**ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ**

**Ինֆորմատիկայի և կիրառական մաթեմատիկայի ֆակուլտետ**

**Ծրագրավորման և ինֆորմացիոն տեխնոլոգիաների ամբիոն**

Նարեկ Աթայան Ռազմիկի

Պատահական ցանցերի հետազոտման ոչ- ստանդարտ ալգորիթմների իրականացումը զուգահեռ ծրագրավորման օգտագործմամբ՝ MPI-ի հիման վրա:

“Հաշվողական համալիրների, համակարգերի և ցանցերի մաթեմատիկական և ծրագրային ապահովում” մասնագիտությամբ

**Դիպլոմային աշխատանք**

**Ղեկավար՝ ֆիզ. մաթ. գիտությունների թեկնածու, Ս. Ավետիսյան**

**Երևան 2016թ.**

СОДЕРЖАНИЕ

**Введение и Постановка задачи--------------------------------------------------------3**

**ГЛАВА 1: Случайные сети-----------------------------------------------------------------5**

1.1: Класическая модель случайных сетей-------------------------------------------6

1.2: Блочно-иерархическая модель-----------------------------------------------------7

1.3: Характеристики случайной сети---------------------------------------------------8

**ГЛАВА 2:** **Реализованне алгоритмы, величины и меры ----------------10**

2.1: Основные Алгоритмы-----------------------------------------------------------------10

2.2: Меры Centrality--------------------------------------------------------------------------13

2.3: Community Detection------------------------------------------------------------------16

**ГЛАВА 3: Используемые инструменты и библиотеки--------------------17**

**ГЛАВА 4: Конфигурация системы----------------------------------------------------21**

**ГЛАВА 5: Архитектура системы-------------------------------------------------------27**

4.1: Архитектура подсистемы “аlternate property computer” -------------------28

4.2: Архитектура подсистемы “graph” ------------------------------------------------ 30

**ГЛАВА 6: Рукаводство для пользователя----------------------------------37**

**Источники и литература------------------------------------------------------------------38**

ВВЕДЕНИЕ И ПОСТАНОВКА ЗАДАЧИ

Начиная с 50-ых годов прошлого века в различных областях науки возник повышенный интерес к большим сетям различных топологий. Началом их изучения можно связать с работами, связанными с случайными графами (ЭР-графы). Случайные графы были использованы для моделирования различных парадигм.

Важность изучения моделей сложных сетей и случайных графов объясняется тем, что выявление статистических свойств дало возможность для более глубокого понимания природы реальных сложных сетей некоторых типов. Сложные сетеподобные конструкции описывают широкое множество систем огромной технологической и интеллектуальной важности.

Теория случайных графов была основана венгерскими математиками Паулем Эрдосом и Альфредом Реньи, и на сегодня есть множество теоретических результатов в этой области. В действительности, топология реальных сетей отклоняется от случайного графа, и необходимо развить инструменты и способы, чтобы описать в количественных терминах основные принципы организации сложных сетей. Необходимо понять топологии взаимодействий между компонентами сети.

Изучение случайных сетей было начато желанием понять различные реальные системы, начиная от сетей коммуникаций до экологических.

Укажем также на новую область приложений алгоритмической теории графов - разработка и реализация параллельных алгоритмов и программ (программирование для суперкомпьютеров, грид-технологии, облачные вычисления).

Названные две области приложения графовых алгоритмов связаны с решением важнейших экстремальных задач на графах и сетях: задачи о крат­чайшем пути и задачи о максимальном потоке.

Социальные сети — это сети в которых вершины есть люди или в некототрых случаях группылюдейа ребра представляют социальные связи между ними. Приемущества таких сетей состоит во многих вероятных интерпретациях касательно того что подразумевается под понятием ребер, и эти понятия могут варьироваться в зависимости от постановки задачи. Ребра могут быть интерпретированы как дружеские, но в то же время как и професионалжные или бизнес а также романтическо-сексуальные связи.

Если мы распологаем информацией о структуре сети, то мыможем вычислить из этого различные полезные величины или меры, которые определяют тополгические особенности сети. Например, такая величина как **Centrality** идентифицирует наиболиее важные вершины сети, в практике мы можем определить наиболее влиятельных персон в социальных сетях, ключевыые инфраструктуры узлов в интернете или в городских сетях, наиболее активныж [переносчиков](https://en.wikipedia.org/wiki/Super-spreader) болезней.

Современная наука сетей принесла значительные успехи в понимание сложных систем. Одним из наиболее важных особенностей графиков, представляющих реальные системы, является структура сообщества, или кластеризация, т. е. организация вершин в кластерах, со многими ребрами, соединяющих вершины одного и того же кластера и сравнительно малыми ребрами, соединяющих вершины из разных кластеров. Такие кластеры, или **общины**, можно рассматривать в качестве достаточно независимых отделений графа, играют такую же роль, как ткани или органы в человеческом теле. Обнаружение общин имеет большое значение в социологии, биологии и информатики, дисциплин, где системы часто представлены в виде графов.

Целью данной работы являетсяа разработка универсальной системы-инструмента для вычисления и анализа статистических свойств различных моделей сложных сетей. Данная система ориентирована на кластерные вычисления с возможностью ее использования для анализа сверхбольших сетей. Данная система обладает большой гибкостью и давольно проста в использовании, засчет нового удобного пользовательского интерфейса включаючего в себя коммандную строку и конфигурационний файл.

Одна из задач заключается в изучении и сравнении существующих методов выделения сообществ, а также в создании новых, более качественных методов. Дополнительная информация, которая может содержаться в вершинах графа, не рассматривалась.

Работа может быть разделена на несколько частей.

1. Ввод необходимых понятий и терминов.
2. Обзор реализованных методов выделения сообществ.
3. Поиск способов объединения результатов работы разных алгоритмов с целью улучшить результат.
4. Тестирование и сравнение методов на модельных и реальных данных.

Отметим, что третий пункт является основным в данной работе. Многие методы имеют в своей основе разные подходы, поэтому логично попробовать объединить эти методы, чтобы компенсировать их слабые стороны. Тогда можно рассчитывать на получение более устойчивого и качественного результата. — заключается в поиске способов агрегирования результатов разных методов выделения сообществ.

ГЛАВА 1: СЛУЧАЙНЫЕ СЕТИ

КЛАСИЧЕСКАЯ МОДЕЛЬ СЛУЧАЙНЫХ СЕТЕЙ

Исторически изучением сложных сетей занималась теория графов. Но, начиная с 1950-ых годов, сложные сети, не имевшие очевидных принципов построения, стали описывать с помощью теории случайных графов, предложенных в качестве наиболее подходящей модели сложных сетей. Впервые случайные графы были изучены венгерскими математиками Полем Эрдосом и Альфредом Реньи. Они предложили свою модель (модель Erdős – Rényi) построения и представления случайных графов. Согласно модели Erdős – Rényi, имеем *N* изолированных вершин и, соединяя каждую пару вершин с вероятностью *p*, создаем граф с приблизительно  случайно распределенными ребрами.

