Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Космических и информационных технологий
институт
Кафедра «Информатика»
кафедра

ОТЧЕТ О ПРАКТИЧЕСКОЙ РАБОТЕ

Задание 6 - Обход графов
тема

Преподаватель — подпись, дата — Р.Ю. Царев (инициалы, фамилия)

Студент КИ18-17/1б 031831229 — номер группы, зачетной книжки — подпись, дата — В.А. Прекель (инициалы, фамилия)

Красноярск 2020

1 Цель работы с постановкой задачи

1.1 Цель работы

Реализовать один из алгоритмов обхода графа: обход в глубину или обход в ширину (по выбору студента).

Программа должна наглядно отображать работы алгоритма.

1.2 Задача работы

Требования к выполнению лабораторной работы:

1. Самостоятельные разработка, тестирование и отладка программы.

2. Строгое соответствие программы и результатов ее работы с полученным заданием.

3. Устойчивость работы программы при любых воздействиях, задаваемых пользователем через интерфейс программы.

4. Предоставление демонстрационных примеров и исходного текста программы для защиты.

5. Предоставление отчета по лабораторной работе, содержащего описание реализованного алгоритма, программы, результатов работы программы.

Условия сдачи лабораторной работы:

- Знание теории по сдаваемому алгоритму.

- Умение объяснить полученные результаты.

- Способность быстро продемонстрировать на компьютере владение предметной областью.

2 Описание реализованного алгоритма

Реализован алгоритм поиска в глубину и ширину. Программа выводит вершины, в каком порядке прошёл поиск. Реализовано на языке C#. Написаны юнит-тесты для различных типов данных используя фреймворк NUnit.

3 Описание программы (листинги кода)

Листинг 1 – Alg_06/Alg_06.Core/Edges.cs

```
      using System;
using System.Collections.Generic;

namespace Alg_06.Core
{
    public class Edges<T> : SortedSet<Edge<T>>
        where T : IComparable
    {
        public override string ToString() => String.Join("; ", this);
    }
}
```

Листинг 2 – Alg_06/Alg_06.Core/Vertex.cs

```
      using System;

namespace Alg_06.Core
{
    public class Vertex<T> : Edges<T>, IComparable, IEquatable<Vertex<T>>
        where T : IComparable
    {
        public Vertex(T value) => Value = value;

        public T Value { get; }

        public int CompareTo(object obj)
        {
            var v = (Vertex<T>) obj;

            return Value.CompareTo(v.Value);
        }

        public bool Equals(Vertex<T>? other)
        {
            if (ReferenceEquals(null, other))
            {
                return false;
            }

            if (ReferenceEquals(this, other))
            {
                return true;
            }

            return Value.CompareTo(other.Value) == 0;
        }

        public override string ToString() => $"{Value}";

        public override bool Equals(object? obj)
        {
            if (ReferenceEquals(null, obj))
            {
                return false;
            }

            if (ReferenceEquals(this, obj))
            {
                return true;
            }
```

```
            if (obj.GetType() != GetType())
            {
                return false;
            }

            return Equals((Vertex<T>) obj);
        }

        public override int GetHashCode() => Value.GetHashCode();

        public static bool operator ==(Vertex<T>? left, Vertex<T>? right) =>
Equals(left, right);

        public static bool operator !=(Vertex<T>? left, Vertex<T>? right) =>
!Equals(left, right);
    }
}
```

Листинг 3 – Alg_06/Alg_06.Core/Vertices.cs

```
      using System;
using System.Collections.Generic;

namespace Alg_06.Core
{
    public class Vertices<T> : SortedDictionary<T, Vertex<T>>
        where T : IComparable
    {
        public override string ToString() => String.Join("; ", Values);
    }
}
```

