Исполнители отчёта:

д.т.н. Родионов А.С. – руководитель НИР

к.т.н. Токтошов Г.Ы.

к.т.н. Мигов Д.А.

Ткачёв К.В.

Кальней А.М.

Трофимова Л.В.

Оглавление

[**Результаты выполнения этапа** 4](#_Toc16089625)

[**1.** **Разработка алгоритма случайного поиска выбора точек подключения к глобальным энергосистемам из заданного множества узлов** 4](#_Toc16089626)

[1.1. Математическая модель 4](#_Toc16089627)

[1.2. Метод факторизации 5](#_Toc16089628)

[**2.** **Разработка работающей автономной программы оптимизации сетей питания базовых станций, включая исходный код программ с комментариями** 8](#_Toc16089629)

[2.1. Общая схема реализованного алгоритма 8](#_Toc16089630)

[2.2. Программная реализация 9](#_Toc16089631)

[**3.** **Постановка задачи минимизации стоимости сети питания** 13](#_Toc16089632)

[Приложение 1 16](#_Toc16089633)

**Содержание этапа:** Оптимизация структуры сети питания базовых станций.

**Планируемые результаты:**

1. Разработка алгоритма выбора точек подключения к глобальным энергосистемам из заданного множества узлов, основанного на методе случайного поиска;
2. Разработка работающей автономной программы оптимизации сетей питания базовых станций, включая исходный код программ с комментариями.

# **Результаты выполнения этапа**

## **Разработка алгоритма случайного поиска выбора точек подключения к глобальным энергосистемам из заданного множества узлов**

## Математическая модель

При разработке алгоритма выбора точек подключения к глобальным энергосистемам из заданного множества узлов, использовалось представление структуры сети гиперсетью, которое допускает рассматривать схемы шахтных выработок и сети базовых станций как единый объект, а в качестве показателя, характеризующего эффективности функционирования сетей питания базовых станций использовалась надежность гиперсети.

Гиперсеть представляет из себя математический объект, которая определяется следующим образом:

Гиперсеть  включает следующие объекты:

 − множество точек пересечений шахтных выработок;

 − множество линейной части шахтных выработок (ветви);

 - множество точек пересечения электрокабелей (базовые станции);

− множество электрокабелей, прокладываемых по выработкам (ребра).

Тогда:

* - отображение определяющего, графа структуру шахтных выработок, т.е. сопоставляет каждому элементу  множество вершин . Тем самым, математически такая структура будет представлена графом вида , который в научных литературах имеет название граф первичной сети (Primary Network –PN).
*  − отображение, сопоставляющее каждому элементу  множество вершин . Тем самым отображение определяет графа , для структуры сети базовых станций, и в научных литературах имеет название граф вторичной сети (Secondary Network –SN).
*  – отображение ребер  в маршруты графа  . Множество всех маршрутов , отображающее каждому ребру  графа  единственный маршрут в графе , назовем вложением графа в . Таким образом, отображение  определяет гиперсеть, обозначаемую как . Пусть
*  - число вершин (количество базовых станций);
*  - число ветвей (множество линейной части шахтных выработок);
*  - число ребер (множество электрокабелей);
*  - вероятность того, что -я ветви исправна (нет разрушений, аварий и т.п.);
*  - вероятность того, что -я ветви неисправна (разрушена, произошла авария и т.п.);
*  - вероятность того, что гиперсеть  имеет связными  выделенных вершин.

Отметим, что самыми ненадежными элементами гиперсети являются ветви первичной сети, они присутствую с вероятностью . Возможны различные варианты разрушения гиперсети. Например, для гиперсети, с двумя выделенными вершинами *s* и *t* (см.рис.1.) возможны следующие виды разрушения:

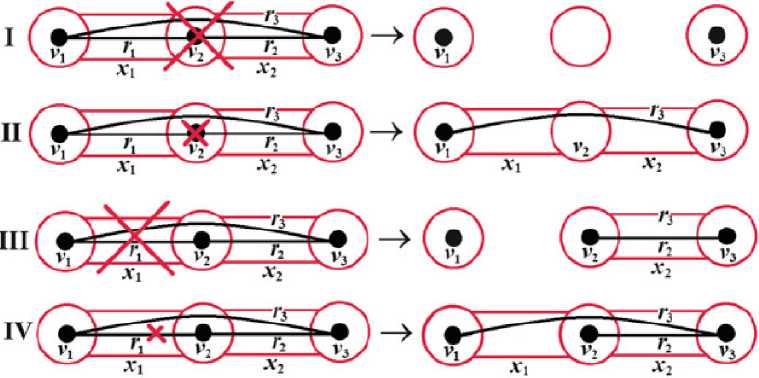


Рис.1. Возможные случаи разрушения

В данном отчете используются варианты I, III и IV, соответственно, для удаления вершин, ветвей и рёбер гиперсети. Для расчета надежность функционирования гиперсети используется метод факторизации.

## Метод факторизации

Используем следующее представление полинома надёжности:

 (1)

где  - имеет смысл общего количество гиперсетей, имеющих связными две целевые вершины, которые можно получить по всем вариантам разрушения гиперсети удалением  ветвей. Коэффициенты полинома надёжности в его классическом представлении:

 (2)

связаны с коэффициентами полинома (1) следующими соотношениями



В дальнейшем в обозначениях будем опускать вероятность, а также имя гиперсети, если оно понятно из контекста.

Получение коэффициентов надежности в виде (1) как лучше нельзя соответствует методу факторизации или Мура-Шеннона (см.рис.2), используем в качестве основного:

 (3)

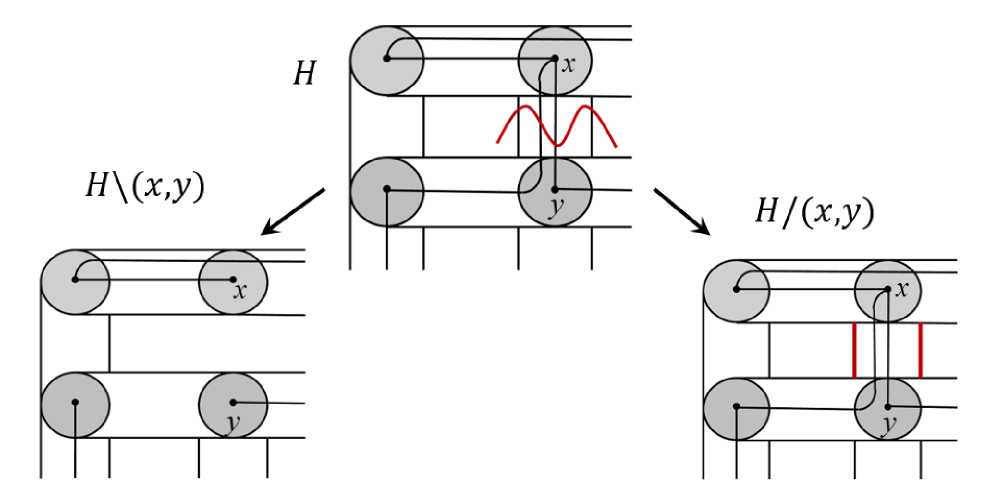


Рис. 2. Метод факторизации (Мура Шеннона)

где – произвольная ветви гиперсети , - вероятность присутствия ветви ,  – гиперсеть, где ветви является «онадёженной» (т.е. вероятность присутствия ветви становится равна единице),  – гиперсеть получаемая из  удалением ветви .