Эрдос и Реньи определяют случайный граф как *N* помеченных вершин, соединенных *n* ребрами, которые выбираются случайным образом из  возможных.

Другое определение случайного графа называют также биноминальной моделью. В этом случае, имея *N* вершин, соединяем каждые 2 из них с вероятностью *p*. В конечном итоге, полученное количество ребер будет случайной величиной с ожидаемым значением *N* → ∞. Если *G*0 — граф с вершинами *P*1, *P*2, …, *PN* и *n* ребрами, вероятность получить его с помощью этого процесса составит:



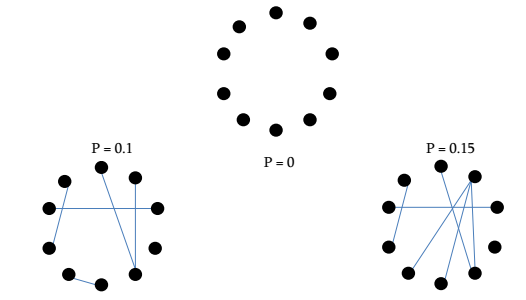


Иллюстрация процесса изменения графа в модели Erdős – Rényi. В начале имеем отдельных вершин (верхняя часть рисунка), затем каждая пара вершин соединяется с вероятностью p. Нижняя часть рисунка показывает два различных этапа формирования графа, соответствующих значениям p = 0,1 и p = 0,15. Мы можем увидеть появление деревьев (дерево порядка 3, нарисованное длинными пунктирными линиями) и циклов (цикл порядка 3, нарисованный короткими пунктирными линиями), а также соединенный кластер, который объединяет половину вершин в p = 0,15 = 1,5 / N.[1]

В данной задачи случайный граф Erdős – Rényi генерируется с помощью распределения Бернулли.

Статистические характеристики случайных сетей*.*

Чтобы понять, чем характеризуются случайные сети, надо определить и исследовать основные концепции моделирования.

***Малые миры (small - world) [3].*** Концепция малых миров довольно просто описывает тот факт, что, несмотря на их огромные размеры, в большинстве сетей существует сравнительно короткий путь между двумя любыми вершинами. Расстояние между двумя вершинами определяется как число ребер наикратчайшего пути, их соединяющего.

## Эрдосом и Реньи показано, что среднее расстояние между двумя вершинами в случайном графе растет как логарифм от числа вершин. Тем не менее, случайные графы являются малыми мирами.

*Коэффициент кластеризации [3].* Тенденция к разбиению графа на кластеры определяется коэффициентом кластеризации (Watts, Strogats 1998). Сначала выберем в сети некоторую вершину , степень которого . Если все соседи вершини были соседами друг с другом, то количество ребер между ними было бы . Отношение между числом  ребер, действительно существующих между вершинами, и общим числом ребер является значением коэффициента кластеризации вершины : . Общий коэффициент кластерности сети находится как среднее значение коэффициентов отдельных вершин. В случайном графе, поскольку ребра распределяются случайным образом, коэффициент кластерности составляет C = p. Правда, Ватс и Строгатс первыми указали на факт, что во многих, если не во всех, реальных сетях коэффициент кластерности обычно значительно больше, чем в случайных сетях с таким же количеством ребер и вершин.

***Распределение степеней [3]*.** Не все вершины сети имеют одинаковое количество ребер. Распределение количества ребер вершины, характеризуется функцией P(k), которая была определена в предыдущем подразделе как вероятность того, что случайно выбранная вершина будет иметь ровно k ребер. Последние результаты исследований показали, что для многих сетей, включая Всемирную паутину (Albert, Jeong, Baraba´si 1999), Интернет (Falautsos 1999), распределение степеней вершин является степенным: . Такие сети называют сетями без масштабирования – scale – free сети (Baraba´si, Albert 1999).

*Связность и диаметр [1]*. Диаметр графа — это максимальное расстояние между любой парой его вершин. Случайные графы имеют тенденцию иметь маленькие диаметры, если *p* не слишком маленький [1]. Диаметр случайных графов изучался многими авторами, и общее заключение такого: для большинства значений *p*, почти все графы с одинаковыми *N* и *p* имеют одинаковые диаметры. Это означает, что, когда мы рассматриваем все графы с *N* вершинами и вероятностью *p*, диапазон значений, в котором диаметры этих графов могут изменяться, очень маленький, обычно сконцентрирован вокруг цифры:

,

где  обозначаем среднюю степень вершин в графе. Для модели Erdős – Rényi получены следующие результаты:

1. если , то граф составлен из изолированных деревьев, и его диаметр равняется диаметру дерева,
2. если , то появляется гигантский кластер,а диаметр графа равняется диаметру кластера, если , и пропорционален , в противном случае,
3. если , то граф полностью связан, то есть граф полный.

БЛОЧНО-ИЕРАРХИЧЕСКАЯ МОДЕЛЬ

**Нерегулярная блочно-иерархическая сеть.** Нерегулярная блочно-иерархическая сетьявляется обобщенным случаем регулярной блочно-иерархической сети. Единственное отличие в том, что количество вложенных подграфов в какой-либо подраф не равен p, a меньше или равно p.

Обозначим через — множество подграфов уровня,

Через — *i*-тый подграф уровня , где (),

|V()| – количество узлов в подграфе ,

– номер первого узла в ,

|C()| – количество вложенных подграфов в графе .

Заметим, что

, }

Пусть =0. Подграфы нулевого уровня:

}

Каждый подграф нулевого уровня представляет собой граф где , , .

Пусть > 0.Предположим, что для уровня подграфы построены, т.е.

Подграфы уровня :

Подграф , строится посредством применения операции oб'единения и операции соединения к подграфам уровня . В обьединяются графы

.

Для каждого , номер первого кластера входящего в него: l, обозначим через .

Граф **(**нерегулярная блочно-иерархическая сеть ) определяется следующим образом:

, где .

Вершины нерегулярной блочно-иерархическая сети будем называть узлами сети, а ребра связями между узлами сети. Уровень назовем уровнем иерархии нeрегулярной блочно-иерархическая сети, а подграфы уровня - кластерами уровня нерегулярной блочно-иерархическая сети. Подграфы

,

назовем кластерами, вложенными в кластер и обозначим n-тый кластер через .