Листинг 4 – Alg_06/Alg_06.Core/Edge.cs

```
      using System;

namespace Alg_06.Core
{
    public class Edge<T> : Tuple<Vertex<T>, Vertex<T>>, IComparable,
IEquatable<Edge<T>>
        where T : IComparable
    {
        public Edge(Vertex<T> item1, Vertex<T> item2) : base(item1, item2)
        {
        }


        public Vertex<T> LessVertex => Item1.CompareTo(Item2) > 0 ? Item2 :
Item1;
        public Vertex<T> GreatVertex => Item1.CompareTo(Item2) > 0 ? Item1 :
Item2;

        public int CompareTo(object obj)
        {
            var e = (Edge<T>) obj;

            var c1 = LessVertex.CompareTo(e.LessVertex);
            var c2 = GreatVertex.CompareTo(e.GreatVertex);
```

```
            return c1 == 0 ? c2 : c1;
        }

        public bool Equals(Edge<T>? other) =>
            other != null && LessVertex.Equals(other.LessVertex) &&
GreatVertex.Equals(other.GreatVertex);

        public bool HasVertex(Vertex<T> v) => Item1 == v || Item2 == v;

        public Vertex<T>? OtherVertex(Vertex<T> v) => v == Item1 ? Item2 : v ==
Item2 ? Item1 : null;

        public override string ToString() => $"{Item1} - {Item2}";

        public override bool Equals(object? obj)
        {
            if (ReferenceEquals(null, obj))
            {
                return false;
            }

            if (ReferenceEquals(this, obj))
            {
                return true;
            }

            if (obj.GetType() != GetType())
            {
                return false;
            }

            return Equals((Edge<T>) obj);
        }

        public override int GetHashCode() => base.GetHashCode();

        public static bool operator ==(Edge<T>? left, Edge<T>? right) =>
Equals(left, right);

        public static bool operator !=(Edge<T>? left, Edge<T>? right) =>
!Equals(left, right);
    }
}
```

Листинг 5 – Alg_06/Alg_06.Core/Graph.cs

```
      using System;
using System.Collections.Generic;
using System.Linq;

namespace Alg_06.Core
{
    public class Graph<T> : Tuple<Vertices<T>, Edges<T>>
        where T : IComparable
    {
        public Graph() : base(new Vertices<T>(), new Edges<T>())
        {
        }

        public Vertices<T> V => Item1;
```

```
        public Edges<T> E => Item2;

        public Vertex<T> AddVertex(T value)
        {
            V[value] = new Vertex<T>(value);
            return V[value];
        }

        public void RemoveVertex(T value)
        {
            E.RemoveWhere(e => e.HasVertex(V[value]));
            V.Remove(value);
        }

        public void RemoveVertex(Vertex<T> v)
        {
            E.RemoveWhere(e => e.HasVertex(v));
            V.Remove(v.Value);
        }

        public void RemoveEdge(Edge<T> e)
        {
            V[e.Item1.Value].RemoveWhere(ed => ed.CompareTo(e) == 0);
            V[e.Item2.Value].RemoveWhere(ed => ed.CompareTo(e) == 0);
            E.RemoveWhere(ed => ed.CompareTo(e) == 0);
        }

        public void RemoveEdge(T value1, T value2)
        {
            var v1 = V[value1];
            var v2 = V[value2];
            v1.RemoveWhere(ed => ed.HasVertex(v1) && ed.HasVertex(v2));
            v2.RemoveWhere(ed => ed.HasVertex(v1) && ed.HasVertex(v2));
            E.RemoveWhere(ed => ed.HasVertex(v1) && ed.HasVertex(v2));
        }

        public Edge<T> AddEdge(T value1, T value2) => AddEdge(V[value1],
V[value2]);

        public Edge<T> AddEdge(Vertex<T> v1, Vertex<T> v2)
        {
            var e = new Edge<T>(v1, v2);
            E.Add(e);
            v1.Add(e);
            v2.Add(e);
            return e;
        }

        public void Dfs(T value, Action<Vertex<T>> action) => Dfs(V[value],
action);

        public void Dfs(Vertex<T> v, Action<Vertex<T>> action) =>
            Dfs(v, action, new SortedDictionary<Vertex<T>, bool>());

        private static void Dfs(Vertex<T> v, Action<Vertex<T>> action,
IDictionary<Vertex<T>, bool> visited)
        {
            visited[v] = true;
            action.Invoke(v);
            foreach (var next in UnvisitedVertices(v, visited))
            {
                Dfs(next, action, visited);
            }
```

```
        }

        public void Bfs(T value, Action<Vertex<T>> action) => Bfs(V[value],
action);

        public void Bfs(Vertex<T> v, Action<Vertex<T>> action) =>
            Bfs(v, action, new SortedDictionary<Vertex<T>, bool>());

        private static void Bfs(Vertex<T> v, Action<Vertex<T>> action,
IDictionary<Vertex<T>, bool> visited)
        {
            var queue = new Queue<Vertex<T>>();
            visited[v] = true;
            queue.Enqueue(v);
            while (queue.Count > 0)
            {
                var cv = queue.Dequeue();
                action.Invoke(cv);
                foreach (var next in UnvisitedVertices(cv, visited))
                {
                    visited[next] = true;
                    queue.Enqueue(next);
                }
            }
        }

        private static IEnumerable<Vertex<T>> UnvisitedVertices(Vertex<T> v,
IDictionary<Vertex<T>, bool> visited) => v
            .Select(i => i.OtherVertex(v) ?? v)
            .Where(next => !(visited.ContainsKey(next) && visited[next]));

        public override string ToString() => $"V: {String.Join(", ", V.Values)};
E: {String.Join(", ", E)}";
    }
}
```