Заметим, что при удалении ветви может быть разрушено более одного ребра, либо ни одного. Данный метод заключается в рекурсивном разбиении гиперсети по ветви на несколько более простых, соответственно, где ветвь «онадёжена» и где удалена. Рекурсия продолжается до получения надёжного пути, соединяющего выделенные вершины, либо до получения несвязанной вторичной сети, также рекурсия оканчивается при получении гиперсети с двумя вершинами.

Также стоит упомянуть, что при получении полинома, складывающегося из несколько независимо получаемых полиномов, необходимо их приведение к одинаковой степени. Для этого полиномы степени, меньшей максимальной, домножались на специальный полином того же вида и тождественно равный единице (степень определялась как разница степеней максимальной и текущего полинома):

 (4)

## **Разработка работающей автономной программы оптимизации сетей питания базовых станций, включая исходный код программ с комментариями**

На основе предложенного метода выбора точек подключения к глобальным энергосистемам из заданного множества узлов был разработан алгоритм, основанный на методе факторизации и случайного поиска. Данный метод позволяет строить случайную гиперсеть с заданной надежностью.

## Общая схема реализованного алгоритма

Алгоритм генерации случайной гиперсети включает следующие этапы:

1. Генерация первого случайного дерева (randomTree1) вторичной сети.
2. Укладка дерева в заданную первичную сеть, нахождением случайного пути ребер первичной сети, реализующего ветвь randomTree1.
3. Генерация второго случайного дерева (randomTree2) вторичной сети, не включающего корневую вершину randomTree1.
4. Укладка второго дерева в ту же первичную сеть, нахождением случайного пути ребер первичной сети, реализующего ветвь randomTree2.

Генерируемые случайные гиперсети предназначены исключительно для тестирования производительности предлагаемых алгоритмов расчёта надёжности гиперсетей.

Общая схема алгоритма расчёта надёжности выглядит следующим образом:

1) Проверка возможности редукции мостов и возможности непосредственного вычисления надежности гиперсети (на данный момент непосредственно вычисляется 2-х вершинная гиперсеть). Если редукция мостов возвращает несвязную гиперсеть или имеется непосредственное вычисление, то выходим из рекурсии, иначе на следующий шаг.

2) Проверка возможности редукции рёбер и редукции цепей. Важен порядок (сначала редукция рёбер, после цепей), так как при обработке «простых» цепей случай висячих вершин во вторичной сети не рассматривается. После каждой редукции выполняются действия из шага 1.

3) Выбор разрешающего элемента (выбирается наиболее «насыщенная» ветвь, т.е. имеющая наибольшее количество маршрутов, которые проходят через неё) и провести факторизацию, т. е. подготовить пару гиперсетей и выполнить рекурсивные вызовы функции. Перед вызовом производятся проверки: для гиперсети с «онадёженой» ветвью – проверка на наличие надёжного пути между выделенными вершинами во вторичной сети и для гиперсети с удалённой ветвью – проверка на связность. Таким образом получаем четыре варианта применения метода факторизации (один вариант – выход из рекурсии, два варианта – один вызов рекурсивной функции, один вариант – два вызова рекурсивной функции, как показано в формуле (3)).

Предложенные алгоритмы программно реализованы.

## Программная реализация

Программа создавалась на IDE JetBrain CLion 2017.3.4 на ПК следующей конфигурации: ASROCK B450M-HDV R4.0, G.Skill Ripjaws V 16GB DDR4, AMD Ryzen 5 3600 3.6 ГГц и 4.2 ГГц 6 ядер 12 потоков, windows 10 pro 64 bit.

Код программы включает следующие модули представленные в таблице 1.

Таблица 1. Основные модули программы

|  |  |
| --- | --- |
| void GetData() | Получение входных данных из файла |
| void ComputeBinomialCoefficients() | Вычисление биномиальных коэффициентов |
| void NormalizeSolution() | Метод изменения размера ветви |
| void Send() | Отправления объекта ядру для расчёта |
| T Recv() | Получение объекта ядру для расчёта |
| void SendControl() | Управление исполнением программы используя MPI |
| void ComputeMENC() | Вычисление MENC |
| void ComputeAPC() | Вычисление APC |
| void ErrorHandler() | Обработчик ошибок |
| void BcastDataByMaster() | Отправка дынных от процесса мастера исполнителям |
| Branch GetSolution() | Получение решения |
| void Master() | Инициализация работы процесса мастера |
| void BcastDataBySlaves() | Получение дынных от процесса мастера |
| void Slaves() | Инициализация работы процессов исполнителей |
| Branch GetAllowingBranch() | Получение разрешай ветви с максимальной насыщенностью |
| Branch SimpleCase () | Явный расчёт гиперсети |
| Branch PairConnectivity() | Расчёт парной связности гиперсети |
| GetRandomNetwork() | Получение случайной сети |
| GetRandomTree() | Получение случайного дерева |
| void Mapping() | Укладка ребра в первичную сеть |
| void SetMapping() | Получение укладки ребра в первичную сеть |
| H TryGetRandomHypernet() | Получение случайной гиперсети необходимой для работы оптимизации (возможен timeout) |
| H GetRandomHypernet() | Получение случайной гиперсети |
| H GetRandomHypernet() | Получение случайной гиперсети необходимой для работы оптимизации |

Инициализация программы

Инициализация MPI

Расчёт вариантов гиперсети

Вычисление коэффициентов

Процесс мастер

Входные данные

Процесс исполнитель

Получение деревьев

Построение гиперсети, укладка деревьев

Проверка условий остановки

Вывод данных

Рис. 3. Схема функционирования программы

Исходный код основных модулей приведён в Приложении 1 (полный код передаётся в электронном виде).