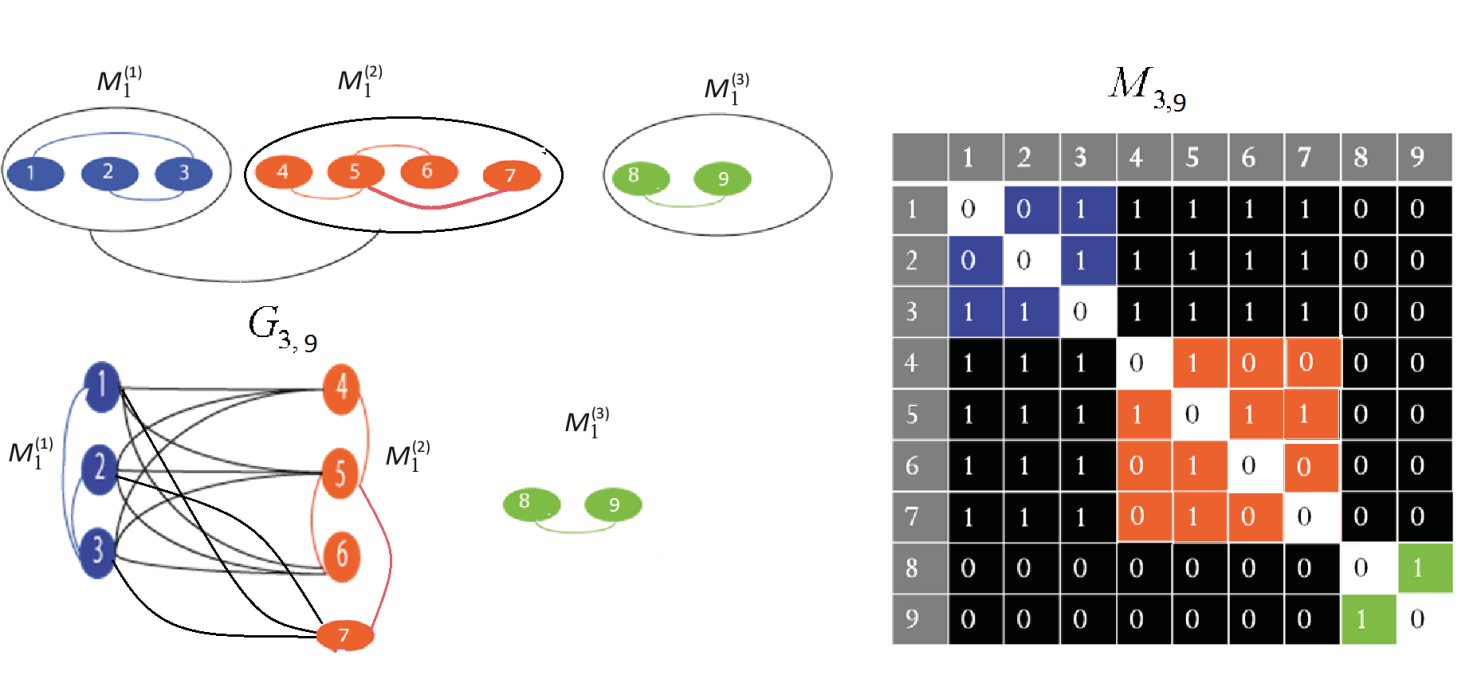
Заметим, что |V()| = , что позволяет вычислять все |V()| во время генерации. Далее во всех алгоритмах будем считать |V()| уже известными значениями.

*Определение*. Пусть нерегулярная блочно-иерархическая сеть с узлами , где - множество кластеров уровня , . Сеть назовем *упорядоченной*, если для любой пары кластеров , , где следует, что если , то i < j. Множество узлов упорядочeнной нерегулярной блочно-иерархической сети упорядочены по вложенности кластеров.



Из построения выше, нерегулярная блочно-иерархическая сеть посредством переименования узлов приводиться к упорядоченной блочно-иерархической сети. В дальнейшем будем рассматривать упорядоченные блочно-иерархические сети, которые будем называть блочно-иерархическими сетями или просто сетями.

**Матрица смежности** **блочно-иерархического сети.** Пусть блочно-иерархическая сеть, - множество его кластеров уровня . Рассмотрим матрицу смежности *Аp,N* графа . Матрица *Ap,N,* представляя смежные вершины сети, представляет также соединения между кластерами . Действительно, пусть , , некоторый кластер уровня  Кластеру соответствует , где l = |C()|, матричных блоков,расположенных вдоль главной диагонали. Каждый матричный блок представляет соединение между парой кластеров , вложенных в кластер . Значения элементов матричного блока равны 1, если два кластера соединены, и равны 0 в противном случае.Mатричный блок назовем связанным, если значения элементов блока равны *1,*  и назовем несвязaнным, если значения элементов блока равны *0*. На Рис1 дан пример блочно- иерархической сети и его матрицы связности .



*Рис1.Блочно-иерархическая матрица и соответствующая блочно-иерархическая сеть . Кластер соединен с kластером . Кластер содержит вложенные кластеры и следовательно, все вершины сети .*

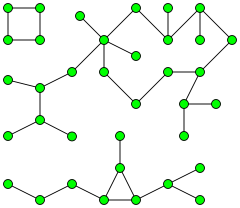
ГЛАВА 2: Реализованне Алгоритмы, величины и меры

Основные Алгоритмы

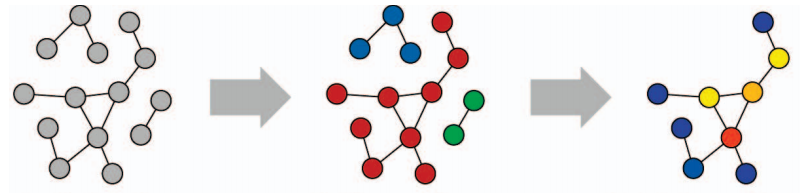
**Connected Components**

**Компонента связности графа G** (или просто **компонента графа** G) — максимальный (по включению) граф подграф графа G. Другими словами, это подграф G(U), порождённый множеством U \subseteq V(G) вершин, в котором для любой пары вершин u, v \in U в графе G существует (u, v)-цепь и для любой пары вершин u \in U, w \notin U не существует (u, w)-цепи.

Вершина, не имеющая инцидентных ребер, является отдельным связенным компонентом.

[Граф с тремя компонентами связанности]

Часто во время анализа социальных сетей исползуется следующий премер.



Запуск k-Betweenness Centrality на самом большом компоненте

Определение связанных компонент

Первоначальный граф

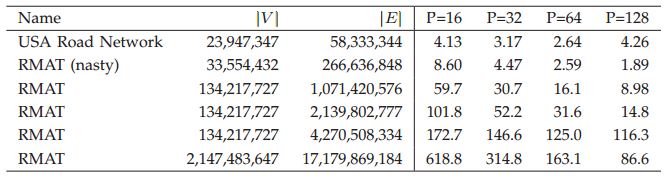
Детали Реализации

Реализованный алгоритм хорошо подходит для обшей памяти (shared memory) и делает возможным параллельизацию по ребрам.

Так, каждая вершина инициализируется уникальным собственном номером. На каждом шаге, соседние вершины жадностно окрашивают друг друга таким образом, что выигрывает вершина с наименьшим идентификатором. Процесс заканчивается, когда каждая вершина имеет тот же цвет, что и его соседи.

Число шагов пропорционально диаметру графа.

Имеющийся Результаты



[*Время запусков (в секундах) алгоритма на разных известных графax при разном каличестве потоков].*

**Shortest Path**

Алгори́тм Де́йкстры (англ. Dijkstra’s algorithm) — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, его используют протоколы маршрутизации OSPF и IS-IS.