Листинг 6 – Alg_06/Alg_06.Core.Tests/GraphTests.cs

```
      using System.Collections.Generic;
using System.Linq;

using NUnit.Framework;

namespace Alg_06.Core.Tests
{
    public class GraphTests
    {
        [Test]
        public void GraphTest1()
        {
            //  0 - 1
            //  | /
            //  2 - 3
            var g = new Graph<int>();
            Assert.That(g.E.Count, Is.EqualTo(0));
            Assert.That(g.V.Count, Is.EqualTo(0));
            var v0 = g.AddVertex(0);
            Assert.That(g.E.Count, Is.EqualTo(0));
            Assert.That(g.V.Count, Is.EqualTo(1));
            Assert.That(g.V.ContainsKey(0), Is.True);
            Assert.That(g.V.ContainsValue(v0), Is.True);
```

```
            Assert.That(g.V[0].Value, Is.EqualTo(0));
            var v1 = g.AddVertex(1);
            Assert.That(g.E.Count, Is.EqualTo(0));
            Assert.That(g.V.Count, Is.EqualTo(2));
            Assert.That(g.V.ContainsKey(1), Is.True);
            Assert.That(g.V.ContainsValue(v1), Is.True);
            Assert.That(g.V[1].Value, Is.EqualTo(1));
            var e01 = g.AddEdge(0, 1);
            Assert.That(g.E.Count, Is.EqualTo(1));
            Assert.That(g.V.Count, Is.EqualTo(2));
            Assert.That(g.E.Contains(e01), Is.True);
            Assert.That(g.V[0].Contains(e01), Is.True);
            Assert.That(g.V[1].Contains(e01), Is.True);
            Assert.That(g.V[0].First(e => e == e01), Is.EqualTo(e01));
            Assert.That(g.V[1].First(e => e == e01), Is.EqualTo(e01));
            var v2 = g.AddVertex(2);
            Assert.That(g.E.Count, Is.EqualTo(1));
            Assert.That(g.V.Count, Is.EqualTo(3));
            Assert.That(g.V.ContainsKey(2), Is.True);
            Assert.That(g.V.ContainsValue(v2), Is.True);
            Assert.That(g.V[2].Value, Is.EqualTo(2));
            var e02 = g.AddEdge(0, 2);
            Assert.That(g.E.Count, Is.EqualTo(2));
            Assert.That(g.V.Count, Is.EqualTo(3));
            Assert.That(g.E.Contains(e02), Is.True);
            Assert.That(g.V[0].Contains(e02), Is.True);
            Assert.That(g.V[2].Contains(e02), Is.True);
            Assert.That(g.V[0].First(e => e == e02), Is.EqualTo(e02));
            Assert.That(g.V[2].First(e => e == e02), Is.EqualTo(e02));
            var e12 = g.AddEdge(1, 2);
            Assert.That(g.E.Count, Is.EqualTo(3));
            Assert.That(g.V.Count, Is.EqualTo(3));
            Assert.That(g.E.Contains(e12), Is.True);
            Assert.That(g.V[1].Contains(e12), Is.True);
            Assert.That(g.V[2].Contains(e12), Is.True);
            Assert.That(g.V[1].First(e => e == e12), Is.EqualTo(e12));
            Assert.That(g.V[2].First(e => e == e12), Is.EqualTo(e12));
            var v3 = g.AddVertex(3);
            Assert.That(g.E.Count, Is.EqualTo(3));
            Assert.That(g.V.Count, Is.EqualTo(4));
            Assert.That(g.V.ContainsKey(3), Is.True);
            Assert.That(g.V.ContainsValue(v3), Is.True);
            Assert.That(g.V[3].Value, Is.EqualTo(3));
            var e23 = g.AddEdge(2, 3);
            Assert.That(g.E.Count, Is.EqualTo(4));
            Assert.That(g.V.Count, Is.EqualTo(4));
            Assert.That(g.E.Contains(e23), Is.True);
            Assert.That(g.V[2].Contains(e23), Is.True);
            Assert.That(g.V[3].Contains(e23), Is.True);
            Assert.That(g.V[2].First(e => e == e23), Is.EqualTo(e23));
            Assert.That(g.V[3].First(e => e == e23), Is.EqualTo(e23));

            var dfs0 = new List<Vertex<int>>();
            g.Dfs(0, v => dfs0.Add(v));
            Assert.That(dfs0.Count, Is.EqualTo(4));
            Assert.That(dfs0.Select(v => v.Value), Is.EquivalentTo(new[] {0, 1,
2, 3}));

            var dfs1 = new List<Vertex<int>>();
            g.Dfs(1, v => dfs1.Add(v));
            Assert.That(dfs1.Count, Is.EqualTo(4));
