МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«Новосибирский национальный исследовательский государственный университет»

(Новосибирский государственный университет, НГУ)

Структурное подразделение Новосибирского государственного университета –

Высший колледж информатики Университета (ВКИ НГУ)

КАФЕДРА ИНФОРМАТИКИ

**РЕАЛИЗАЦИЯ ПРОГРАММНОЙ БИБЛИОТЕКИ ДЛЯ РАБОТЫ С МНОГОУРОВНЕВЫМИ СЕТЯМИ**

Квалификация техник-программист

|  |  |
| --- | --- |
| Руководитель | Родионов A.С.  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_2022 г. |
| Студент 4 курса  803в2 | Новиков С.Я.  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2022 г. |
| Нормоконтроль | «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2022 г. |

Новосибирск

2022

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc1)

[1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ 5](#_Toc2)

[2 ПОСТАНОВКА ЗАДАЧИ 7](#_Toc3)

[3 АНАЛОГИ 8](#_Toc4)

[4 ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ К ПРОГРАММНОМУ СРЕДСТВУ 10](#_Toc5)

[5 НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ К ПРОГРАММНОМУ СРЕДСТВУ 11](#_Toc6)

[5.1 Требования к программному обеспечению 11](#_Toc7)

[5.2 Требования к аппаратному обеспечению 11](#_Toc8)

[5.3 Требования к надежности 11](#_Toc9)

[6 ХАРАКТЕРИСТИКА ВЫБРАННЫХ ПРОГРАММЫХ СРЕД И СРЕДСТВ 12](#_Toc10)

[7 АЛГОРИТМ РЕШЕНИЯ ЗАДАЧИ 13](#_Toc11)

[7.1 Математическая постановка 13](#_Toc12)

[7.2 Алгоритмы 13](#_Toc13)

[8 ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ 14](#_Toc14)

[8.1 Входные данные 14](#_Toc15)

[8.2 Выходные данные 14](#_Toc16)

[9 ОТЛАДКА И ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 15](#_Toc17)

[10 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 16](#_Toc18)

[ЗАКЛЮЧЕНИЕ 18](#_Toc19)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 19](#_Toc20)

[ПРИЛОЖЕНИ А 20](#_Toc21)

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ, УСЛОВНЫХ ОБОЗНАЧЕНИЙ И ТЕРМИНОВ

Граф - математическая абстракция реальной системы любой природы, объекты которой обладают парными связями. Граф как математический объект есть совокупность двух множеств — множества самих объектов, называемого множеством вершин и множества их парных связей, называемого множеством рёбер. Элемент множества рёбер есть пара элементов множества вершин.

Гиперсеть - математическая модель для описания сложных систем сетевой структуры, с помощью которой решают задачи проектирования и оптимизации различных сетей

# ВВЕДЕНИЕ

В большинстве случаев при анализе сетей различной природы (транспортных, дорожных, телекоммуникационных, социальных и пр.) используются графовые, иногда гиперграфовые, модели. Однако, многие сетевые системы включают себя связи на разных уровнях. Примеры многоуровневых сетей:

1. решение транспортных задач;
2. оптимизация структурированных кабельных сетей;
3. социальных взаимоотношений;

Каждая сеть имеет свою задачу анализа:

1. в транспортных: нахождение маршрута с пересадками;
2. в кабельных: анализ надёжности виртуальных соединений при возможных обрывах проводов;
3. в социальных: нахождений удалённых связей;

Есть различные модели многоуровневых сетей, одна из которых (абстрактные гиперсети) разработана в ИВМиМГ СО РАН профессором Попковым В.К. Эта модель применялась для решения различных задач проектирования и оптимизации сетей связи, транспортных сетей, структурированной кабельной канализации и др. При этом каждый автор заново описывал и реализовывал гиперсетевые структуры программно, используя как массивы и структуры данных в классических языках программирования типа Паскаль и C, так и объектное представление и списочные структуры в C++, C#, Java и Delfi. Всё это затрудняет создание единых библиотек и затрудняет преемственность в разработках.

Целью моей работы является создание единой системы объектов, позволяющей достаточно экономно по памяти и при этом удобно в использовании описывать гиперсети и использовать эти описания в решении различных задач.

Важным аспектом работы является разработка человекочитаемых текстовых форматов для загрузки-выгрузки данных гиперсетей. С помощью этой функции будет вводится или выводится вся информация о графе в нескольких представлениях, например, список инцидентности и матрица смежности.

# 1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Гиперсеть это математическая модель для описания сложных систем сетевой структуры, с помощью которой, решают задачи проектирования и оптимизации различных сетей, таких как: сети связи, транспортные сети, информационные сети и т.д. Данные задачи актуальны и востребованы. С ростом инфраструктур, подлежащий оптимизации, растет и объем данных, которые требуется обрабатывать. Кроме того, все эти задачи сугубо прикладные. А для прикладных задач характерно дальнейшая поддержка, модификация и документация. Данное обстоятельство диктует требования системы, которая будет агрегировать данные, получение после выполнения алгоритмов и по требованию выдавать их либо специалисту, либо другим алгоритмам для дальнейшей работы. На данные момент целостной системы и каких-либо стандартов для работы с данными, формализованными на языке теории гиперсетей не существует.

Ключевой особенностью гиперсетей является возможность описывания имеющих более 2 структур каждая из которых может быть вложена в другую. Например, рассматривая транспортную сеть города мы имеем дело сетью улиц, в которую отображены (вложены) множество маршрутов общественного транспорта. Традиционно, при решении задач в перечисленных областях использовались графы и гиперсети в качестве математической модели. Отдельные задачи хорошо изучены и решены в рамках теории графов. Но на практике, когда необходимо построить реальную систему, мы не можем абстрагироваться от всех взаимосвязей и внешних факторов, которые оказывают влияние на итоговую структуру гиперсети . В некоторых задачах, мы имеем дело с нестационарными структурами - структуры, которые изменяются с течением времени. В таких случаях поставить четкую математическую задачу практически невозможно. Даже сама формулировка окажется громоздкой, не говоря уже о решении. Либо придется пренебрегать каким-либо воздействием. и решать частную задачу. Для решения некоторого ряда задач можно использовать другие известные модели, такие как с вложенные графы. Но гиперсети являются более широкой моделью (вложенные графы могут быть описаны так же в рамках гиперсетей), которая может, достаточно лаконично описать сложные иерархические, многоуровневые сетевые модели.

# 2 ПОСТАНОВКА ЗАДАЧИ

На сегодняшний момент не существует универсального решения, для унификации взаимодействий с гиперсетями. Не существует единого стандарта их описания при реализации алгоритмов решения прикладных задач. Целью данной работы является создание программного комплекса, который:

1. позволит сохранять данные в виде гиперсети ;
2. обеспечит удобные механизмы получения и обновления данных;
3. обеспечит возможность частичного получения данных и в различных представлениях;

Для реализации поставленной цели, в рамках работы будут решены следующие задачи:

1. проектирование архитектуры программного комплекса;
2. разработка и реализация алгоритмов для генерации часто используемых представлений. Таких как: список инцидентности, матрица смежности;
3. разработка и реализация для общения пользователя с системой;

# 3 АНАЛОГИ

В интернете можно найти системы можно найти системы для отрисовки и расчета параметров сети, но они не предоставляют нужного функционала для гиперсетей, поэтому было принято решение создать свою библиотеку взаимодействия с гиперсетью . Примеры таких систем:

1. graphonline.ru - сайт для визуализации графа и применения на нем

встроенных алгоритмов.