Запуск исполняемой программы осуществляется из командной строки с указанием файла входных данных:

> mpi hypernettest.exe <dataNet.dat

Название файла с данными условное, он имеет следующую структуру (пример файла с исходными данными даётся в электронном виде):

TestHypernet(56, 78, 0, 5) //начальные данные работы алгоритма 78 - количество вершин в графе, 111 - количество ветвей (первичная сеть), 0 - количество рёбер для генерации методом GetRandomHypernet (вторичная сеть), 5 - количество вершин для оптимизации

1 //номер ветви

1 2 //вершины ветви

0 //список номеров рёбер в ветви (0 - разделитель)

…

1 14 28 42 56 // вершины для оптимизации

Альтернативный путь запуска заключается в установке дополнительного программного обеспечения в виде JetBrain CLion либо Visual Studio, сборке и запуске проекта.

Тестирование программы проведено методом проверки классов эквивалентности входных данных.

Проведённое тестирование программы показало:

1. Алгоритмы корректны.

2. Распараллеливание программы с применением MPI даёт существенное ускорение расчётов.

3. Выявлены узкие места:

a. Генератор случайных гиперсетей требует улучшения в части временных затрат, которые в основном связаны с неэффективным процессом укладки вторичных сетей в первичную.

b. Множество конечных вариантов, позволяющих точный расчёт, требует расширения, это приведёт к существенному сокращению количества рекурсивных вызовов базовой процедуры.

4. Требуется проведение теоретических исследований в части использования методов декомпозиции больших гиперсетей, возможно с переходом на приближённые расчёты.

Таблица 2. Результаты тестирования TestHypernet(56, 78, 0, 5) для вершин 1, 14, 28, 42, 56 с двумя вкладываемыми деревьями по 28 вершин.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| FirstRoot | SecondRoot | TreeNode Intersections | UnconnectedTreeNodes | Value p=0,9 | Value p=0,99 | Time |
| 1 | 14 | 15 | 15 | 0.0407232977 | 0.4157836441 | 1346.8140091 |
| 1 | 28 | 13 | 13 | 0.0310345565 | 0.4338464790 | 2798.2923923 |
| 1 | 42 | 14 | 14 | 0.0479243250 | 0.4334229575 | 1454.7369369 |
| 1 | 56 | 16 | 16 | 0.0231686730 | 0.3813093623 | 5966.4090239 |
| 14 | 28 | 13 | 13 | 0.0387163247 | 0.4399992764 | 2697.0939706 |
| 14 | 42 | 12 | 12 | 0.0210452860 | 0.4342569015 | 1175.7869887 |
| 14 | 56 | 17 | 17 | 0.0579587931 | 0.3862364054 | 3851.7349142 |
| 28 | 42 | 15 | 15 | 0.0371822396 | 0.4181287775 | 4550.2585846 |
| 28 | 56 | 17 | 17 | 0.0423185781 | 0.3762632220 | 3475.1335170 |
| 42 | 56 | 16 | 16 | 0.0216806601 | 0.3693568435 | 1907.7624152 |
| 1 | 14 | 15 | 15 | 0.0407232977 | 0.4157836435 | 1346.8140094 |

Анализ результатов тестирования показывает, что:

1. Уровень степени перекрытия вторичных сетей существенно повышает надёжность (двухфидерная схема очевидно предпочтительнее однофидерной по критерию надёжности).
2. Увеличение надёжности ребра на 9 сотых приводит, в среднем, к повышению надёжности сети минимум на десятичный порядок.

## **Постановка задачи минимизации стоимости сети питания**

Рассмотрим задачу размещения контроллеров для обеспечения электроснабжения заданного набора базовых станций с минимальными затратами.

Предполагается, что в шахте уже размещены базовые станции (БС), и имеются места для возможного размещения контроллеров – так называемые АОШы. В качестве входных данных предлагаемый алгоритм использует ряд параметров, поясняемых ниже, и матрицу расстояний [*Dist*] размера *N×A*, где *N* – количество БС, *A* – количество АОШей, а *Dist(i,j)* – расстояние от БС с номером *i* до АОШ с номером *j*. В данном случае предполагается, что стоимость подключения пропорциональна длине прокладываемого кабеля, т.е. расстоянию. В противном случае вместо матрицы расстояний может быть использована матрица стоимости подключения БС с номером *i* до АОШ с номером *j*, без каких-либо изменений в предлагаемом ниже алгоритме.

В базовом варианте постановки задано количество размещаемых контроллеров – *S*. В общем случае это число неизвестно, и заданы две верхних грани: *S –* минимальное количество размещаемых контроллеров и *S* – максимальное количество размещаемых контроллеров. Таким образом, определение оптимального значения параметра *S* из интервала *S* ≤ *S* ≤  *S,* является отдельной подзадачей. В самом общем случае можно положить *S*=1, *S*=Σ1A*Cap*(*j*), где *Cap*(*j*) – количество контроллеров, которые могут быть размещены в *j-*м АОШе. Через *Ai*будем обозначать номер АОШа, в котором располагается контроллер, к которому подключена БС с номером *i*, а через *Aj* будем обозначать номер АОШа, в котором располагается контроллер с номером *j*. Также в качестве параметров заданы стоимости размещения контроллера в конкретном АОШе – *Cost*(*j*), и константа *ConCap –* количество БС, которые могут быть запитаны от одного контроллера (полагается по умолчанию равной двадцати).

Тогда задача может быть сформулирована как задача размещения контроллеров в заданных местах для подключения к ним БС с минимальными затратами в условиях вышеописанных ограничений. Математически задача формулируется следующим образом: разместить *S* (*S* ≤ S ≤  *S*) контроллеров в Ω позиций и приписать каждой БС *i* соответствующий контроллер *Ai* для минимизации целевой функции

с учётом ограничений на *Cap*(*j*) – количество контроллеров, которые могут быть размещены в *j-*м АОШе и *ConCap –* количество БС, которые могут быть запитаны от одного контроллера.

Точный алгоритм решения подразумевает решение двух подзадач: перебор вариантов размещения (Задача А), которых даже в базовом варианте и нахождение оптимального назначения всех БС к расставленным в Задаче А контроллерам (Задача В). Ввиду большой трудоёмкости такого подхода предлагается приближённый алгоритм. Как и в случае точного решения, Задачи А и В могут решаться независимо, т.е. в процессе решения Задачи А каким-либо способ получается вариант размещения контроллеров, для которого решается задача В, далее продолжается выполнение Задачи А, и т.д. до получения приемлемого решения общей задачи. В случае неизвестного *S* подобный процесс придётся запускать для весех *S* из интервала *S* ≤ S ≤  *S*.

Для решения Задачи А могут быть использованы различные биоинспирированные техники или какой-либо детерминированный эвристический метод. В первом случае это может быть генетический алгоритм, во втором – хорошо известный метод направленного перебора комбинаций, предложенный Лином в 60-е годы 20-го века, успешно используемый для решения различный задач дискретной оптимизации, например, для поиска р-медианы. Реализация общего алгоритма требует проведения теоретических исследований для выбора наилучшего способа кодирования решений, разработки частных алгоритмов получения допустимых решений, а также операторов скрещивания и мутации, не выводящих решения из заданного класса.