Детали Реализации

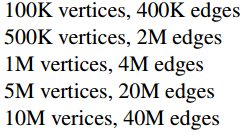
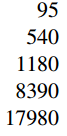
Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

**Инициализация**. Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от a до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.

**Шаг алгоритма**. Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина u, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, в которые ведут рёбра из u, назовём соседями этой вершины. Для каждого соседа вершины u, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки u и длины ребра, соединяющего u с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину u как посещённую и повторим [шаг алгоритма](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B#.D0.A8.D0.B0.D0.B3).

[Имеющийся Результаты]

*Время выпольнения работы по милисекундам*.

**Eigen Values**

В [математике](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) **спектральная** [**теория графов**](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2) — это изучение свойств [графов](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) [характеристических многочленов](https://ru.wikipedia.org/wiki/%D0%A5%D0%B0%D1%80%D0%B0%D0%BA%D1%82%D0%B5%D1%80%D0%B8%D1%81%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D1%87%D0%BB%D0%B5%D0%BD_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D1%8B), [собственных векторов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B2%D0%B5%D0%BA%D1%82%D0%BE%D1%80) и [собственных значений](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B2%D0%B5%D0%BA%D1%82%D0%BE%D1%80) матриц, связанных с графом, таких, как его [матрица смежности](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0_%D1%81%D0%BC%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8) или [матрица Кирхгофа](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0_%D0%9A%D0%B8%D1%80%D1%85%D0%B3%D0%BE%D1%84%D0%B0).

Спектральная теория графов занимается также параметрами, которые определяются путём умножения собственных значений матриц, связанных с графом, таких, как [число Колен де Вердьера](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82_%D0%9A%D0%BE%D0%BB%D0%B5%D0%BD_%D0%B4%D0%B5_%D0%92%D0%B5%D1%80%D0%B4%D1%8C%D0%B5%D1%80%D0%B0).

**Собственный вектор** — понятие в [линейной алгебре](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%B0%D1%8F_%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0), определяемое для [квадратной матрицы](https://ru.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%B4%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0) или произвольного [линейного преобразования](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BE%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) как [вектор](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), умножение матрицы на который или применение к которому преобразования даёт [коллинеарный](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%BB%D0%B8%D0%BD%D0%B5%D0%B0%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) вектор — тот же вектор, умноженный на некоторое [скалярное](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D0%B0%D0%BB%D1%8F%D1%80) значение, называемое **собственным числом** матрицы или линейного преобразования.

*Пусть  L  —* [*линейное пространство*](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) *над* [*полем*](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_(%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0)) * K ,  A\colon L \to L  —* [*линейное преобразование*](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BE%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)*.*

*Собственным вектором линейного преобразования A называется такой ненулевой* [*вектор*](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80_(%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0)) * x \in L , что для некоторого  \lambda \in K *

*\ A x = \lambda x*

*Собственным значением линейного преобразования  A называется такое число  \lambda \in K , для которого существует собственный вектор, то есть уравнение  A x = \lambda x имеет ненулевое решение  x \in L .*

*Упрощённо говоря, собственный вектор — любой ненулевой вектор x, который отображается оператором в коллинеарный \lambda x, а соответствующий скаляр \lambda называется собственным значением оператора.*

Детали Реализации

Для реализации был выбран отдельный пакет **Eigen [5].**

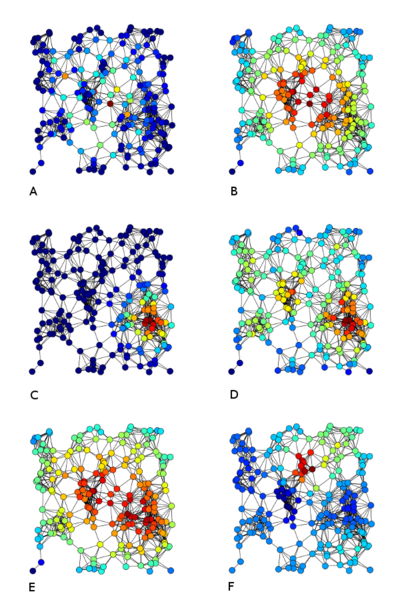
Предворительно происходит процесс конструкции матрицы смежности исходя от структуры нашего графа.

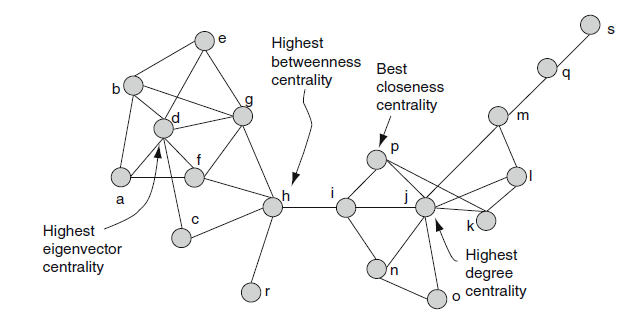
Меры Centrality

*Одно из базовых проблем в анализе сетей является измерение важности отдельной вершины или ребра в сети.*

*Количественная сосредоточенность и связность помогают нам определить участки сети, которые могут играть интересные роли. Метрика выбора зависит от применения и от топологии сети.*

*A) Betweenness centrality, B) Closeness centrality, C) Eigenvector centrality, D) Degree centrality, E) Harmonic centrality and F) Katz centralit*

****



**Betweenness Centrality**

Основой для этой меры является вычисление количеств кратчайших путей.

*Обозначим общее количество кратчайших путей между вершинами* ***s*** *и* ***t*** *как* ***σst*** *. а число путей проходящих через вершиной* ***v****:* ***σst(v)****. А также обозначим фракцию кратчайших путей между* ***s*** *и* ***t*** *как* **δst(v)** = **σst(v) ∕ σst**

“betweenness centrality” для вершины **v** определяется через:

Это свойство измеряет контроль вершины, имеющийся над общинами в сети и может использоваться для идентификации ключевых участников сети. Индексы высокой сосредоточенности показывают, что вершина может достигать других вершин на относительно кратчайших путях, или что вершина лежит на значительных долях кратчайших путях, соединяющих пары других вершин.

Детали Реализации

Паралелизм может эксплуатироватса в двух степях:

1. BFS/SSSP расчеты для каждой вершины может быть выпольнена конкурентно, а сумы “centrality” меняться атомично.

1. Сам процесс может параллелизироваться. Сам процесс визита соседей вершин можеть быть выполено параллельно.

Для многопоточных систем (таких как MTA-2) более оптимальным явлвется второй метод.

Имеющийся Результаты

Иследование показало, что в обшем случае возможно вычислить для всех вершин **n:**

****

**Eigenvector Centrality**

Концепцией позади идеи еigenvector сentrality является факт, что в сущности во многих случаях важность вершины завышается за счет связей с другими вершинами, которые сами по себе являются важными. Вместо того чтобы поднимать важность

вершины всего лишь на единицу за каждого имеющегося соседа, EC задает каждой вершине очко пропорциональное сумме очков, соответствующих его соседям.