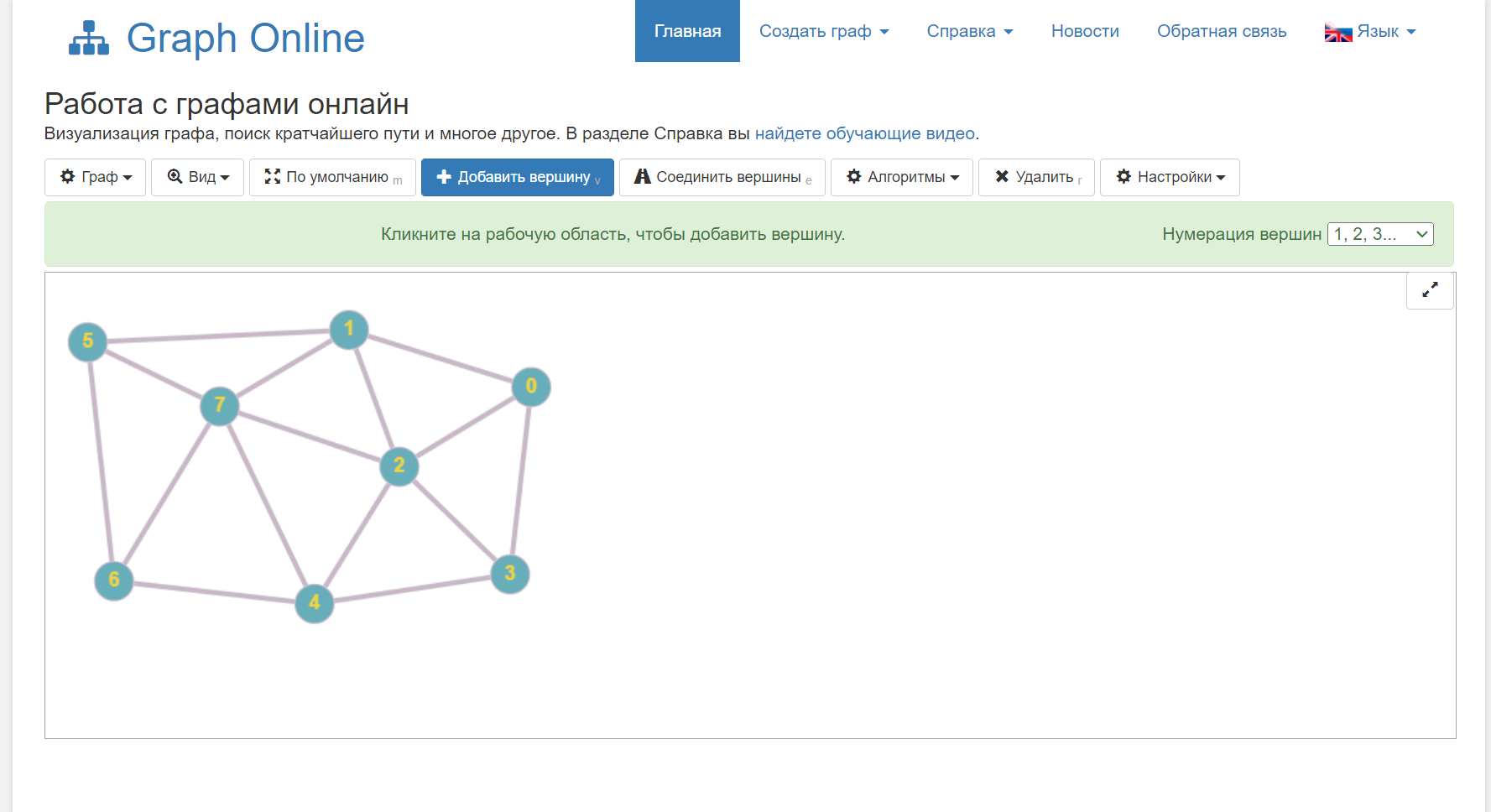


Рисунок 1 – Скриншот сайта graphonline.ru

Этот сайт имеет графический интерфейс и набор некоторых алгоритмов, но отсутствует поддержка гиперсетей, что делает его не пригодным для работы с гиперсетями.

1. programforyou.ru/graph-redactor – сайт для работы с графами.

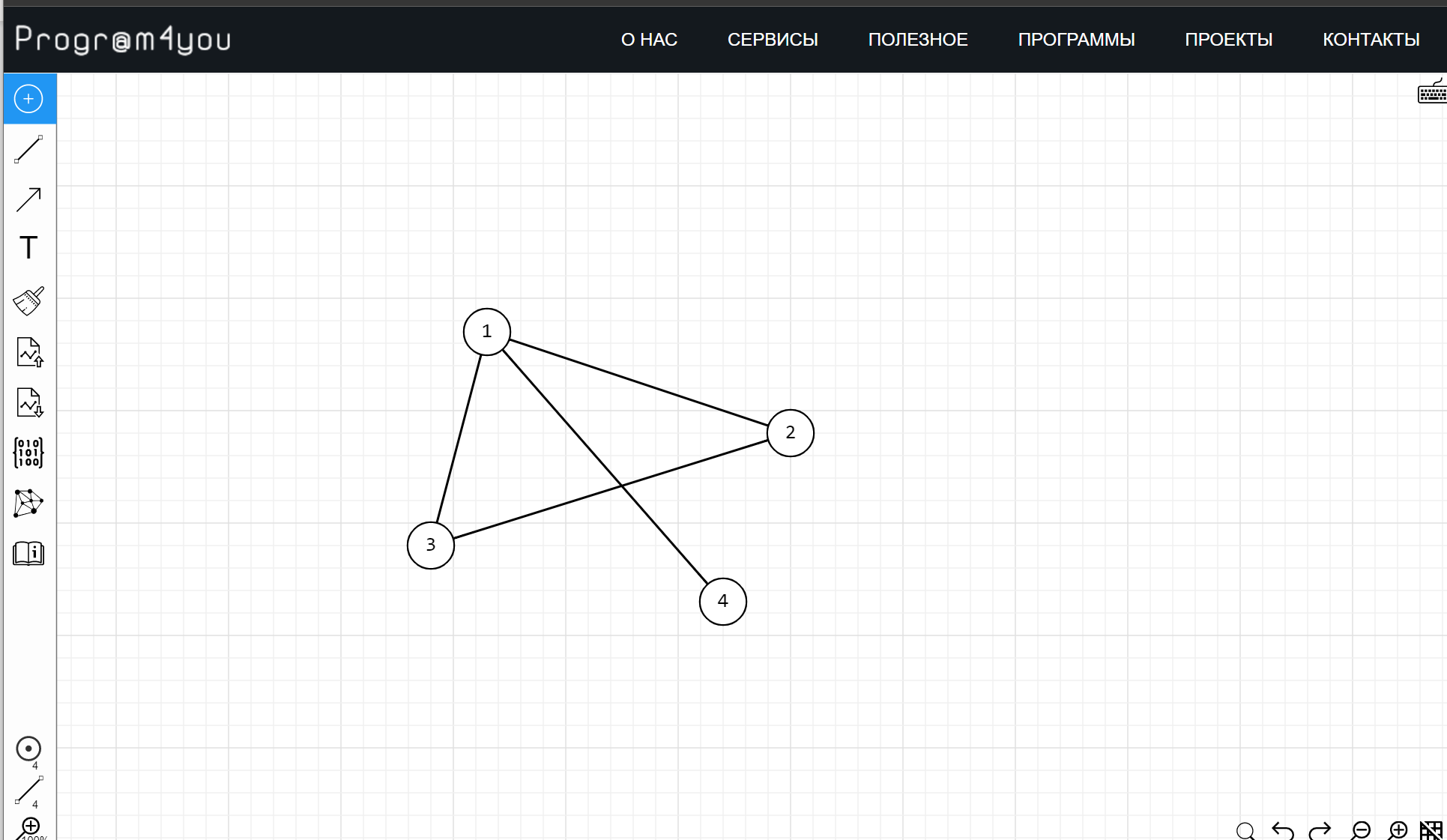


Рисунок 2 – Скриншот сайта programforyou.ru/graph-redactor

Сайт владеет более обширным функционалом, по сравнению с предыдущим сайтом, такие, как чтение или сохранение графа в файл, более обширный выбор представления графа. Но, как у предыдущего сайта, отсутствует поддержка гиперсетей .

