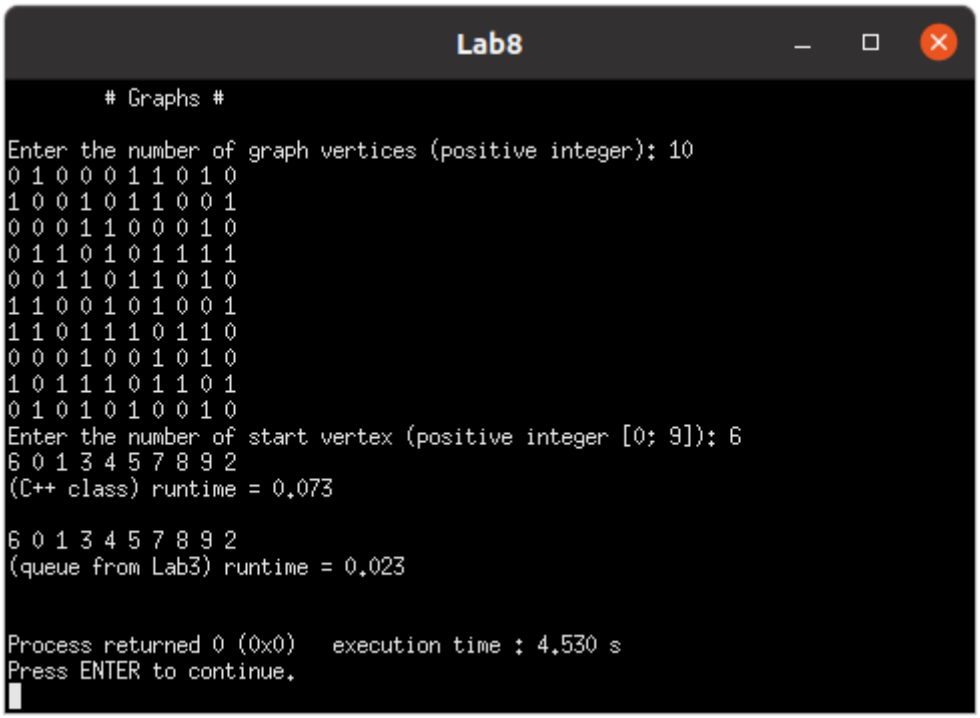
```
    for(i = 0; i < size; i++)
        visited[i] = 0;

    start_time = clock();
    bfs(list, graph, visited, size, v); //My queue
    end_time = clock();
    search_time = end_time - start_time;
    cout << "\n(queue from Lab3) runtime = " << search_time / 1000.0;
    printf("\n\n");

    return 0;
}


void BFS(int **graph, int *visited, int size, int v)
{
    queue<int> q;
    q.push(v);
    visited[v] = 1;

    while(!q.empty())
    {
        v = q.front();
        printf("%d ", v);
        q.pop();

        for(int i = 0; i < size; i++)
        {
            if((graph[v][i] == 1) && (visited[i] != 1))
            {
                q.push(i);
                visited[i] = 1;
            }
        }
    }

}

int** Creategraph(int **graph, int size)
{
    srand(time(NULL));

    int i = 0, j = 0;

    // Memory allocation
    graph = (int **)(malloc(sizeof(int *) * size));
    for(i = 0; i < size; i++)
        graph[i] = (int *)(malloc(sizeof(int *) * size));

    // Filling the matrix
    for(i = 0; i < size; i++)
        for(j = i; j < size; j++)
        {
            graph[i][j] = rand() % 2;
            graph[j][i] = graph[i][j];
            if(i == j) graph[i][j] = 0;
            if(graph[i][j] == 1);
        }
```