Детали Реализации

В самом начале для каждой вершины i в качестве текущего значения xi представляющего из себя EC данной вершины задается значение 1. Это значение конечно же не идеально но засчет него мы можем получить другую более хорошую величину x'i , которая получается за счет суммы xj являющихся соседеми вершины i

Где Аij является элементом смежности матрицы. Повтаряя этот процесс для

получения лучших результатов после t шагов мы получим вектор EC x(t), определяемый данной формулой

Теперь мы можем записать х(0) как линейную комбинацию эйгенвекторов vi матрицы

смеженности:

при каком-то подходящем выборе констант ci. Тогда

где это эигенчисла А, а наибольший из них. Поскольку / < 1 для всех i ≠ 1, все термы в сумме кроме первого убывают по экспоненциально по мере роста t, и следовательно:

Другими словами, предельный вектор ЕС пропорционален головнуму эйгенвектору матрицы смежоности. Эквиуалентно мы можем сказать что ЕС х удовлетворяет уравнению

Это концепция ЕС, впервые предложенная Bonacich в 1987. Как и обещено ЕС вершины i пропорциональна сумме ЕС ее соседей i:

Что дает eigenvector centrality приятную особенность того, что она мошет быть большой либо по причине того, что у вершины великое мношество соседей, либо по потому, что ее соседи являются важными (либо оба фактора однавременно). По анологии индивидуум в социальнойсети может быть важным либо потому что он или она знает много людей (даже если они сами по себе не очень важны сами по себе), либо потому что знает много важных людей.

Последнее уравнение конечно же не фиксирует нормализацию EC, но во многих случаях это не имеет большого значения, поскольку нас в основном интересует у каких вершин это значение выше и у каких ниже, а не абсолютные значение меры.

Community Detection

Самой популярной и общепризнанной мерой качества для данной задачи является значение модулярности (modularity). Функционал был предложен Ньюманом и Гирваном в ходе разработки алгоритма кластеризации вершин графа. Модулярность — это скалярная величина из отрезка [−1, 1], которая количественно описывает неформальное определение структуры сообществ, данное выше:



где A — Матрица смежности графа, Aij — (i, j) элемент матрицы, di — степень i вершины графа, Ci — метка вершины (номер сообщества, к которому относится вершина), m — общее количество ребер в графе. δ (Ci, Cj) — дельта-функция: равна единице, если Ci = Cj, иначе нулю.

Задача поиска выделения сообществ в графе сводится к поиску таких Ci, которые максимизируют значение модулярности.

Формула имеет множество обобщений, например, для взвешенных графов, под Aij понимается вес ребра, соединяющего вершины i и j, а m =



**Методы выделения сообществ**

1. *Betweenness* — алгоритм на основе коэффициента “центральности по посредничеству” (Betweenness), определяемый как количество кратчайших путей между всеми парами вершин, проходящих через данное ребро.
2. *Fastgreedy* — метод, основанный на жадной оптимизации функции модуляр- ности.
3. *Multilevel* — метод, основанный на многоуровневой оптимизации функции мо- дулярности, c использованием эвристики, предложенной в статье.
4. *LabelPropogation* — метод, основанный на присвоении меток к каждой вершине. Каждый раз выбирается метка с максимальной встречаемостью среди смежных вершин.
5. *Walktrap* — подход, основанный на случайном блуждании. Использует идею о том, что короткие случайные блуждания не приводят к выходу из текущего сообщества.
6. *Infomap* — метод случайного блуждания, основанный на понятии информационных потоков в сетях, кодирования и сжатия информации.
7. *Eigenvector* — метод, основанный на собственных векторах матрицы модуляр- ности, которая получается из матрицы смежности.

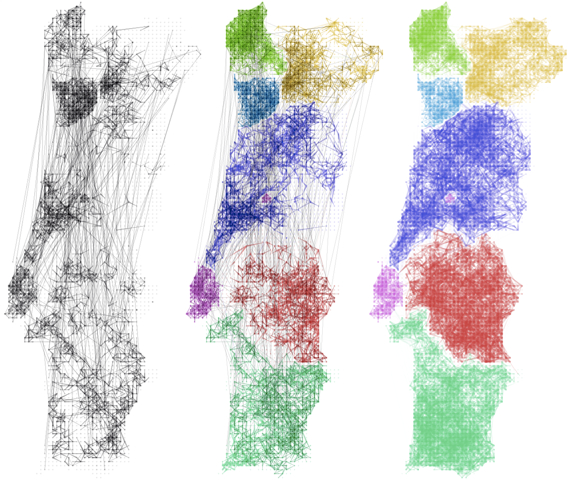
Существующие стратегии поиска выполняют одно из следующих действий, чтобы развивать начальнер разделение:

* *Слияние двух сообществ,*
* *Разделение сообщества на две части,*
* *Перемешение узлов между двумя различными сообществами*

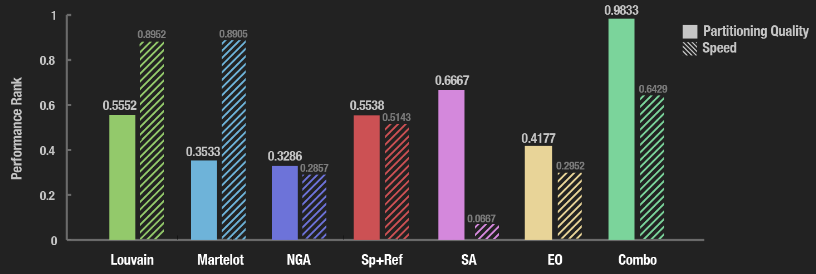
Предложенный алгоритм соединеняет все три действия. После выбора первоначального разделения, сделанного из одного сообщества, следующие шаги повторяются до тех пор, как результаты итераций приводит к желаемому результату.

1. Для каждого исходного сообщества вычесляется наилучшee возможнoe перераспределение всех исходных узлов в каждом целевом сообществе (либо существующих или новых).
2. Выполняется лучшее слияние /разделение/рекомбинация.

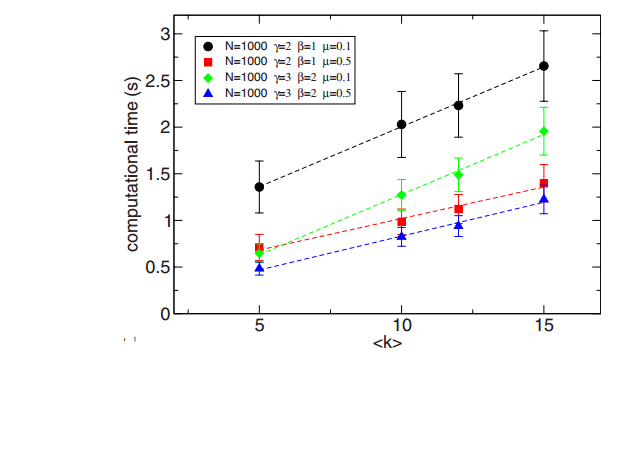
Поскольку предлагаемая методика сочетает в себе все три возможных типа шагов, этот алгоритм назван “Combo”.