# 4 ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ К ПРОГРАММНОМУ СРЕДСТВУ

К разрабатываемым модуля были выведены следующие функциональные требования:

1. создание первичной сети;
2. добавление графа вторичной сети в первичную сеть;
3. добавление ветвей в граф вторичной сети;
4. связывание вершин ребрами;
5. механизм удобного взаимодействия с гиперсетью;
6. сохранения гиперсети в постоянной памяти;
7. генерация часто используемых представлений графа;
8. загрузка-выгрузка из текстового формата данных гиперсетей;

# 5 НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ К ПРОГРАММНОМУ СРЕДСТВУ

## 5.1 Требования к программному обеспечению

Кроссплатформенность – возможность работы на следующих операционных системах:

* Windows 7 и новее.
* Linux с версией ядра 4.4 и новее.

## 5.2 Требования к аппаратному обеспечению

Низкие системные требования, включающие в себя работу на большинстве устройств, для обеспечения качественной работы в составе большинства проектов.

## 5.3 Требования к надежности

В процессе выполнения расчетов не должно возникать критических ошибок. Программа должна по результатам работы выдавать точный результат.

# 6 ХАРАКТЕРИСТИКА ВЫБРАННЫХ ПРОГРАММЫХ СРЕД И СРЕДСТВ

Для реализации модулей было используется язык C#, и среда разработки Visual Studio. Также в качестве системы управления версий был выбран Git.

C# – Объектно-ориентированный язык программирования, относящийся к C-подобным. Преимуществом данного языка можно считать быстродействие и безопасность. ближайшими конкурентами могут послужить Java, у которой есть проблемы со скоростью, и C++, который имеет прямой доступ к памяти, увеличивая количество ошибок при разработке приложения.

Microsoft Visual Studio – интегрированная среда разработки программного обеспечения. Предоставляет удобные средства для создания и редактирования исходного кода, и бесплатна для пользования студентам. Но из-за наличия большого количества интрументов и функционала, требует много оперативной памяти. Поддерживает множество плагинов для улучшения и упрощения процесса написания кода. Она является наиболее технологичной и практичной средой разработки на языке C#.

Git – Популярная система управления версиями, хорошо отличающаяся от конкурентов своей быстротой, безопасностью и удобством. С помощью него можно вносить изменения в проект, не боясь последствий, из-за возможности откатить их в прошлое, до текущих изменений состояния. Эта возможность позволяет вносить изменения, не боясь остаться с не рабочим приложением.

# 7 АЛГОРИТМ РЕШЕНИЯ ЗАДАЧИ

## 7.1 Математическая постановка

Простая гиперсеть H = (X,V,R;P,F,W) состоит из следующих объектов (см. рис 1): X = (, . . . ,) – набор вершин; V = (, . . . ,) – набор ветвей (ребер графа первичной сети); R = (, . . . ,) – набор ребер (ребер графа вторичной сети); P : V → X × X – отображение, которое определяет граф P N = (X,V), который называется первичной сетью; W : R → X × X – отображение, которое определяет граф SN = (X,R), который называется вторичной сетью; F : R → – отображение ребер в маршруты PN.

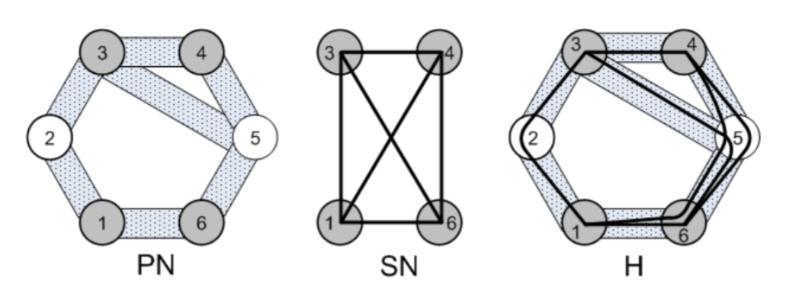


Рисунок 1 – Пример гиперграфа

## 7.2 Алгоритмы

Для демонстрации работы библиотеки, на ней будут выполнены несколько алгоритмов:

* Укладка вторичной сети в первичную.
* Нахождение маршрута общественного транспорта.

# 8 ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

## Входные данные

Класс, представляющий гиперсеть можно получить: определив ее внутри приложения, использовав метода управления гиперсетью, либо посредством считывания из текстового файла (см. рис 2).

Текстовый файл должен хранить гиперсеть в предопределенном формате, удобно читаемый человеком, сам граф может быть описан в разных представлениях: список смежности, матрица инцидентности.

## Выходные данные

Есть возможность записать гиперсеть в текстовый файл (см. рис 2). При сохранении гиперсети в файл, есть возможность выбора представления: список смежности, либо матрица инцидентности.

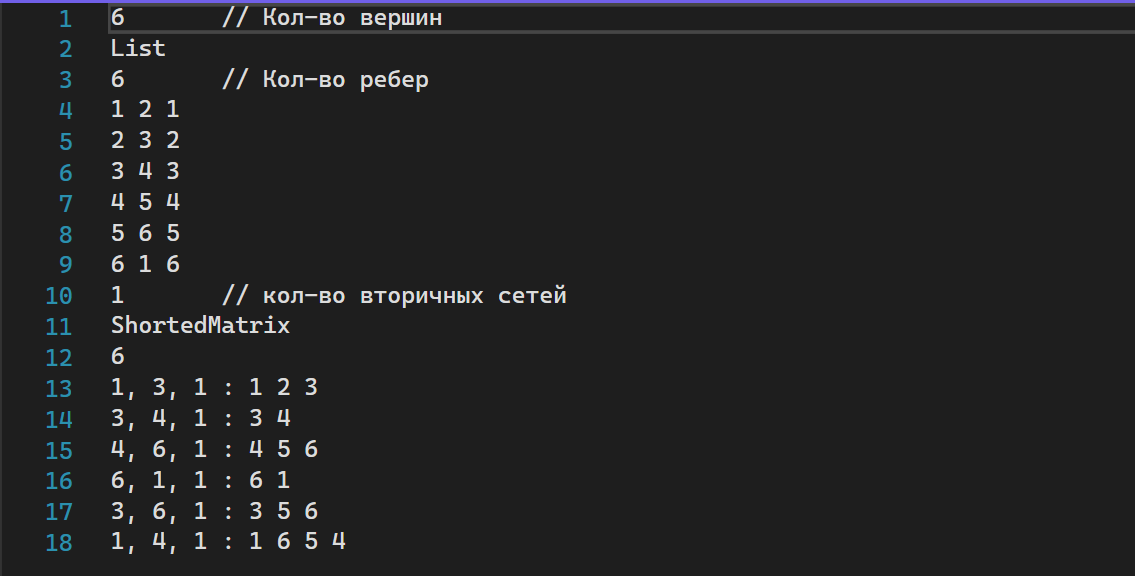


Рисунок 2 – Пример гиперсети, записанной в текстовый файл

# ОТЛАДКА И ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

Были проведены различные тесты, для проверки работоспособности приложения:

* Функциональные тесты – проверка методов на правильность их выполнения.
* Тестирование производительности – проверка библиотеки, создавая на нее значительную нагрузку.
* Smoke-тестирование – тест, проверяющий, что все базовые методы работают.

После результатов тестирования была проведена оптимизация приложения, и исправление всех найденных багов.

# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Так как результат разрабатываемого программного средства – библиотека, то для ее использования, пользователь должен загрузить ее, и добавить ссылку на проект, после этого необходимо будет подключить пространство имен библиотеки. Пример добавления библиотеки в проект указан на рисунке 3.

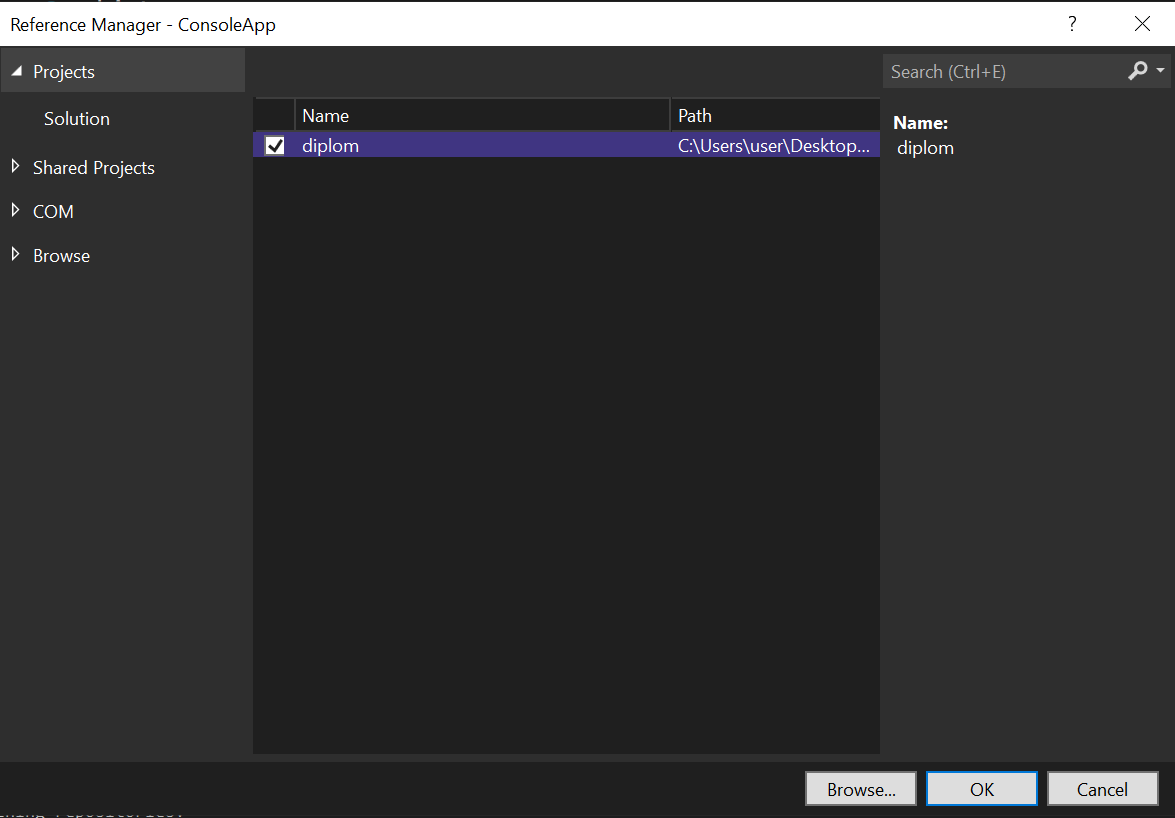


Рисунок 3 – Пример добавление библиотеки в проект

После добавление библиотеки, появится возможность пользовать ее методами. На рисунке 4 представлен пример использования гиперсети, чтения гиперсети из файла, применения алгоритма, удаления вторичной сети из первичной, и сохранения гиперсети без вторичной сети в файлы в формате матрицы инцидентности. На рисунках 4 и 5 представлена гиперсеть до и после выполнения программы.