Решать Задачу В предлагается также приближённым алгоритмом:

1) Для каждого размещённого контроллера *j* находим его «окружность» Ω*j* – т.е. те БС, которые располагаются к нему ближе чем к другим контроллерам. Если |Ω*j*|> *ConCap,* то оставляем в Ω*j* ровно *ConCap* БС, остальные заносим в множество Ω, которое изначально перед выполнением Шага 1) является пустым.

2) Перебираем элементы множества Ω. Для каждого ищем ближайший контроллер *j*, такой что |Ω*j*|< *ConCap* и вносим эту БС в окрестность Ω*j.*

3) Вычисляем значение целевой функции *С* и возвращаемся к выполнению Задачи А, передавая *С* и набор Ω*j.*.

4) Если не удаётся найти такой контроллер для какой-либо БС, то решения для данной конфигурации контроллеров не существует. Присваиваем целевой функции значение бесконечности и возвращаемся к выполнению Задачи А.

На выходе алгоритма решения Задачи А (например, методом Лина) с использованием предложенного алгоритма получим значение *С* целевой функции и набор Ω*j* для каждого контроллера *j* для найденной оптимальной конфигурации контроллеров. В случае, если *S* не было изначально известно, и ещё не все допустимые значения *S* перебраны, запускаем решение Задачи А с новым *S.* Когда все возможные *S* будут перебраны, выбираем то из решений Задачи А, у которого минимальное значение целевой функции.

В перспективе возможно дополнить рассмотренную постановку требованием обеспечения заданной надёжности для проектируемой сети электроснабжения БС, т.е. рассмотреть задачу размещения контроллеров для обеспечения электроснабжения заданного набора базовых станций с заданным уровнем надёжности и с минимальными затратами.

Реализация предложенных алгоритмов выходит за рамки настоящего договора в силу существенных временных затрат и является предметом переговоров для дальнейших совместных работ.

# Приложение 1

Исходный код основных модулей программы оптимизации выбора точек подключения к внешним сетям на языке C++ (полный код передаётся заказчику в электронном виде).

#include "Globals.h"

#include "Funcs.h"

#include "DTO.h"

// инициализация глобальных переменных

std::ifstream input;

std::ofstream output;

int n = 0, m = 0, k = 0, l = 0;

int ReliableHypernets = 0, UnconnectedHypernets = 0, TwoNodesHypernets = 0, ChainsReduced = 0,

UnconnectedNodesReduced = 0, PairConnectivityCalls = 0, EdgesReduced = 0, ComplexChains = 0, HelpProcessors = 0,

TreeNodeIntersections = 0, UnconnectedTreeNodes = 0;

int FirstRoot, SecondRoot;

std::vector<Branch> Bin;

const double p = 0.9, z = 0.1;

unsigned long long int TotalBytesTransfer = 0;

int seed = time(0);

// проверка уникальности индекса в списке

template <class T>

bool IsUniqueId(const T &items, const int &id) {

int count = 0;

for(auto &item : items) {

if (item.GetId() == id) {

count++;

}

}

return count < 2;

}

// получение данных из файла

void GetData(std::vector<Branch>& branches, std::vector<Node>& nodes, std::vector<Route>& routes,

std::vector<int>& testNodes) {

if (IS\_TEST\_TIME == 1) {

n = TEST\_HYPERNET\_NODES;

m = TEST\_HYPERNET\_BRANCHES;

k = TEST\_HYPERNET\_EDGES;

std::cout << "Input graph : H(" << n << ", " << m << ", " << k << ")" << std::endl;

return;

}

if (!input.is\_open()) {

throw "GetData: File can not be opened!";

}

char str[50];

input.getline(str, 50);

std::cout << "Input graph : " << str << std::endl;

int buf;

input >> buf; n = buf;

input >> buf; m = buf;

input >> buf; k = buf;

if (IS\_TEST\_HYPERNET == 1) {

input >> buf; l = buf;

}

// Fill all nodes

for (int i = 0; i < n; i++) {

Node node = Node(i, false);

nodes.push\_back(node);

}

std::vector<std::vector<int>> branchRouteIds;

// Read all branches from input.txt

for (int i = 0; i < m; i++) {

input >> buf; int id = buf;

input >> buf; int firstNode = buf - 1;

input >> buf; int secondNode = buf - 1;

std::vector<int> vector;

input >> buf;

while (buf != 0) {

int routeId = buf;

vector.push\_back(routeId);

input >> buf;

}

branchRouteIds.push\_back(vector);

branches.push\_back(Branch::GetSimpleBranch(id, firstNode, secondNode));

if (!IsUniqueId(branches, id)) {

throw "GetData: not unique branch id";

}

}

// Read all routes from input.txt

for (int i = 0; i < k; i++) {

std::vector<int> vector;

input >> buf; int id = buf;

input >> buf;

while (buf != 0) {

int node = buf - 1;

vector.push\_back(node);

input >> buf;

}

auto ptr = std::make\_shared<std::vector<int>>(vector);

routes.emplace\_back(id, ptr);

if (!IsUniqueId(routes, id)) {

throw "GetData: not unique route id";

}

}

// Read all routes from input.txt

for (int i = 0; i < l; i++) {

input >> buf;

testNodes.push\_back(--buf);

}

// Fill branch Routes by ids

for (int i = 0; i < branches.size(); i++) {

auto vector = branchRouteIds[i];

for (auto &routeId : vector) {

auto it = std::find\_if(routes.begin(), routes.end(), [routeId](Route &item) ->

bool { return routeId == item.Id; });

if (it != routes.end()) {

branches[i].GetRoutes().push\_back(routes[it - routes.begin()]);

}

}

}

// Input should end by $$$

input >> str;

if (strcmp(str, "$$$") != 0) {

throw "GetData: Incorrect entry";

}

}

// вычисление биномиальных коэффицентов

void ComputeBinomialCoefficients() {

Bin.resize(m + 1, Branch::GetBranch(m + 1, 0));

for (int i = 0; i < Bin.size(); i++) {

Bin[i].SetPower(i);

if (i != 0) {

for (int j = 1; j < m + 1; j++) {

Bin[i].GetC()[j] = Bin[i - 1].GetC()[j - 1] + Bin[i - 1].GetC()[j];

}

}

}

}

// метод необходимый для изменения размера ветви

void NormalizeSolution(Branch &branch){

branch.GetC().resize(m + 1);

if (branch.GetPower() < m) {

branch = branch \* Bin[m - branch.GetPower()];