[*гражданские разделительные линии в основных странах*]



*Средняя оценка нормированной производительности каждого алгоритма с точки зрения качества и скорости разбиения*



*Анализ работы алгоритма Combo.*

ГЛАВА 2: ИСПОЛЬЗУЕМЫЕ ИНСТРУМЕНТЫ И БИБЛИОТЕКИ

Одно из главных задая системы выбор методологий и инструментов, которые бы дали високую скорость выполнение задач. Для этого были использованы следующие инструменты:

 Параллелизм – Для обеспечения высокой производительности решено использовать кластерные вычисления на основе технологии MPI.

 Язык программирования - Одной из главных задач — это выбор языка программирования, так как производительность программы во многом зависит от использованного языка. Как кандидаты были расмотрены следующие языки:

o С++11 – C++ чрезвычайно мощный язык, содержащий средства создания эффективных программ практически любого назначения, от низкоуровневых утилит и драйверов до сложных программных комплексов самого различного назначения. В частности, поддерживаются различные стили и технологии программирования, включая традиционное директивное программирование, ООП, имеется возможность работы на низком уровне с памятью, адресами, портами.

 Возможность создания обобщённых алгоритмов для разных типов данных, их специализация и вычисления на этапе компиляции, используя шаблоны.

 Кроссплатформенность. Доступны компиляторы для большого количества платформ, на языке C++ разрабатывают программы для самых различных платформ и систем.

 Эффективность. Язык спроектирован так, чтобы дать программисту максимальный контроль над всеми аспектами структуры и порядка исполнения программы. Ни одна из языковых возможностей, приводящая к дополнительным накладным расходам, не является обязательной для использования — при необходимости язык позволяет

Для решения многих кодовых задач были использаны библиотеки boost. Boost является одной из самых известных и наиболее часто используемых программистами С++ библиотек, обеспечивающих высокую техническую эффективность. В данной работе были использованы следующие библиотеки:

 Boost.MPI - это библиотека реализующая интерфейс стандарта MPI абстрагируясь от конкретной имплементации. Самое главное приемущество заключается в том что с помощю Boost.MPI можно передавать стандартные контейнеры C++ через границы процессов на разных машинах.

 Boost.Serialization – Эта библиотека предназначена для сериализации и десериализации данных с помощью стандартных архиваторав Boost. Одной из важных особенностей является способность сериализации стандартных котейнеров C++. Также эта библиотека используется для переноса данных в Boost.MPI.

 Boost.Filesystem - предоставляет переносимые и удобные средства для манипулирования путей, файлов и каталогов.

 Boost.Program\_options - Библиотека program\_options позволяет разработчикам программ получать настройки программ, то есть (имя, значение) пар от пользователя, с помощью обычных методов, таких как командной строки и файла конфигурации.

 Boost.property\_tree – Библиотека property\_tree является удобным парсером для xml файлов

* Eigen library <http://eigen.tuxfamily.org/index.php?title=Main_Page>

При проектировании системы были использованы следующие паттерны проектирования:

 Паттерн Command (команда)

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

 Паттерн Mediator (посредник)

o Паттерн Mediator определяет объект, инкапсулирующий взаимодействие множества объектов. Mediator делает систему слабо связанной, избавляя объекты от необходимости ссылаться

друг на друга, что позволяет изменять взаимодействие между ними

независимо.

o Паттерн Mediator вводит посредника для развязывания множества взаимодействующих объектов.

o Заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".

 Паттерн Singleton (Одиночка)

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений

или драйвер дисплея. В таких случаях необходимо уметь создавать

единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа. Паттерн Singleton предоставляет такие возможности.

 Паттерн Abstract Factory (абстрактная фабрика)

Используйте паттерн Abstract Factory (абстрактная фабрика) если:

o Система должна оставаться независимой как от процесса создания

новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения new в коде приложения нежелательно (подробнее об этом в разделе Порождающие паттерны).

o Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

 Паттерн Factory Method (фабричный метод)

В системе часто требуется создавать объекты самых разных типов. Паттерн Factory Method (фабричный метод) может быть полезным в решении следующих задач:

o Система должна оставаться расширяемой путем добавления объектов новых типов. Непосредственное использование выражения new является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. Тогда такие операции как добавление в систему объектов новых типов или замена объектов одного типа на другой будут затруднительными (подробнее в разделе Порождающие паттерны). Паттерн Factory Method позволяет системе оставаться независимой как от самого процесса порождения объектов, так и от их типов.

o Заранее известно, когда нужно создавать объект, но неизвестен его тип.

 Паттерн Facade (фасад)

При проектировании сложных систем, зачастую применяется т.н. принцип декомпозиции, при котором сложная система разбивается на более мелкие и простые подсистемы. Причем, уровень декомпозиции (ее глубину) определяет исключительно проектировщик. Благодаря такому подходу, отдельные компоненты системы могу быть разработаны изолированно, затем интегрированы вместе. Однако возникает, очевидная на первый взгляд, проблема — высокая связность модулей системы. Это проявляется, в первую очередь, в большом объеме информации, которой модули обмениваются друг

с другом. К тому же, для подобной коммуникации одни модули должны обладать достаточной информацией о природе других модулей.

 Паттерн External Polymorphizm(Holder)

КОНФИГУРАЦИЯ СИСТЕМЫ

Одной из главных задач разработки системы — это разработка командного языка для работы с разными задачами. Но есть проблема во время использования MPI, так как есть задачи, которые должны запускаться на кластерах, но и есть такие, что достаточно запустить одним процессом. С другой стороны, нужно было, чтобы команды, которые тратят много времени на расчетах (к примеру, траектория четырехугольников) запускались в background режиме, чтобы была возможность продолжать работу с командной строкой – запускать новые задачи во время выполнения предыдущих. Из этого выходит, что компилировать все задачи и интерфейс командной строки вместе в один бинарный файл нецелесообразно. Для этого было решено распределить командный интерфейс и отдельные задачи в разные бинарные файлы, команды в свою очередь должны запускать соответствующий бинарник (в режиме MPI или одного процесса) в background режиме.