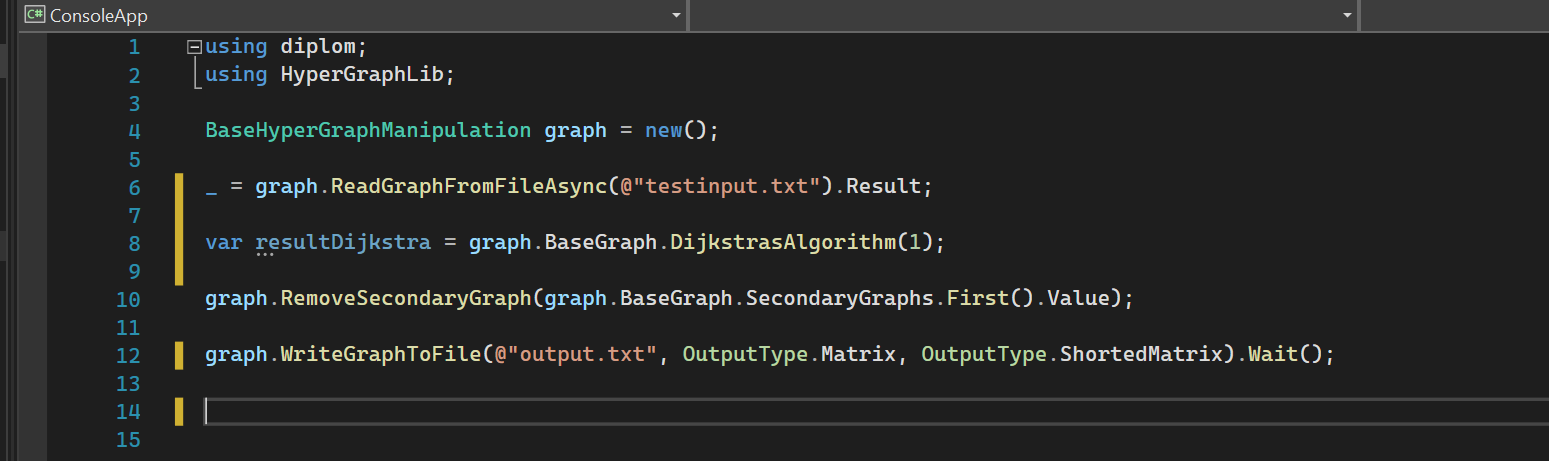


Рисунок 4 – Пример использования библиотеки

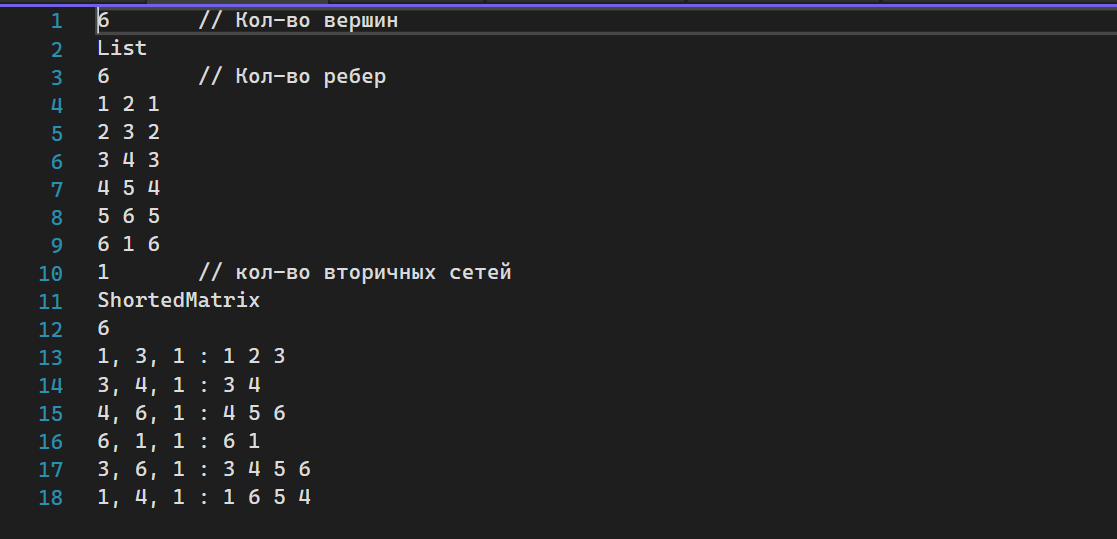


Рисунок 5 – Входной файл с гиперсетью

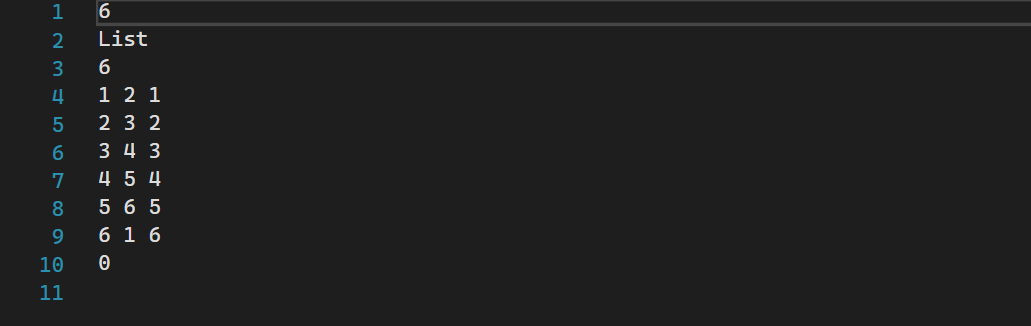


Рисунок 6 – Результат выполнения программы, гиперсеть без вторичной сети

# ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы я получил:

1. Знание предметной области гиперсетей и их математической части.
2. Опыт проектирования и тестирования приложения.
3. Опыт использования инструментов: git, visualstudio.

Результатом дипломной работы является рабочая библиотека, имеющая функционал, описанной в функциональных требованиях. Библиотека имеет все необходимые методы, с помощью которых пользователь может взаимодействовать с гиперсетью.

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Попков В. К. Математичиские модели связности. — Новосибирск : ИВМиМГ СО РАН, 2006. — 490 с.
2. Оре О. Графы и их применение. — Рипол Классик, 2002.
3. Golumbic M. C., Kaplan H., Shamir R. Graph sandwich problems // Journal of Algorithms. — 1995. — Т. 19, № 3. — С. 449—473.
4. Poulovassilis A., Levene M. A nested-graph model for the representation and manipulation of complex objects // ACM Transactions on Information Systems (TOIS). — 1994. — Т. 12, № 1. — С. 35—68.
5. Levene M., Loizou G. A graph-based data model and its ramifications // IEEE Transactions on Knowledge and Data Engineering. — 1995. — Т. 7, № 5. — С. 809—823.
6. Попков В. К. Гиперсети и структурные модели сложных систем // Математические и имитационные модели сложных систем. — Новосибирск, 1981. — С. 26—48.
7. Попков В. К. Применение теории S-гиперсетей для моделирования систем сетевой структуры // Проблемы информатики. — Новосибирск, 2010. — № 4. — С. 17—40.
8. Галямов В. А. О задаче построения структурированных кабельных систем // Новосиб. Ун-та. — Новосибирск, 2005. — С. 36.
9. Dijkstra F., Andree B., Koymans K., van der Hama J., Grosso P., de Laat C. A multi-layer network model based on ITU-T G.805 // Computer Networks, 2008. N 52. P 1927 - 1937.
10. Rodionov A., Rodionova O. Random Hypernets in Reliability Analysis of Multilayer Networks // Springer International Publishing Switzerland 2015, Computational Problems in Science and Engineering, Lecture Notes in Electrical Engineering 343, P. 307 - 313.

# ПРИЛОЖЕНИ А

public class MyCollection<TData> : Dictionary<int, TData>, IConnectionCollection<TData>

{ }

public class BaseHyperGraphManipulation : IBaseHyperGraphManipulation

{

private PrimaryGraph \_graph = new()

{

Branches = new MyCollection<Branch>(),

Nodes = new MyCollection<Node>(),

Edges = new MyCollection<Edge>(),

SecondaryGraphs = new MyCollection<SecondaryGraph>(),

};

public PrimaryGraph BaseGraph

{

get => \_graph;

set => \_graph = value;

}

public int LastId { private set; get; }

private int GetId()

{

return ++LastId;

}

public SecondaryGraph AddSecondaryGraph()

{

return AddSecondaryGraph(CreateSecondaryGraph());

}

public SecondaryGraph AddSecondaryGraph(SecondaryGraph secondaryGraph)

{

secondaryGraph.Id = GetId();

secondaryGraph.PrimaryGraph = \_graph;

\_graph.SecondaryGraphs.Add(secondaryGraph.Id, secondaryGraph);

return secondaryGraph;

}

public Branch AddBranch()

{

return AddBranch(CreateBranch());

}

public Branch AddBranch(Branch branch)

{

branch.Id = GetId();

branch.PrimaryGraph = \_graph;

\_graph.Branches.Add(branch.Id, branch);

foreach (var i in branch.Edges)

{

i.Value.Branches.Add(branch.Id, branch);

}

branch.First.Branches.Add(branch.Id, branch);

branch.Second.Branches.Add(branch.Id, branch);

branch.SecondaryGraph.Branches.Add(branch.Id, branch);

return branch;

}

public Node AddNode()

{

return AddNode(CreateNode());

}

public Node AddNode(Node node)

{

node.Id = GetId();

node.PrimaryGraph = \_graph;

\_graph.Nodes.Add(node.Id, node);

return node;

}

public Edge AddEdge(Node first, Node second)

{

return AddEdge(CreateEdge(first, second));

}

public Edge AddEdge(Edge edge)

{

edge.Id = GetId();

edge.PrimaryGraph = \_graph;

edge.First.Edges.Add(edge.Id, edge);

edge.Second.Edges.Add(edge.Id, edge);

\_graph.Edges.Add(edge.Id, edge);

return edge;

}

public Edge CreateEdge()

{

return new Edge()

{

Branches = new MyCollection<Branch>(),

PrimaryGraph = \_graph,

};

}

public Edge CreateEdge(Node first, Node second)

{

Edge edge = CreateEdge();

edge.First = first;

edge.Second = second;

return edge;

}

public Node CreateNode()

{

return new Node()

{

Edges = new MyCollection<Edge>(),

Branches = new MyCollection<Branch>(),

PrimaryGraph = \_graph,

};

}

public SecondaryGraph CreateSecondaryGraph()

{

return new SecondaryGraph()

{

Branches = new MyCollection<Branch>(),

Nodes = new MyCollection<Node>(),

};

}

public Branch CreateBranch()

{

return new Branch()

{

Edges = new MyCollection<Edge>(),

};

}

public Branch CreateBranch(SecondaryGraph secondaryGraph)

{

return new Branch()

{

Edges = new MyCollection<Edge>(),

SecondaryGraph = secondaryGraph,

PrimaryGraph = \_graph

};

}

public Branch CreateBranch(Node first, Node second)

{

return new Branch()

{

Edges = new MyCollection<Edge>(),

First = first,

Second = second,

};

}

public Branch CreateBranch(Node first, Node second, SecondaryGraph secondaryGraph)

{

return new Branch()

{

Edges = new MyCollection<Edge>(),

SecondaryGraph = secondaryGraph,

PrimaryGraph = \_graph,

First = first,

Second = second,

};

}

public void RemoveNode(Node node)

{

BaseGraph.Nodes.Remove(node.Id);

var edges = BaseGraph.Edges.Values.Where(i => i.First == node || i.Second == node);

var braches = BaseGraph.Branches.Values.Where(i => i.First == node || i.Second == node);

foreach (var i in edges)