}

}

// отправления объекта ядру для расчёта

template <class T>

void Send(const T &object, int processorNumber){

std::stringstream ss;

boost::archive::binary\_oarchive oarchive{ss};

oarchive << object;

int length = ss.str().size();

TotalBytesTransfer += length;

MPI\_Send((void \*) ss.str().c\_str(), length, MPI\_BYTE, processorNumber, SEND\_RECV\_TAG, MPI\_COMM\_WORLD);

}

// получение объекта ядру для расчёта

template <class T>

T Recv() {

MPI\_Status status;

MPI\_Probe(MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

int length;

MPI\_Get\_count(&status, MPI\_PACKED, &length);

char data[length + 1];

MPI\_Recv(data, length, MPI\_PACKED, MPI\_ANY\_SOURCE, SEND\_RECV\_TAG, MPI\_COMM\_WORLD, &status);

data[length] = '\0';

std::stringstream ss;

ss.write((const char \*) data, length);

boost::archive::binary\_iarchive iarchive(ss);

T object;

iarchive >> object;

return object;

}

// управление исполнением программы используя MPI (уровень 2)

void SendControl(std::vector<H> &hypernetList, int &size) {

std::stack<int> freeProcessors;

for (int i = 1; i < size; ++i) {

freeProcessors.push(i);

}

while (!freeProcessors.empty() && !hypernetList.empty()) {

DTO data = DTO(hypernetList.back(), Branch::GetBranch(0));

hypernetList.pop\_back();

Send(data, freeProcessors.top());

freeProcessors.pop();

}

int value;

MPI\_Status status;

while (freeProcessors.size() != size - 1) {

MPI\_Probe(MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

if (status.MPI\_TAG == I\_AM\_FREE\_TAG) {

MPI\_Recv(&value, 0, MPI\_INT, status.MPI\_SOURCE, I\_AM\_FREE\_TAG, MPI\_COMM\_WORLD, &status);

if (!hypernetList.empty()) {

DTO data = DTO(hypernetList.back(), Branch::GetBranch(0));

hypernetList.pop\_back();

Send(data, status.MPI\_SOURCE);

} else {

freeProcessors.push(status.MPI\_SOURCE);

}

} else if (status.MPI\_TAG == I\_NEED\_HELP\_TAG) {

MPI\_Recv(&value, 0, MPI\_INT, status.MPI\_SOURCE, I\_NEED\_HELP\_TAG, MPI\_COMM\_WORLD, &status);

int helpProcessor = 0;

if (freeProcessors.size() > 0) {

helpProcessor = freeProcessors.top();

freeProcessors.pop();

}

MPI\_Send(&helpProcessor, 1, MPI\_INT, status.MPI\_SOURCE, I\_NEED\_HELP\_TAG, MPI\_COMM\_WORLD);

}

}

}

// управление исполнением программы используя MPI (уровень 1)

void SendControl(H &H, int &size) {

std::stack<int> freeProcessors;

for (int i = 1; i < size; ++i) {

freeProcessors.push(i);

}

DTO data = DTO(H, Branch::GetBranch(0));

Send(data, freeProcessors.top());

freeProcessors.pop();

int value;

MPI\_Status status;

while (freeProcessors.size() != size - 1) {

MPI\_Probe(MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

if (status.MPI\_TAG == I\_AM\_FREE\_TAG) {

MPI\_Recv(&value, 0, MPI\_INT, status.MPI\_SOURCE, I\_AM\_FREE\_TAG, MPI\_COMM\_WORLD, &status);

freeProcessors.push(status.MPI\_SOURCE);

} else if (status.MPI\_TAG == I\_NEED\_HELP\_TAG) {

MPI\_Recv(&value, 0, MPI\_INT, status.MPI\_SOURCE, I\_NEED\_HELP\_TAG, MPI\_COMM\_WORLD, &status);

int helpProcessor = 0;

if (freeProcessors.size() > 0) {

helpProcessor = freeProcessors.top();

freeProcessors.pop();

}

MPI\_Send(&helpProcessor, 1, MPI\_INT, status.MPI\_SOURCE, I\_NEED\_HELP\_TAG, MPI\_COMM\_WORLD);

}

}

}

// вычисление MENC

void ComputeMENC(const H& initialHypernet, int &size) {

std::vector<H> hypernetList;

for (int i = 1; i < n; i++) {

auto H = initialHypernet;

if (i != 1) {

H.RenumerateNodes(i, 1);

}

if (H.IsSNconnected()) {

if (IS\_TWO\_LEVEL\_PARALLELIZATION == 1) {

hypernetList.push\_back(H);

} else {

SendControl(H, size);

}

}

}

if (IS\_TWO\_LEVEL\_PARALLELIZATION == 1) {

SendControl(hypernetList, size);

}

}

// вычисление APC

void ComputeAPC(const H& initialHypernet, int &size) {

std::vector<H> hypernetList;

for (int i = 0; i < n; i++) {

for (int j = i + 1; j < n; j++) {

auto H = initialHypernet;

if (i != 0 || j != 1) {

if (i != 0 && j != 1) {

H.RenumerateNodes(i, 0);

H.RenumerateNodes(j, 1);

}

if (i == 0 && j != 1) {

H.RenumerateNodes(j, 1);

}

if (i != 0 && j == 1) {

H.RenumerateNodes(i, 0);

}

}

if (H.IsSNconnected()) {

if (IS\_TWO\_LEVEL\_PARALLELIZATION == 1) {

hypernetList.push\_back(H);

} else {

SendControl(H, size);

}

}

}

}

if (IS\_TWO\_LEVEL\_PARALLELIZATION == 1) {

SendControl(hypernetList, size);

}

}

// обработчик ошибок

void ErrorHandler(const char \*str) {

std::cout << "--------------------------------" << std::endl;

std::cout << "Occurred next error:" << std::endl;

std::cout << str << std::endl;

std::cout << "--------------------------------" << std::endl;

}

// вычисление гиперсети

void ComputeHypernet(H &initialHypernet, int &size, int &option) {

initialHypernet.RemoveEmptyBranches();

if (option == 1) {

ComputeAPC(initialHypernet, size);

} else if (option == 2) {

ComputeMENC(initialHypernet, size);

}

}

// отправка дыннх от процесса мастера испольнителям

void BcastDataByMaster() {

std::stringstream ss;

boost::archive::binary\_oarchive oarchive{ss};

oarchive << Bin;

int length = ss.str().size();

TotalBytesTransfer += length + 4\*sizeof(int);

MPI\_Bcast(&length, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

MPI\_Bcast((void \*) ss.str().c\_str(), length, MPI\_BYTE, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