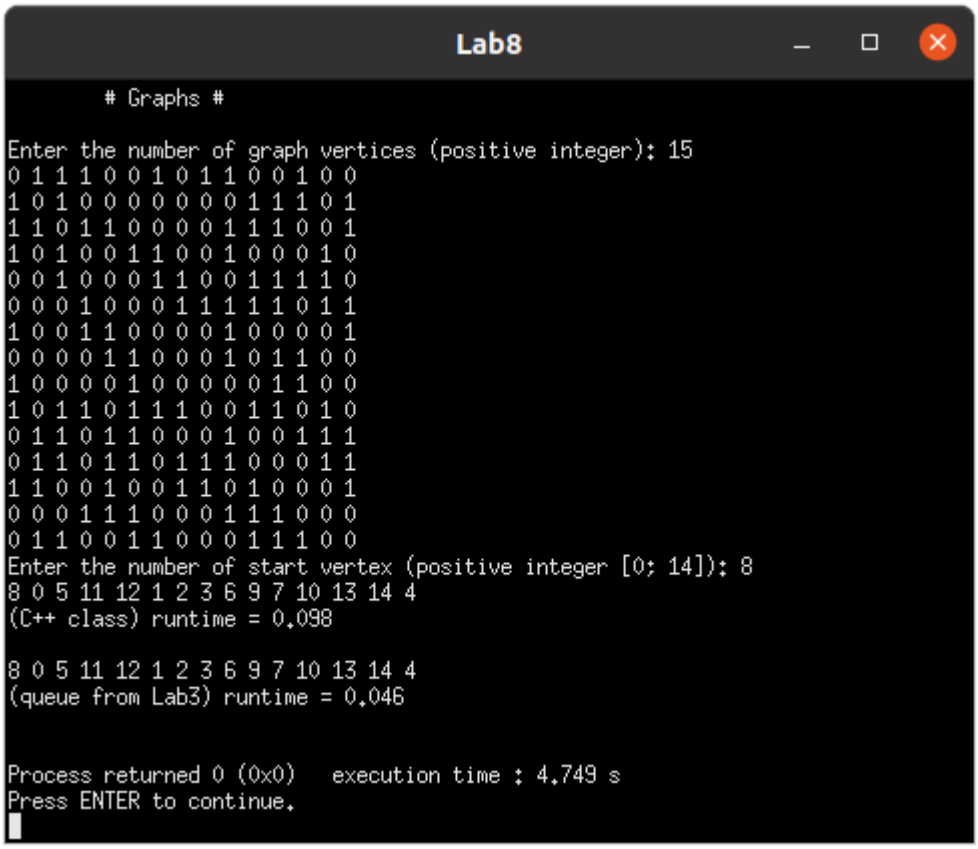
```
    return graph;
}



struct Queue
{
    int data;   // Числовой элемент очереди
    struct Queue *next; // Указатель на следующий элемент очереди
};

struct Queue *init(int element)
{
    struct Queue *p = NULL; // Создание указателя на структуру

    if((p = new struct Queue) == NULL)  // Выделение памяти под структуру
    {
        printf("Unable to allocate memory: ");
        exit(1);
    }

    p -> data = element;    // Присваивание введённого значения полю данных
    p -> next = NULL;   // Установка на нулевой указатель

    return p;
}

void Push(struct Queue **list ,int element)
{
    struct Queue *tmp = *list;  // Сохранение оригинального указателя на голову

    if(tmp != NULL) // Проверка на существование списка
    {
    struct Queue *new_element = init(element);  // Создание нового элемента

    while(tmp -> next != NULL)
    {
        tmp = tmp -> next;
    }

    tmp -> next = new_element;
    }
    else if(tmp == NULL)    // Инициализация списка, если его нет
    {
        *list = init(element);
    }
}

struct Queue *Pop(struct Queue *list)
{
    res = list -> data; // Получение элемента очереди

    struct Queue *to_delete = list; // Переназначение указателя на первый элемент
    list = list -> next;    //Переназначение первого указателя на следующий
    free(to_delete);    // Очистка памяти по предыдущему указателю

    return list;    // Возвращение нового указателя на вершину очереди
}

void bfs(struct Queue* list, int **graph, int *visited, int size, int v)
```

```
{
   Push(&list, v);
   visited[v] = 1;

   while(list != NULL)
   {
      list = Pop(list);
      v = res;
      printf("%d ", v);

      for(int i = 0; i < size; i++)
      {
         if((graph[v][i] == 1) && (visited[i] != 1))
         {
            Push(&list, i);
            visited[i] = 1;
         }
      }
   }
}
```

Приложение Б
Замеры времени обхода графа в ширину






Lab8
Graphs
Enter the number of graph vertices (positive integer): 15
0 1 1 1 0 0 1 0 1 1 0 0 1 0 0
1 0 1 0 0 0 0 0 0 0 1 1 1 0 1
1 1 0 1 1 0 0 0 0 1 1 1 0 0 1
1 0 1 0 0 1 1 0 0 1 0 0 0 1 0
0 0 1 0 0 0 1 1 0 0 1 1 1 1 0
0 0 0 1 0 0 0 1 1 1 1 1 0 1 1
1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1
0 0 0 0 1 1 0 0 0 1 0 1 1 0 0
1 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0
1 0 1 1 0 1 1 1 0 0 1 1 0 1 0
0 1 1 0 1 1 0 0 0 1 0 0 1 1 1
0 1 1 0 1 1 0 1 1 1 0 0 0 1 1
1 1 0 0 1 0 0 1 1 0 1 0 0 0 1
0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
0 1 1 0 0 1 1 0 0 0 1 1 1 0 0
Enter the number of start vertex (positive integer [0; 14]): 8
8 0 5 11 12 1 2 3 6 9 7 10 13 14 4
(C++ class) runtime = 0.098
8 0 5 11 12 1 2 3 6 9 7 10 13 14 4
(queue from Lab3) runtime = 0.046
Process returned 0 (0x0) execution time : 4.749 s
Press ENTER to continue.