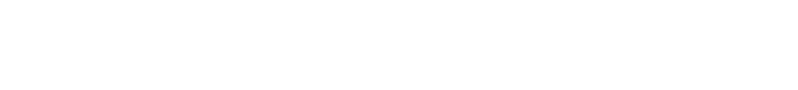
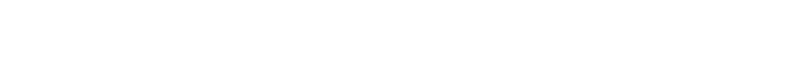
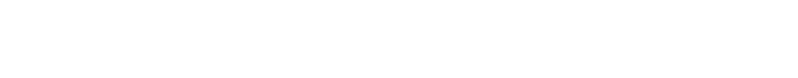
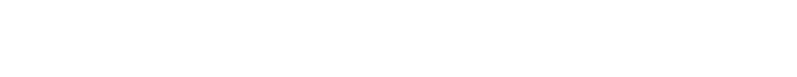
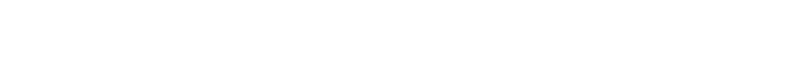
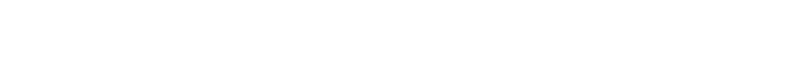
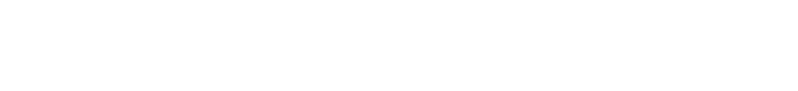
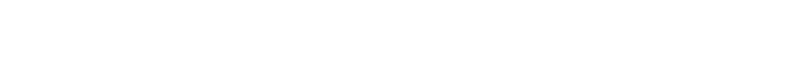
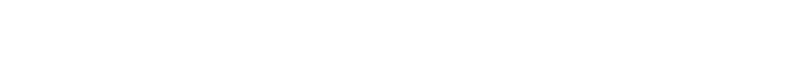
**Структура конфигурационного файла**

Конфигарационный файл является ондим из важнейшых узлов системы. Через конфиг фаил юзер может задать все основнвые параметры изходя от которых производятся вычисления и запускаются соотвествующие бинарники подсистем.

|  |
| --- |
|  |
|  | <config> |
| Графы для которых будут производится вычисления-- | <graphs> |
|  | <graph-path></graph-path> |
| --Графы для которых будут производится вычисления | </graphs> |
| Граф, который будет разделен на сообщества | <community-graph></community-graph> |
| Свойства, для которых будет работать система-- | <properties> |
|  | <property></property> |
|  | <property></property> |
|  | <property></property> |
|  | <property></property> |
| --Свойства для которых будет работать система | </properties> |
| Вычислить распределения | <calculate-distribution></calculate-distribution> |
| Вычислить средние значения | <calcuate-average></calcuate-average> |
| Максимальное каличество mpi процессов | <max-parallel-proc-count></max-parallel-proc-count> |
|  | </config> |

**Конфигурация корневой директории**

Основные компоненты системы. Все компоненты содержаться в директориях корневой директории root, в соответствующих подкаталогах .



|  |
| --- |
| Корневая директория |
| Подсистема командного интерфейса |
| Новый тип графов. В основном используется как библиотека в других подсистемах |
| Генератор графов нового типа |
| Конвертор графов |
| Подсистема, отвечающая за статистическое исследования траектории |
| Подсистема отвечающая за разбиения графа на сообшеств |
| Подсистема содержащая интерфейс блочно-  иерархического графа. |
| Конечное местоположение бинарников, ресурсов и результатов иследования. |
| Makefile корневой директории. |

root

core

graph

graph\_generator

graph\_converter

alternate\_property\_computer

community detector

hier\_graph

Install

Makefile



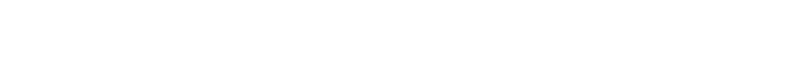
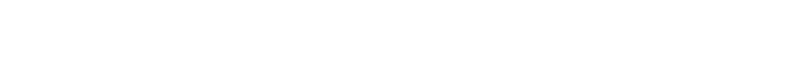
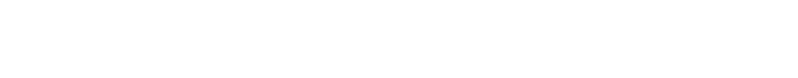
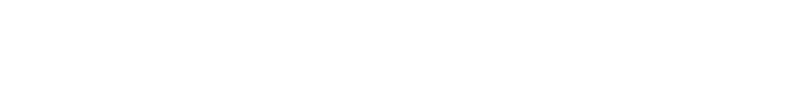
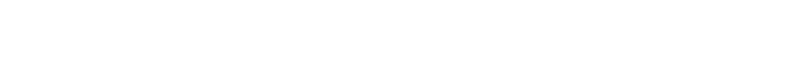
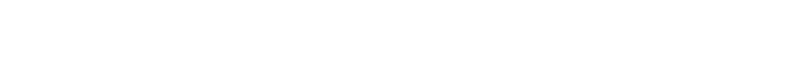
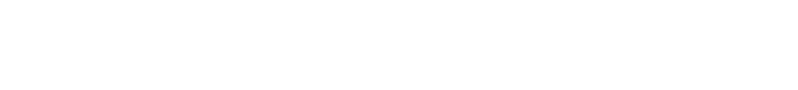
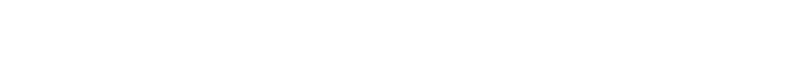
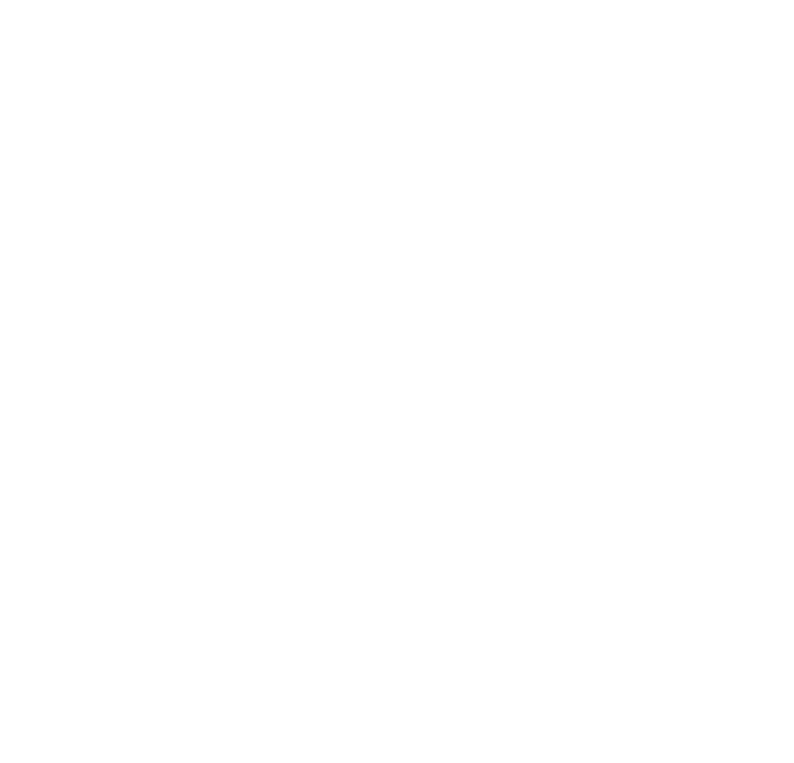
Makefile



Конфигурация и автоматизация системы выполнена с помощью технологии GNU/make. Makefile корневой директории ответственен за вызов команды “make” в поддиректориях корневой директории, которые записаны в переменной “SUBDIRS”, то есть первоначальный шаг добавление новой подсистемы — это добавление имени новой директории в выше упомянутую переменную перед именем “install” так как конфигурация системы требует, чтобы вызов “make” в директории “install” вызвалось в последную очередь. Возникает вопрос: Зачем регистрировать подсистемы ручную, а не автоматически. Ответ прост: чтобы дать возможность включения подсистем, не зависимых от корневой (к примеру, для тестирования).