MPI\_Bcast(&n, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

MPI\_Bcast(&m, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

MPI\_Bcast(&k, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

}

// вывод расчётных данных

void PrintSolution(Branch &solution, double &time) {

std::cout << "Time of programm " << time << " sec" << std::endl;

std::cout << "PairConnectivityCalls " << PairConnectivityCalls << std::endl;

std::cout << "Reductions : " << std::endl;

std::cout << " UnconnectedNodesReduced " << UnconnectedNodesReduced << std::endl;

std::cout << " EdgesReduced " << EdgesReduced << std::endl;

std::cout << " ChainsReduced " << ChainsReduced << std::endl;

if (IS\_DEBUG == 1) {

std::cout << " ComplexChains " << ComplexChains << std::endl;

}

std::cout << "Were ends of recursion : " << ReliableHypernets + UnconnectedHypernets +

TwoNodesHypernets << std::endl;

std::cout << " ReliableHypernets " << ReliableHypernets << std::endl;

std::cout << " UnconnectedHypernets " << UnconnectedHypernets << std::endl;

std::cout << " TwoNodesHypernets " << TwoNodesHypernets << std::endl;

if (IS\_DEBUG == 1) {

std::cout << "TotalBytesTransfer : " << TotalBytesTransfer << std::endl;

std::cout << "HelpProcessors : " << HelpProcessors << std::endl;

}

if (!solution.IsZero()) {

NormalizeSolution(solution);

std::cout << "Value at point " << p << ": " << std::setprecision(11) << solution.GetPolynomialValue(p)

<< std::endl;

} else {

std::cout << "unconnected hypernet" << std::endl;

}

}

// получение решения

Branch GetSolution(int &size, int &option, std::vector<Branch> branches, std::vector<Node> nodes,

std::vector<Route> routes, double &time) {

double startTime, averageTime = 0; // переменные начала времени расчёта и среднего времени расчёта

// выбор тестирования по конфигурации приложения (время, оптимизация, расчёт гиперсети)

if (IS\_TEST\_TIME == 1) {

for (int i = 0; i < TEST\_HYPERNETS; i++) {

H initialHypernet = GetRandomHypernet();

if (IS\_DEBUG == 1) {

initialHypernet.LogHypernet();

}

startTime = MPI\_Wtime();

ComputeHypernet(initialHypernet, size, option);

double endTime = MPI\_Wtime();

output << "Time = " << endTime - startTime << std::endl;

averageTime += endTime - startTime;

}

} else {

H initialHypernet;

// выбор метода получения гиперстеи (случайно или из аргументов функции)

if (IS\_TEST\_HYPERNET == 1) {

initialHypernet = GetRandomHypernet(branches, nodes);

if (IS\_DEBUG == 1) {

initialHypernet.LogHypernet();

}

} else {

initialHypernet = H(std::move(branches), std::move(nodes), std::move(routes));

}

startTime = MPI\_Wtime();

ComputeHypernet(initialHypernet, size, option);

}

// запрос значений от процессов испольнителей

for (int i = 1; i < size; i++) {

MPI\_Send(&i, 0, MPI\_INT, i, SEND\_SOLUTION\_TAG, MPI\_COMM\_WORLD);

}

Branch sum;

// сохранение среденего времени

if (IS\_TEST\_TIME == 1 && TEST\_HYPERNETS > 1) {

averageTime = averageTime / TEST\_HYPERNETS;

output << "Average time = " << averageTime;

return sum;

}

MPI\_Status status;

for (int i = 1; i < size; i++) {

Branch branch = Recv<Branch>(); // получение значения от процесса исполнителя

sum = sum + branch;

int buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, RELIABLE\_HYPERNETS\_COUNT\_TAG, MPI\_COMM\_WORLD, &status);

ReliableHypernets += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, UNCONNECTED\_HYPERNET\_COUNT\_TAG, MPI\_COMM\_WORLD, &status);

UnconnectedHypernets += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, TWO\_NODES\_HYPERNET\_COUNT, MPI\_COMM\_WORLD, &status);

TwoNodesHypernets += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, CHAINS\_REDUCED\_TAG, MPI\_COMM\_WORLD, &status);

ChainsReduced += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, UNCONNECTED\_NODES\_REDUCED\_TAG, MPI\_COMM\_WORLD, &status);

UnconnectedNodesReduced += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, PAIR\_CONNECTIVITY\_CALLS\_TAG, MPI\_COMM\_WORLD, &status);

PairConnectivityCalls += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, EDGES\_REDUCED\_TAG, MPI\_COMM\_WORLD, &status);

EdgesReduced += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, UNSIMPLE\_CHAINS\_TAG, MPI\_COMM\_WORLD, &status);

ComplexChains += buff;

MPI\_Recv(&buff, 1, MPI\_INT, MPI\_ANY\_SOURCE, HELP\_PROCESSORS\_TAG, MPI\_COMM\_WORLD, &status);

HelpProcessors += buff;

unsigned long long int longBuff;

MPI\_Recv(&longBuff, 1, MPI\_UNSIGNED\_LONG\_LONG, MPI\_ANY\_SOURCE, TOTAL\_BYTES\_TRANSFER\_TAG, MPI\_COMM\_WORLD, &status);

TotalBytesTransfer += longBuff;

}

// обработка полченного значения в завсимости от выбранного критериия

if (option == 1) {

if (IS\_NUMBER\_COMPUTATION == 1) {

sum.SetValue(sum.GetValue() / Bin[n].GetC()[2]);

} else {

for (int i = 0; i < sum.GetC().size(); i++) {

auto sumVector = sum.GetC();

sumVector[i] = sumVector[i] / Bin[n].GetC()[2];

sum.SetC(sumVector);

}

}

} else if (option == 2) {

sum = sum + Branch::GetUnity();

}

time = MPI\_Wtime() - startTime;

return sum;

}

// инициализация работы процесса мастера

void Master(int size) {

input.open("input.txt");

output.open("output.txt");

setlocale(LC\_ALL, "");

std::vector<Branch> branches;

std::vector<Node> nodes;

std::vector<Route> routes;

std::vector<int> testNodes;

GetData(branches, nodes, routes, testNodes);

std::cout << "Press 1 to get APC polynomial" << std::endl;

std::cout << "Press 2 to get MENC polynomial" << std::endl;

int option;

std::cin >> option;

if (option != 1 && option != 2) {

std::cout << "Wrong number" << std::endl;

return;

}

ComputeBinomialCoefficients();

BcastDataByMaster();

double time;

if (IS\_TEST\_HYPERNET != 1) {

Branch solution = GetSolution(size, option, branches, nodes, routes, time);

PrintSolution(solution, time);

for (auto &item : solution.GetC()) {

output << std::setprecision(14) << item << " ";

}

output << std::endl;