{

BaseGraph.Edges.Remove(i.Id);

foreach (var j in i.Branches.Values)

{

BaseGraph.Branches.Remove(j.Id);

j.SecondaryGraph.Branches.Remove(j.Id);

}

}

foreach (var i in braches)

{

BaseGraph.Branches.Remove(i.Id);

}

foreach (var i in BaseGraph.SecondaryGraphs.Values)

{

i.Nodes.Remove(node.Id);

}

node.Edges = null;

node.Branches = null;

node.PrimaryGraph = null;

}

public void RemoveEdge(Edge edge)

{

BaseGraph.Edges.Remove(edge.Id);

foreach (var j in edge.Branches.Values)

{

BaseGraph.Branches.Remove(j.Id);

j.SecondaryGraph.Branches.Remove(j.Id);

}

edge.First.Edges.Remove(edge.Id);

edge.Second.Edges.Remove(edge.Id);

edge.First = null;

edge.Second = null;

edge.Branches = null;

edge.PrimaryGraph = null;

}

public void RemoveBranch(Branch branch)

{

BaseGraph.Branches.Remove(branch.Id);

branch.SecondaryGraph.Branches.Remove(branch.Id);

branch.First.Branches.Remove(branch.Id);

branch.Second.Branches.Remove(branch.Id);

foreach (var i in branch.Edges.Values)

{

i.Branches.Remove(branch.Id);

}

branch.First = null;

branch.Second = null;

branch.SecondaryGraph = null;

branch.Edges = null;

branch.PrimaryGraph = null;

}

public void RemoveSecondaryGraph(SecondaryGraph secondaryGraph)

{

BaseGraph.SecondaryGraphs.Remove(secondaryGraph.Id);

foreach (var i in secondaryGraph.Branches.Values)

{

RemoveBranch(i);

}

secondaryGraph.Nodes = null;

secondaryGraph.PrimaryGraph = null;

}

}

public static class GraphReader

{

private static int currentLine = 0;

private static Edge ItemsToEdge(Node firstItem, Node secondItem, long weight, IBaseHyperGraphManipulation graph)

{

Edge edge = graph.CreateEdge(firstItem, secondItem);

edge.Weight = weight;

//graph.AddEdge(edge);

return edge;

}

private static Branch ItemsToBranch(Node firstItem, Node secondItem,SecondaryGraph secondaryGraph, long weight, IBaseHyperGraphManipulation graph)

{

Branch branch = graph.CreateBranch(firstItem, secondItem, secondaryGraph);

branch.Weight = weight;

//graph.AddBranch(branch);

return branch;

}

public static async Task<PrimaryGraph> ReadGraphFromFileAsync(this IBaseHyperGraphManipulation graph, string filepath)

{

currentLine = 0;

//var graph = new PrimaryGraph();

using StreamReader stream = new(filepath);

string input;

int size;

List<SecondaryGraph> secondaryGraphs = new();

List<Node> nodes = new();

List<Edge> edges = null;

List<Branch> branches = new();

// чтение кол-во вершин

input = await ReadFormatedLineAsync(stream);

if (int.TryParse(input, out size) == false)

{

ThrowException("Unexpected number");

}

for (int i = 0; i < size; i++)

{

nodes.Add(graph.CreateNode());

graph.AddNode(nodes[i]);

}

// чтение типа входных данных (список смежности или матрица инцедентности)

input = await ReadFormatedLineAsync(stream);

switch (input)

{

case "list":

{

edges = await ReadEdgesList(stream, graph, nodes);

break;

}

case "matrix":

{

edges = await ReadEdgeMatrix(stream, size, graph, nodes);

break;

}

default:

ThrowException("Wrong input type, expected 'list' or 'matrix'");

break;

}

edges.ForEach(i => graph.AddEdge(i));

// чтение кол-во подграфов

input = await ReadFormatedLineAsync(stream);

if (int.TryParse(input, out size) == false)

{

ThrowException("Unexpected number");

}

int subgraphCount = size;

for (int i = 0; i < size; i++)

{

var result = await ReadSecondaryGraph(stream, subgraphCount, graph, nodes);

graph.AddSecondaryGraph(result);

}

return graph.BaseGraph;

}

private static async Task<List<Edge>> ReadEdgeMatrix(StreamReader stream, int length,

IBaseHyperGraphManipulation graph,

List<Node> nodes)

{

string input;

List<Edge> result = new();

for (int i = 0; i < length; i++)

{

input = await ReadFormatedLineAsync(stream);

var inputArray = SplitStringToLongArray(input);

if (inputArray.Length != length)

{

ThrowException("Matrix array must have length " + length + ", insted of " + inputArray.Length);

}

for (int j = 0; j < length; j++)

{

if (inputArray[j] > 0)

{

result.Add(ItemsToEdge(nodes[i], nodes[j], inputArray[j], graph));

}

}

}

return result;

}

private static async Task<List<Edge>> ReadEdgesList(StreamReader stream,

IBaseHyperGraphManipulation graph,

List<Node> nodes)

{

string input;

List<Edge> result = new();

input = await ReadFormatedLineAsync(stream);

// Чтение длины списка смежности

if (int.TryParse(input, out var length) == false)

{

ThrowException("Number expected");

}

// Чтение списка смежности в формате: "{первая вершина} {вторая вершина} {вес ребра}"

for (int i = 0; i < length; i++)

{

input = await ReadFormatedLineAsync(stream);

var array = SplitStringToIntArray(input);

if (array.Length != 3)

{

ThrowException("List must have 2 or 3 digits");

}

if (!(array[0] > 0 || array[0] < nodes.Count ||

array[1] > 0 || array[1] < nodes.Count))

{

ThrowException("Node id must be between 0 or " + nodes.Count);

}

long weight = array[2];

int firstNode = array[0] - 1;

int secondNode = array[1] - 1;

result.Add(ItemsToEdge(nodes[firstNode], nodes[secondNode], weight, graph));

}

return result;

}

private static async Task<SecondaryGraph> ReadSecondaryGraph(StreamReader stream,

int length,

IBaseHyperGraphManipulation graph,

List<Node> nodes)

{

SecondaryGraph result = graph.CreateSecondaryGraph();

string input;

int size;

List<Branch> branches = null;

// Чтение типа ввода ветвей (список/матрица)

input = await ReadFormatedLineAsync(stream);

switch (input)

{

case "list":

{

branches = await ReadBranchesList(stream, graph, nodes, result);

break;

}

case "shortedmatrix":

{

branches = await ReadBranchesMatrix(stream, graph, nodes, result);

break;

}

default:

ThrowException("Wrong input type, expected 'list' or 'shortedmatrix'");

break;

}

branches.ForEach(i => graph.AddBranch(i));

return result;

}

private static async Task<List<Branch>> ReadBranchesList(StreamReader stream,

IBaseHyperGraphManipulation graph,

List<Node> nodes,

SecondaryGraph secondaryGraph)

{

string input;

input = await ReadFormatedLineAsync(stream);

// Чтение количества ветвей

if (int.TryParse(input, out var length) == false)

{

ThrowException("Unexpected number");

}

List<Branch> result = new();

// Чтение ветвей в формате: "{первая вершина} {вторая вершина} {вес ветви}"

for (int i = 0; i < length; i++)

{

input = await ReadFormatedLineAsync(stream);

var array = SplitStringToIntArray(input);

if (array.Length != 3)

{

ThrowException("List must have 2 or 3 digits");

}

if (!(array[0] > 0 || array[0] < nodes.Count ||

array[1] > 0 || array[1] < nodes.Count))

{

ThrowException("Node id must be between 0 or " + nodes.Count);