            Assert.That(dfs1.Select(v => v.Value), Is.EquivalentTo(new[] {1, 0,
2, 3}));
```

```
            var dfs2 = new List<Vertex<int>>();
            g.Dfs(2, v => dfs2.Add(v));
            Assert.That(dfs2.Count, Is.EqualTo(4));
            Assert.That(dfs2.Select(v => v.Value), Is.EquivalentTo(new[] {2, 0,
1, 3}));

            var dfs3 = new List<Vertex<int>>();
            g.Dfs(3, v => dfs3.Add(v));
            Assert.That(dfs3.Count, Is.EqualTo(4));
            Assert.That(dfs3.Select(v => v.Value), Is.EquivalentTo(new[] {3, 2,
0, 1}));

            var bfs0 = new List<Vertex<int>>();
            g.Bfs(0, v => bfs0.Add(v));
            Assert.That(bfs0.Count, Is.EqualTo(4));
            Assert.That(bfs0.Select(v => v.Value), Is.EquivalentTo(new[] {0, 1,
2, 3}));

            var bfs1 = new List<Vertex<int>>();
            g.Bfs(1, v => bfs1.Add(v));
            Assert.That(bfs1.Count, Is.EqualTo(4));
            Assert.That(bfs1.Select(v => v.Value), Is.EquivalentTo(new[] {1, 0,
2, 3}));

            var bfs2 = new List<Vertex<int>>();
            g.Bfs(2, v => bfs2.Add(v));
            Assert.That(bfs2.Count, Is.EqualTo(4));
            Assert.That(bfs2.Select(v => v.Value), Is.EquivalentTo(new[] {2, 0,
1, 3}));

            var bfs3 = new List<Vertex<int>>();
            g.Bfs(3, v => bfs3.Add(v));
            Assert.That(bfs3.Count, Is.EqualTo(4));
            Assert.That(bfs3.Select(v => v.Value), Is.EquivalentTo(new[] {3, 2,
0, 1}));

            g.RemoveEdge(0, 2);
            Assert.That(g.E.Count, Is.EqualTo(3));
            Assert.That(g.V.Count, Is.EqualTo(4));
            Assert.That(g.E.Contains(e02), Is.False);
            Assert.That(g.V[0].Contains(e02), Is.False);
            Assert.That(g.V[2].Contains(e02), Is.False);
            Assert.That(g.V[0].FirstOrDefault(e => e == e02), Is.Null);
            Assert.That(g.V[2].FirstOrDefault(e => e == e02), Is.Null);

            g.RemoveVertex(1);
            Assert.That(g.E.Count, Is.EqualTo(1));
            Assert.That(g.V.Count, Is.EqualTo(3));
            Assert.That(g.V.ContainsKey(1), Is.False);
            Assert.That(g.V.ContainsValue(v1), Is.False);
        }

        [Test]
        public void GraphTest2()
        {
            var g = new Graph<int>();
            var v = Enumerable.Range(0, 8).Select(i => g.AddVertex(i)).ToList();
            g.AddEdge(1, 2);
            g.AddEdge(0, 2);
            g.AddEdge(0, 3);
            g.AddEdge(4, 3);
            g.AddEdge(2, 4);
```

```
            g.AddEdge(2, 7);
            g.AddEdge(7, 4);
            g.AddEdge(6, 7);
            g.AddEdge(6, 4);
            g.AddEdge(5, 6);

            var dfs0 = new List<Vertex<int>>();
            g.Dfs(0, v1 => dfs0.Add(v1));
            Assert.That(dfs0.Select(v1 => v1.Value), Is.EquivalentTo(new[] {0,
2, 1, 4, 3, 6, 5, 7}));

            var bfs0 = new List<Vertex<int>>();
            g.Bfs(0, v1 => bfs0.Add(v1));
            Assert.That(bfs0.Select(v1 => v1.Value), Is.EquivalentTo(new[] {0,
2, 3, 1, 4, 7, 6, 5}));
        }

        [Test]
        public void GraphTest3()
        {
            var g = new Graph<string>();
            g.AddVertex("First");
            g.AddVertex("Second");
            g.AddVertex("Third");
            g.AddVertex("Fourth");

            g.AddEdge("First", "Second");
            g.AddEdge("First", "Fourth");

            var dfs0 = new List<Vertex<string>>();
            g.Dfs("First", v1 => dfs0.Add(v1));
            Assert.That(dfs0.Select(v1 => v1.Value), Is.EquivalentTo(new[]
{"First", "Fourth", "Second"}));

            var bfs0 = new List<Vertex<string>>();
            g.Bfs("First", v1 => bfs0.Add(v1));
            Assert.That(bfs0.Select(v1 => v1.Value), Is.EquivalentTo(new[]
{"First", "Fourth", "Second"}));
        }
    }
}
```