} else {

for(int i=0; i<testNodes.size() - 1; i++) {

FirstRoot = testNodes[i];

for(int j=i+1; j<testNodes.size(); j++) {

SecondRoot = testNodes[j];

Branch solution = GetSolution(size, option, branches, nodes, routes, time);

if (IS\_DEBUG == 1) {

PrintSolution(solution, time);

}

output << FirstRoot + 1 << " " << SecondRoot + 1 << " " << TreeNodeIntersections << " "

<< UnconnectedTreeNodes << " ";

output << std::setprecision(15) << solution.GetPolynomialValue(p) << " ";

output << std::setprecision(15) << solution.GetPolynomialValue(0.99) << " ";

output << time << std::endl;

TreeNodeIntersections = 0;

UnconnectedTreeNodes = 0;

}

}

}

for (int i = 1; i < size; i++) {

MPI\_Send(&i, 0, MPI\_INT, i, STOP\_TAG, MPI\_COMM\_WORLD);

}

}

// полчение дыннх от процесса мастера

void BcastDataBySlaves() {

int length;

MPI\_Bcast(&length, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

char data[length + 1];

MPI\_Bcast(data, length, MPI\_BYTE, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

data[length] = '\0';

std::stringstream ss;

ss.write((const char \*) data, length);

boost::archive::binary\_iarchive iarchive(ss);

iarchive >> Bin;

MPI\_Bcast(&n, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

MPI\_Bcast(&m, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

MPI\_Bcast(&k, 1, MPI\_INT, HOST\_PROCESSOR, MPI\_COMM\_WORLD);

}

// инициализация работы процессов исполнителей

void Slaves(int rank) {

BcastDataBySlaves();

int value;

Branch sum;

MPI\_Status status;

do {

MPI\_Probe(MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

if (status.MPI\_TAG == SEND\_RECV\_TAG) {

DTO data = Recv<DTO>();

sum = sum + PairConnectivity(data.H, data.Branch);

MPI\_Send(&value, 0, MPI\_INT, HOST\_PROCESSOR, I\_AM\_FREE\_TAG, MPI\_COMM\_WORLD);

}

if (status.MPI\_TAG == SEND\_SOLUTION\_TAG) {

MPI\_Recv(&value, 1, MPI\_INT, MPI\_ANY\_SOURCE, SEND\_SOLUTION\_TAG, MPI\_COMM\_WORLD, &status);

Send(sum, HOST\_PROCESSOR);

MPI\_Send(&ReliableHypernets, 1, MPI\_INT, HOST\_PROCESSOR, RELIABLE\_HYPERNETS\_COUNT\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&UnconnectedHypernets, 1, MPI\_INT, HOST\_PROCESSOR, UNCONNECTED\_HYPERNET\_COUNT\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&TwoNodesHypernets, 1, MPI\_INT, HOST\_PROCESSOR, TWO\_NODES\_HYPERNET\_COUNT, MPI\_COMM\_WORLD);

MPI\_Send(&ChainsReduced, 1, MPI\_INT, HOST\_PROCESSOR, CHAINS\_REDUCED\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&UnconnectedNodesReduced, 1, MPI\_INT, HOST\_PROCESSOR, UNCONNECTED\_NODES\_REDUCED\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&PairConnectivityCalls, 1, MPI\_INT, HOST\_PROCESSOR, PAIR\_CONNECTIVITY\_CALLS\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&EdgesReduced, 1, MPI\_INT, HOST\_PROCESSOR, EDGES\_REDUCED\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&ComplexChains, 1, MPI\_INT, HOST\_PROCESSOR, UNSIMPLE\_CHAINS\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&HelpProcessors, 1, MPI\_INT, HOST\_PROCESSOR, HELP\_PROCESSORS\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&TotalBytesTransfer, 1, MPI\_UNSIGNED\_LONG\_LONG, HOST\_PROCESSOR, TOTAL\_BYTES\_TRANSFER\_TAG,

MPI\_COMM\_WORLD);

sum = Branch::GetZero();

ReliableHypernets = 0;

UnconnectedHypernets = 0;

TwoNodesHypernets = 0;

ChainsReduced = 0;

UnconnectedNodesReduced = 0;

PairConnectivityCalls = 0;

EdgesReduced = 0;

ComplexChains = 0;

HelpProcessors = 0;

TotalBytesTransfer = 0;

}

} while (status.MPI\_TAG != STOP\_TAG);

}

int main(int argc, char \*\*argv) {

int rank, size;

// инциализация MPI

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (size < 2) {

if (rank == 0)

std::cerr << "Require at least 2 tasks" << std::endl;

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

// обработчик ошибок

try {

rank == 0 ? Master(size) : Slaves(rank);

} catch (const std::overflow\_error &e) {

std::cout << "throw std::overflow\_error (same type rule)" << std::endl << e.what();

} catch (const std::runtime\_error &e) {

std::cout << "throw std::underflow\_error (base class rule)" << std::endl << e.what();

} catch (const std::exception &e) {

std::cout << "throw std::logic\_error (base class rule)" << std::endl << e.what();

} catch (const char \*str) {

ErrorHandler(str);

} catch (...) {

std::cout << "throw std::string or int or any other unrelated type";

}

MPI\_Finalize(); // окончание работы MPI

input.close();

output.close();

return 0;

}

#include "Funcs.h"

#include "Globals.h"

// генератор случайных сетей для построения случайной гиперсети

std::vector<Branch> GetRandomNetwork(int &nodesCount, int &edgeCount) {

std::vector<Branch> network;

std::vector<int> nodes1; // набор вершин 1

for (int i = 0; i < nodesCount; i++) {

nodes1.push\_back(i);

}

std::vector<int> nodes2; // набор вершин 2

int j = rand() % (nodes1.size() - 1);

// цикл построения случайного дерева, гарантирует связность сети

for (int i = 0; i < nodesCount - 1; i++) {

nodes2.push\_back(nodes1[j]);

nodes1.erase(nodes1.begin() + j);

int firstPosition = nodes1.size() > 1 ? rand() % (nodes1.size() - 1) : 0;

int secondPosition = nodes2.size() > 1 ? rand() % (nodes2.size() - 1) : 0;

j = firstPosition;

network.push\_back(Branch::GetSimpleBranch(network.size(), nodes1[firstPosition], nodes2[secondPosition]));

}

// расстановка оставшихся рёбер

for (int i = 0; i < edgeCount - nodesCount + 1; i++) {

int firstNode = rand() % nodesCount, secondNode = rand() % nodesCount;

while (firstNode == secondNode)

secondNode = rand() % nodesCount;

network.push\_back(Branch::GetSimpleBranch(network.size(), firstNode, secondNode));

}

return network;