}

long weight = array[2];

int firstNode = array[0] - 1;

int secondNode = array[1] - 1;

result.Add(ItemsToBranch(nodes[firstNode], nodes[secondNode], secondaryGraph, weight, graph));

}

// Чтение длины списка смежности

input = await ReadFormatedLineAsync(stream);

if (int.TryParse(input, out length) == false)

{

ThrowException("Unexpected number");

}

// Формат чтения ребер ветви: "{номер ветви} {первая вершина} {вторая вершина}"

for (int i = 0; i < length; i++)

{

input = await ReadFormatedLineAsync(stream);

var array = SplitStringToIntArray(input);

if (array.Length != 3)

{

ThrowException("List must have 2 or 3 digits");

}

if (!(array[0] > 0 || array[0] <= result.Count ||

array[1] > 0 || array[1] <= nodes.Count ||

array[2] > 0 || array[2] <= nodes.Count))

{

ThrowException("Node id must be between 0 or " + nodes.Count);

}

var firstNode = nodes[array[1] - 1];

var secondNode = nodes[array[2] - 1];

var edge = firstNode.Edges

.Where(i => i.Value.First == secondNode || i.Value.Second == secondNode)

.Select(i => i.Value)

.FirstOrDefault();

if (edge == null)

{

ThrowException($"Edge between nodes {array[1]} and {array[2]} not found");

}

var branchNumber = array[0] - 1;

result[branchNumber].Edges.Add(edge.Id, edge);

secondaryGraph.Nodes.TryAdd(firstNode.Id, firstNode);

secondaryGraph.Nodes.TryAdd(secondNode.Id, secondNode);

}

return result;

}

private static async Task<List<Branch>> ReadBranchesMatrix(StreamReader stream,

IBaseHyperGraphManipulation graph,

List<Node> nodes,

SecondaryGraph secondaryGraph)

{

string input;

// Чтение количества ветвей

input = await ReadFormatedLineAsync(stream);

if (int.TryParse(input, out var length) == false)

{

ThrowException("Unexpected number");

}

List<Branch> result = new();

// Чтение ветвей в формате: " {первая вершина} {вторая вершина} {вес} : {Перечисление ребер ... (1, 2, 3, 4, 5)} "

// Есть краткая форма, состоящая из одного ребра " {первая вершина} {вторая вершина} {вес} "

for (int i = 0; i < length; i++)

{

input = await ReadFormatedLineAsync(stream);

var inputArray = input.Split(':', StringSplitOptions.RemoveEmptyEntries);

if (inputArray.Length <= 0 || inputArray.Length > 2)

{

ThrowException("Wrong matrix format");

}

// чтение и добавление ветви

var longArray = SplitStringToLongArray(inputArray[0]);

int firstNode = (int)longArray[0] - 1;

int secondNode = (int)longArray[1] - 1;

long weight = longArray[2];

result.Add(ItemsToBranch(nodes[firstNode], nodes[secondNode], secondaryGraph, weight, graph));

if (inputArray.Length == 1)

{

var edge = nodes[firstNode].Edges

.Where(i => i.Value.First == nodes[firstNode] && i.Value.Second == nodes[secondNode]

|| i.Value.Second == nodes[firstNode] && i.Value.First == nodes[secondNode])

.Select(i => i.Value)

.FirstOrDefault();

var AddedBranch = result[result.Count - 1];

AddedBranch.Edges.Add(edge.Id, edge);

continue;

}

// чтение пути ветви

var intArray = SplitStringToIntArray(inputArray[1]);

if (intArray.Length < 2)

{

ThrowException("Branch path too small");

}

for (int j = 0; j < intArray.Length - 1; j++)

{

int firstEdgeNode = intArray[j] - 1;

int secondEdgeNode = intArray[j + 1] - 1;

var edge = nodes[firstEdgeNode].Edges

.Where(i => i.Value.First == nodes[firstEdgeNode] && i.Value.Second == nodes[secondEdgeNode]

|| i.Value.Second == nodes[firstEdgeNode] && i.Value.First == nodes[secondEdgeNode])

.Select(i => i.Value)

.FirstOrDefault();

if (edge == null)

{

ThrowException($"Edge between nodes {firstEdgeNode + 1} and {secondEdgeNode + 1} not found");

}

var AddedBranch = result[result.Count - 1];

try

{

AddedBranch.Edges.Add(edge.Id, edge);

} catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

secondaryGraph.Nodes.TryAdd(secondEdgeNode, edge.First.GetOtherNodeFromEdge(edge));

}

secondaryGraph.Nodes.TryAdd(intArray[0] - 1, nodes[intArray[0] - 1]);

secondaryGraph.Nodes.TryAdd(firstNode, nodes[firstNode]);

secondaryGraph.Nodes.TryAdd(secondNode, nodes[secondNode]);

}

return result;

}

private static int[] SplitStringToIntArray(string str)

{

int[] result;

var splitedInput = str.Split(new char[] { ',', ' ', '.' },

StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);

result = Array.ConvertAll(splitedInput, s => {

if (int.TryParse(s, out var result) == false)

{

ThrowException("Except number");

}

return result;

});

return result;

}

private static long[] SplitStringToLongArray(string str)

{

long[] result;

//var splitedInput = str.Split(new char[] { ',', ' ', '.', });

var splitedInput = str.Split(new char[] { ',', ' ', '.' },

StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);

result = Array.ConvertAll(splitedInput, s => {

if (long.TryParse(s, out var result) == false)

{

ThrowException("Except number");

}

return result;

});

return result;

}

private static async Task<string> ReadFormatedLineAsync(StreamReader stream)

{

string input;

do

{

currentLine += 1;

input = await stream.ReadLineAsync();

int commentIndex = input.IndexOf("//");

if (commentIndex != -1)

input = input.Substring(0, commentIndex);

} while (string.IsNullOrWhiteSpace(input));

if (input == null)

{

ThrowException("Not enought data in file");

}

return (input).Trim().ToLower();

}

private static Exception ThrowException(string str)

{

throw new Exception(str + "\nLine: " + currentLine);

}

}

public enum OutputType

{

List, // Список смежности

Matrix, // Матрица инцидентности

ShortedMatrix

}

public static class GraphWriter

{

public static async Task WriteGraphToFile(this IBaseHyperGraphManipulation graph,

string filepath,

OutputType edgeOutputType,

OutputType branchOutputType)

{

using StreamWriter stream = new StreamWriter(filepath);

string input;

// пишется кол-во вершин

stream.WriteLine(graph.BaseGraph.Nodes.Count);

// пишется тип вывода

stream.WriteLine(edgeOutputType.ToString());

// выводится граф в виде OutputType

switch (edgeOutputType)

{

case OutputType.List:

{

var list = graph.BaseGraph.AsList();

stream.WriteLine(list.Count);

foreach (var item in list)

{

stream.WriteLine(item.FirstNode.ToString() + ' '

+ item.SecondNode.ToString() + ' '

+ item.Weight.ToString());

}

break;

}

case OutputType.Matrix:

{

var matrix = graph.BaseGraph.AsMatrix();

StringBuilder output = new(matrix.size\*3);

foreach (var i in matrix.matrix)

{

for (int j = 0; j < matrix.size; j++)

{

if (j == matrix.size - 1)

output.Append(i[j]);

else

output.Append(i[j] + " ");

}

stream.WriteLine(output);

output.Clear();

}

break;

}

default:

throw new Exception("Unknown type of edge representation");

}

// Кол-во вторичных сетей

stream.WriteLine(graph.BaseGraph.SecondaryGraphs.Count);

foreach (var secondaryGraph in graph.BaseGraph.SecondaryGraphs.Values)

{

// Тип вывода ветвей

stream.WriteLine(branchOutputType.ToString());