Листинг 7 – Alg_06/Alg_06.Console/Program.cs

```
      using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Alg_06.Core;

namespace Alg_06.Console
{
    public class Program
    {
        private static string SearchAndOut(Action<int, Action<Vertex<int>>>
search, int start, Graph<int> g)
        {
            var res = new List<int>();
            search(start, v => res.Add(v.Value));
            return String.Join(" ", res);
```

```
        }

        public static void Main(string[] args)
        {
            System.Console.InputEncoding = Encoding.UTF8;
            System.Console.OutputEncoding = Encoding.UTF8;

            var g = new Graph<int>();

            while (true)
            {
                try
                {
                    System.Console.WriteLine("Введите номера вершин через
пробел: ");
                    var a = System.Console.ReadLine()
                        .Split(new[] {" "},
StringSplitOptions.RemoveEmptyEntries)
                        .Select(Int32.Parse)
                        .Distinct()
                        .Select(g.AddVertex)
                        .ToList();

                    break;
                }
                catch (Exception e)
                {
                    System.Console.Error.WriteLine($"Ошибка: {e.Message}\n");
                }
            }

            while (true)
            {
                try
                {
                    System.Console.WriteLine("Вводите через Enter 2 номера
вершины через пробел, обозначающих ребро: ");

                    while (true)
                    {
                        var es = System.Console.ReadLine();
                        if (es == "")
                        {
                            break;
                        }

                        var esp = es.Split(new[] {" "},
StringSplitOptions.RemoveEmptyEntries)
                            .Select(Int32.Parse)
                            .ToList();

                        if (esp.Count == 2)
                        {
                            g.AddEdge(esp[0], esp[1]);
                        }
                    }

                    break;
                }
                catch (Exception e)
                {
                    System.Console.Error.WriteLine($"Ошибка: {e.Message}\n");
                }
```

```
            }

            while (true)
            {
                try
                {
                    System.Console.WriteLine($"Вершины графа: {g.V}");
                    System.Console.WriteLine($"Рёбра графа: {g.E}");

                    System.Console.WriteLine("С какой вершины начать обход?");
                    var s = Int32.Parse(System.Console.ReadLine());

                    System.Console.WriteLine($"Поиск в глубину:
{SearchAndOut(g.Dfs, s, g)}");
                    System.Console.WriteLine($"Поиск в ширину:
{SearchAndOut(g.Bfs, s, g)}");

                    break;
                }
                catch (Exception e)
                {
                    System.Console.Error.WriteLine($"Ошибка: {e.Message}\n");
                }
            }

            System.Console.ReadKey();
        }
    }
}
```