}

// получение случайного дерева заданного размера с запретами использования конкретных вершин

std::vector<Branch> GetRandomTree(const int &nodesCount, std::vector<int> &nodes,

const std::vector<int> &forbiddenNodes) {

std::vector<Branch> tree;

while (nodes.size() != nodesCount) {

int node = rand() % (n - 1);

std::vector<int> newForbiddenNodes;

newForbiddenNodes.reserve(nodes.size() + forbiddenNodes.size());

newForbiddenNodes.insert(newForbiddenNodes.end(), nodes.begin(), nodes.end());

newForbiddenNodes.insert(newForbiddenNodes.end(), forbiddenNodes.begin(), forbiddenNodes.end());

if (std::find(newForbiddenNodes.begin(), newForbiddenNodes.end(), node) == newForbiddenNodes.end()) {

nodes.push\_back(node);

}

}

std::vector<int> firstSetNodes = nodes;

std::vector<int> secondSetNodes;

int j = rand() % (firstSetNodes.size() - 1);

for (int i = 0; i < nodesCount - 1; i++) {

secondSetNodes.push\_back(firstSetNodes[j]);

firstSetNodes.erase(firstSetNodes.begin() + j);

int firstPosition = firstSetNodes.size() > 1 ? rand() % (firstSetNodes.size() - 1) : 0;

int secondPosition = secondSetNodes.size() > 1 ? rand() % (secondSetNodes.size() - 1) : 0;

j = firstPosition;

tree.push\_back(Branch::GetSimpleBranch(-1, firstSetNodes[firstPosition], secondSetNodes[secondPosition]));

}

return tree;

}

// укладка ребра в первичную сеть

void Mapping(const std::vector<Branch> &primaryNetwork, std::vector<std::vector<int>> &nodeRotes,

std::vector<std::vector<Branch>> &branchRoutes, std::vector<bool> &isVisited, std::vector<int> &nodeRote,

std::vector<Branch> &branchRoute, int &node, const int &endNode, const int &startTime) {

isVisited[node] = true;

if(node == endNode) {

nodeRotes.push\_back(nodeRote);

branchRoutes.push\_back(branchRoute);

}

for(auto &branch : primaryNetwork) {

int searchTime = clock() - startTime;

if (searchTime >= TEST\_HYPERNET\_MAPPING\_TIME) {

throw "Timeout";

}

if (IS\_TEST\_HYPERNET\_SIMPLE\_MAPPING == 1 && !nodeRotes.empty() && !branchRoutes.empty()) {

return;

}

if (H::IsIncident(node, branch)) {

int incidentNode = branch.GetFirstNode() != node ? branch.GetFirstNode() : branch.GetSecondNode();

if (!isVisited[incidentNode]) {

nodeRote.push\_back(incidentNode);

branchRoute.push\_back(branch);

Mapping(primaryNetwork, nodeRotes, branchRoutes, isVisited, nodeRote, branchRoute, incidentNode,

endNode, startTime);

nodeRote.erase(std::find(nodeRote.begin(), nodeRote.end(), incidentNode));

auto it = std::find\_if(branchRoute.begin(), branchRoute.end(), [branch](Branch &item) -> bool {

return item == branch;

});

branchRoute.erase(it);

}

}

}

isVisited[node] = false;

}

// получение укладки ребра в первичную сеть

void SetMapping(std::vector<Branch> &primaryNetwork, std::vector<Branch> &secondaryNetwork,

std::vector<Route> &routes) {

for (auto &edge : secondaryNetwork) {

int node = edge.GetFirstNode();

std::vector<bool> isVisited(n, false);

std::vector<int> nodeRote{node};

std::vector<Branch> branchRoute;

std::vector<std::vector<int>> nodeRotes;

std::vector<std::vector<Branch>> branchRoutes;

Mapping(primaryNetwork, nodeRotes, branchRoutes, isVisited, nodeRote, branchRoute, node, edge.GetSecondNode(),

clock());

int randomIndex = nodeRotes.size() > 1 ? rand() % (nodeRotes.size() - 1) : 0;

nodeRote = nodeRotes[randomIndex];

branchRoute = branchRoutes[randomIndex];

routes.emplace\_back(routes.size(), std::make\_shared<std::vector<int>>(nodeRote));

for(auto &item : branchRoute) {

std::find(primaryNetwork.begin(), primaryNetwork.end(), item) -> GetRoutes().push\_back(routes.back());

}

}

}

// получение случайной гиперсети необходимой для работы алгоритма оптимизации (возможен timeout)

H TryGetRandomHypernet(std::vector<Branch> primaryNetwork, std::vector<Node> &nodes) {

srand(seed++);

std::vector<int> firstTreeNodes{FirstRoot};

std::vector<int> forbiddenNodes{SecondRoot};

auto firstRandomTree = GetRandomTree(TEST\_HYPERNET\_TREE\_SIZE, firstTreeNodes, forbiddenNodes);

std::vector<Route> routes;

SetMapping(primaryNetwork, firstRandomTree, routes);

if (IS\_TEST\_HYPERNET\_DOUBLE\_TREE\_MAP == 1) {

std::vector<int> secondTreeNodes{FirstRoot};

forbiddenNodes.clear();

auto secondRandomTree = GetRandomTree(TEST\_HYPERNET\_TREE\_SIZE, secondTreeNodes, forbiddenNodes);

SetMapping(primaryNetwork, secondRandomTree, routes);

for (auto &node : firstTreeNodes) {

if (std::find(secondTreeNodes.begin(), secondTreeNodes.end(), node) != secondTreeNodes.end()) {

TreeNodeIntersections++;

}

}

UnconnectedTreeNodes = n - firstTreeNodes.size() - secondTreeNodes.size() +

TreeNodeIntersections;

}

return H(std::move(primaryNetwork), std::move(nodes), std::move(routes));

}

// получение случайной гиперсети

H GetRandomHypernet() {

srand(seed++);

auto primaryNetwork = GetRandomNetwork(n, m);

auto secondaryNetwork = GetRandomNetwork(n, k);

std::vector<Route> routes;

SetMapping(primaryNetwork, secondaryNetwork, routes);

std::vector<Node> nodes;

for (int i = 0; i < n; i++) {

Node node = Node(i, false);

nodes.push\_back(node);

}

return H(std::move(primaryNetwork), std::move(nodes), std::move(routes));

}

// получение случайной гиперсети необходимой для работы алгоритма оптимизации

H GetRandomHypernet(std::vector<Branch> &primaryNetwork, std::vector<Node> &nodes) {

try {

return TryGetRandomHypernet(primaryNetwork, nodes);

} catch (const char \*str) {

if (IS\_DEBUG == 1) {

output << str << std::endl;

}

TreeNodeIntersections = 0;

UnconnectedTreeNodes = 0;

return GetRandomHypernet(primaryNetwork, nodes);

}

}