// Количества ветвей

stream.WriteLine(secondaryGraph.Branches.Count);

foreach (var branch in secondaryGraph.Branches.Values)

{

switch (branchOutputType)

{

case OutputType.ShortedMatrix:

StringBuilder output = new(10 + branch.Edges.Count \* 3);

output.Append($"{branch.First.Id} {branch.Second.Id} {branch.Weight}: ");

var node = branch.First;

Edge pastEdge = null;

do

{

output.Append($"{node.Id} ");

pastEdge = branch.Edges

.Where(i => (i.Value.First == node || i.Value.Second == node)

&& i.Value != pastEdge)

.FirstOrDefault().Value;

if (pastEdge == null)

{

break;

throw new Exception("TODO: NAME ERROR");

}

node = node.GetOtherNodeFromEdge(pastEdge);

} while (true);

//output.Append(branch.Second.Id);

stream.WriteLine(output);

output.Clear();

break;

default:

throw new Exception("Unknown type of branch representation");

}

}

}

}

}

public interface IConnectionCollection<TData> : IDictionary<int, TData>

{ }

public interface IConnectionFactory

{

IConnectionCollection<TData> CreateConnectionCollection<TData>();

}

public class PrimaryGraph

{

public IConnectionCollection<Node> Nodes { get; set; }

public IConnectionCollection<Edge> Edges { get; set; }

public IConnectionCollection<SecondaryGraph> SecondaryGraphs { get; set; }

public IConnectionCollection<Branch> Branches { get; set; }

public int NodeCount { get => Nodes.Count; }

}

public class SecondaryGraph

{

public int Id { get; set; }

public IConnectionCollection<Branch> Branches { get; set; }

public IConnectionCollection<Node> Nodes { get; set; }

public PrimaryGraph PrimaryGraph { get; set; }

}

public class Node

{

public int Id { get; set; }

public string Name { get; set; }

public IConnectionCollection<Edge> Edges { get; set; }

public IConnectionCollection<Branch> Branches { get; set; }

public PrimaryGraph PrimaryGraph { get; set; }

}

public class Edge

{

public int Id { get; set; }

public long Weight { get; set; }

public Node First { get; set; }

public Node Second { get; set; }

public IConnectionCollection<Branch> Branches { get; set; }

public PrimaryGraph PrimaryGraph { get; set; }

public Edge() { }

public Edge(Node first, Node second)

{

First = first;

Second = second;

}

public string DebugEdgeInfo { get => First.Id + ":" + Second.Id; }

}

public class Branch

{

public int Id { get; set; }

public long Weight { get; set; }

public Node First { get; set; }

public Node Second { get; set; }

public IConnectionCollection<Edge> Edges { get; set; }

public SecondaryGraph SecondaryGraph { get; set; }

public PrimaryGraph PrimaryGraph { get; set; }

public string DebugEdgeInfo { get => First.Id + ":" + Second.Id; }

}

}

public interface IBaseHyperGraphManipulation

{

PrimaryGraph BaseGraph { get; set; }

Node CreateNode();

Edge CreateEdge();

Edge CreateEdge(Node first, Node second);

SecondaryGraph CreateSecondaryGraph();

Branch CreateBranch();

Branch CreateBranch(Node first, Node second);

Branch CreateBranch(SecondaryGraph secondaryGraph);

Branch CreateBranch(Node first, Node second, SecondaryGraph secondaryGraph);

Node AddNode();

Node AddNode(Node node);

Edge AddEdge(Node first, Node second);

Edge AddEdge(Edge edge);

SecondaryGraph AddSecondaryGraph();

SecondaryGraph AddSecondaryGraph(SecondaryGraph secondaryGraph);

//SecondaryGraph AddSecondaryGraphWithBrances(SecondaryGraph secondaryGraph);

Branch AddBranch();

Branch AddBranch(Branch branch);

void RemoveNode(Node node);

void RemoveEdge(Edge edge);

void RemoveBranch(Branch branch);

void RemoveSecondaryGraph(SecondaryGraph secondaryGraph);

}

public static class NodeExtensions

{

public static bool IsFirstNodeInEdge(this Node node, Edge edge)

{

return node.Id == edge.First.Id;

}

public static Node GetOtherNodeFromEdge(this Node node, Edge edge)

{

return node.Id == edge.First.Id

? edge.Second : edge.First;

}

public static Node GetOtherNodeFromBranch(this Node node, Branch branch)

{

return node.Id == branch.First.Id

? branch.Second : branch.First;

}

}

public static class GraphExtension

{

public static (long[][] matrix, int size) AsMatrix(this PrimaryGraph graph)

{

int size = graph.Nodes.Count;

long[][] result = new long[size][];

int index = -1;

var nodeIndexes = graph.Nodes

.OrderBy(i => i.Key)

.ToDictionary(i => i.Key, i => ++index);

var nodes = graph.Nodes

.OrderBy(i => i.Key)

.ToArray();

for (int i = 0; i < size; i++)

{

var node = nodes[i].Value;

result[i] = new long[size];

for (int j = 0; j < size; j++)

{

result[i][j] = 0;

}

foreach (var edge in node.Edges)

{

var anotherNode = node.GetOtherNodeFromEdge(edge.Value);

result[i][nodeIndexes[anotherNode.Id]] = edge.Value.Weight;

}

}

return (result, size);

}

public static List<(int FirstNode, int SecondNode, long Weight)> AsList(this PrimaryGraph graph)

{

List<(int, int, long)> result = new(graph.Edges.Count);

var edges = graph.Edges.OrderBy(i => i.Key);

foreach (var edge in edges)

{

result.Add((

edge.Value.First.Id,

edge.Value.Second.Id,

edge.Value.Weight));

}

return result;

}

public static IDictionary<int, (long Value, bool IsVisited)> DijkstrasAlgorithm(this PrimaryGraph graph,

int FirstNode)

{

Dictionary<int, (long Value, bool IsVisited)> nodeValue = new();

foreach (var node in graph.Nodes)

{

nodeValue.Add(node.Key, (-1, false));

}

Node findMin()

{

Node minNode = null;

long minValue = -1;

foreach(var nodes in nodeValue)

{

if (!nodes.Value.IsVisited

&& (nodes.Value.Value < minValue || minValue == -1)

&& nodes.Value.Value != -1)

{

minValue = nodes.Value.Value;

minNode = graph.Nodes[nodes.Key];

}

}

return minNode;

}

Node currentNode = graph.Nodes

.Where(i => i.Value.Id == FirstNode)

.First()

.Value;

while (true)

{

var currentNodeValue = nodeValue[currentNode.Id];

if (nodeValue[currentNode.Id].IsVisited)

continue;

(Node firstNode, Node secondNode, long Weight)[] edgesWithBranches =

currentNode.Edges

.Select(i => (i.Value.First, i.Value.Second, i.Value.Weight))

.Concat(currentNode.Branches

.Select(i => (i.Value.First, i.Value.Second, i.Value.Weight)))

.OrderBy(i => i.Weight)

.ToArray();

foreach (var i in edgesWithBranches)

{

var otherNode = i.firstNode == currentNode ? i.secondNode : i.firstNode;

var otherNodeValue = nodeValue[otherNode.Id];

if (otherNodeValue.IsVisited)

continue;

if (otherNodeValue.Value == -1)

{

otherNodeValue.Value = currentNodeValue.Value + i.Weight;

}

else

{

if (currentNodeValue.Value + i.Weight < otherNodeValue.Value)

{

otherNodeValue.Value = currentNodeValue.Value + i.Weight;

}

}

nodeValue[otherNode.Id] = otherNodeValue;

}

currentNodeValue.IsVisited = true;

nodeValue[currentNode.Id] = currentNodeValue;

currentNode = findMin();

if (currentNode == null)

{

break;

}

}

return nodeValue;

}

}