4 Результаты работы программы

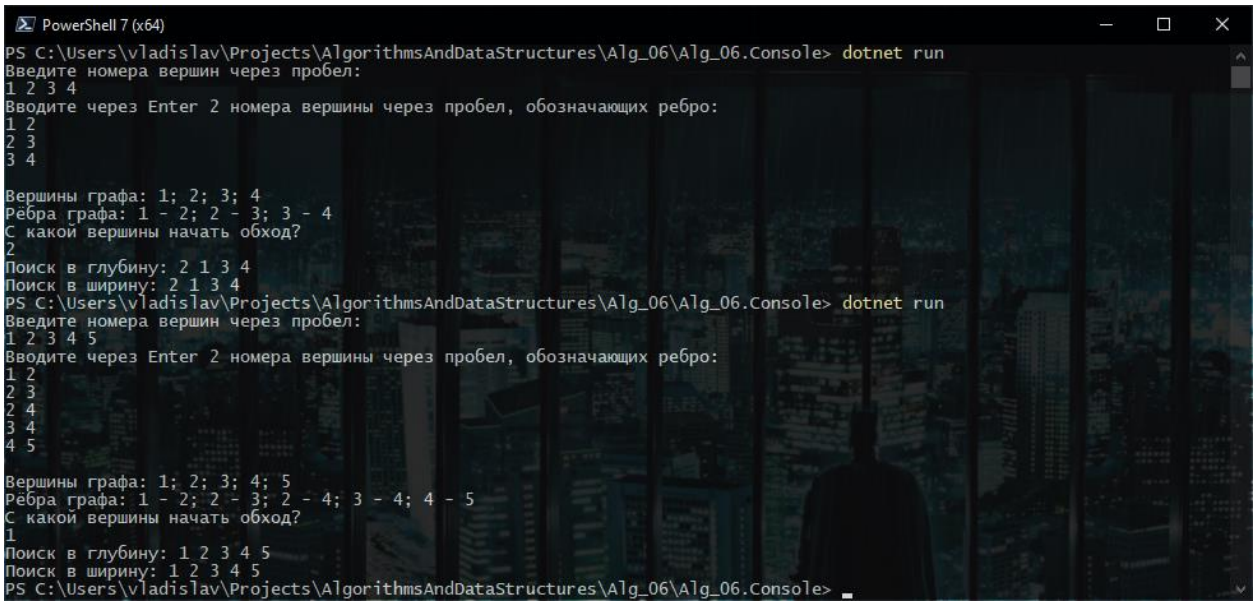

Рисунок 1 – Запуск 1

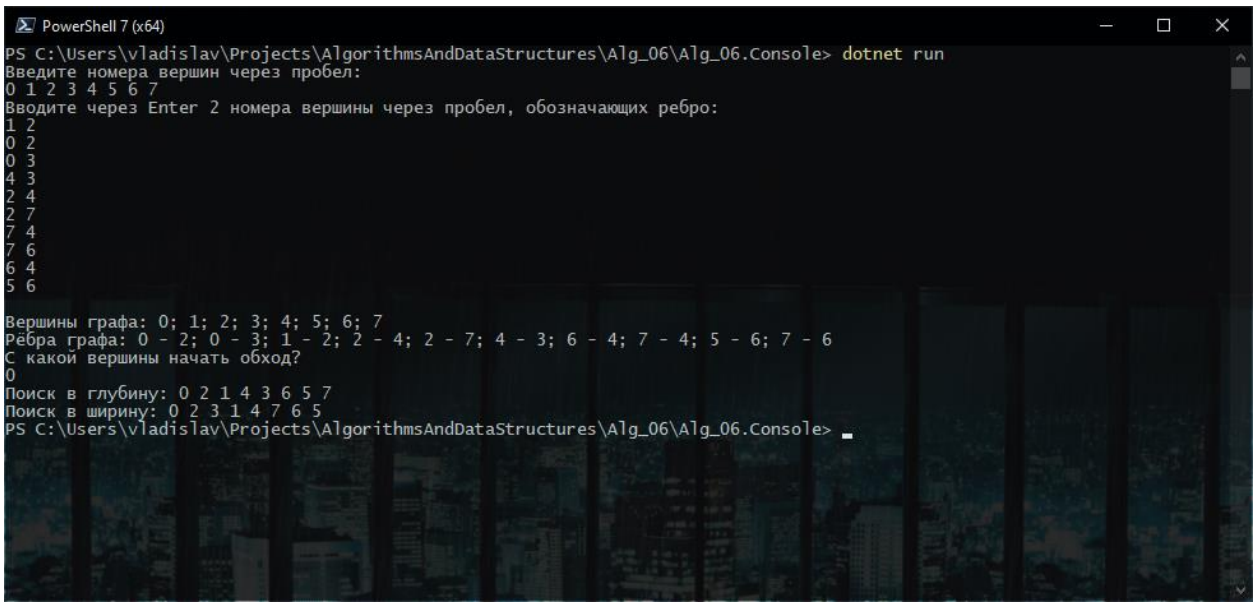

Рисунок 2 – Запуск 2

Рисунок 3 – Запуск тестов