**Конфигурация директории install**

Ниже приведена таблица директории install.



install

bin

global\_results

boost\_graphs

standart\_graphs

ng\_graphs

mus

Makefile

|  |
| --- |
| Директория install |
| Все бинарники которые должны использоваться командным интерфейсом, перемешаются здесь |
| Чтобы обеспечить аккуратность системы все результаты должны записываются здесь |
| Граф-файлы типа boost.Graph |
| Граф-файлы стандартного типа |
| Граф-файлы нового тип |
| Фалы с значениями mu, нужные для расчетов траектории. |
| Makefile директории install |

Makefile директории “install” ответственен за копирования бинарников из поддиректорий корневой директории, которые записаны в переменной “SUBDIRS”,

то есть чтобы командный интерфейс смог работать с бинарником новой подсистемы нужно добавить имя новой директории в выше упомянутую переменную. Регистрация подсистемы выполняется ручную, а не автоматически по аналогичной причине переведенной заранее (в этом случае инженеру дается возможность не включать новую систему в командный интерфейс, к примеру подсистема “graph” не включена так как она предназначена для использования как библиотеку в других подсистемах).

**Конфигурация подсистем**

Ниже приведена общая таблица подсистем.

subsystem

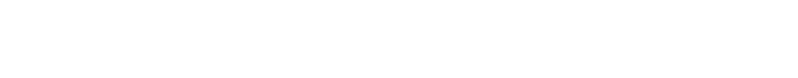
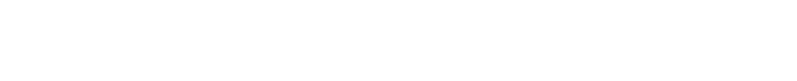
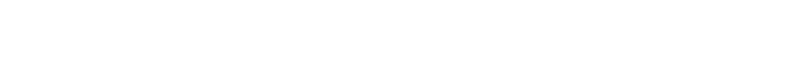
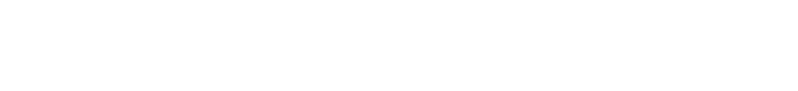
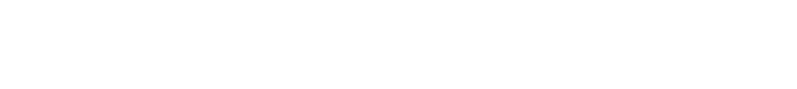
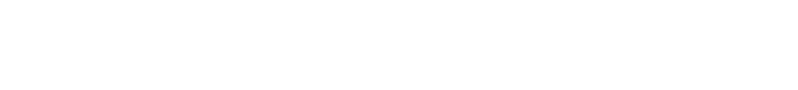
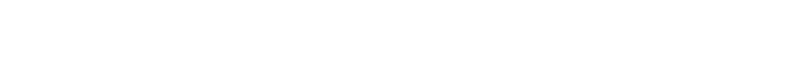
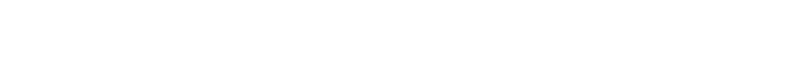
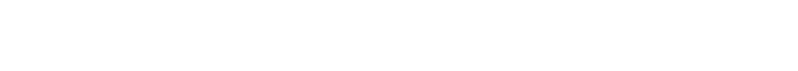
Module of subsytem

|  |
| --- |
| Подсистема системы |
| Модуль подсистемы |
| Makefile модуля подсистемы |
| Директория где перемещаются компиляционные файлы (.o файлы) |
| Директория где перемещаются все компиляционные файлы и генерируется бинарник. |
| Makefile директории lib |
| Конфигурационный файл doxygen для генерации документации |
| Общая структура Makefile-ов модулей которая должна бить импортирована в Makefile-и модулей |
| Makefile подсистемы. |

Makefile

objs

lib



Makefile

Doxyfile

common.mk

Makefile

Makefile подсистемы ответственен за вызов команды “make” в модулях подсистемы, которые записаны в переменной “SUBDIRS”, то есть включить модуль в подсистему нужно добавить имя модуля в выше упомянутую переменную перед именем “lib” так как конфигурация системы требует, чтобы вызов “make” в директории “lib” вызвалось в подледную очередь.

Makefile модуля генерирует компиляционные файлы и транспортирует их в директорию “objs”. Инженер, который работает над подсистемой должен знать что проблемы линковки не будут выявляться если вызвать “make” только в модулях. Заметем что Makefile модуля должен обязательно импортировать “common.mk”

Makefile с родительской директории, так как она содержит настройки компилятора подсистемы.

common.mk Makefile находится в каждой подсистеме, а не в корневой единожды, потому что нужно дать возможность подсистем работать с разными библиотеками и компиляторами.

Makefile директории “lib” ответственен за перенос компиляционных файлов в директорию “lib” и генерацию бинарника подсистемы. Отметем что именно отсюда Makefile директории “install” берет бинарник.

АРХИТЕКТУРА СИСТЕМЫ

1. Основные требования к системе:
   1. Возможности параллелизации на вычислительных кластерах.
   2. Разбивка подсистем на по возможности независимые модули
   3. Высокая производительность
   4. Гибкость
   5. Способность к легкому и удобному расширению
   6. Сильная независимость программных модулей
   7. Читабельность (придерживание к coding-style и doxygen документация)
2. Преимущества каторыми должна отличатся система
   1. Возможность параллельного вычесление данных свойств сразу для ряда отличных друг от друга сетей
   2. Разбиение сетей на наиболее связанные компоненты и далнейьшая работа и анализ произвадимые с полученными подсетями
   3. Широкий выбор свойств для вычесления
   4. Возможность усреднения полученных результатов
   5. Интуитивный и удобный интерфейс
3. Разработка конфигурационного файла
   1. Конфигурация анализируемыx графов
   2. Конфигурация числа процессов (а вдальнейшем и кластеров) на которых должны производится вычесления
4. Добавлеие новых типов случаиных сетей

В данной главе будут рассмотрены архитектуры подсистем и их особенности. Имплементационные особенности. Способы расширения. Для больших подсистем будут приведены функциональные схемы работы.

АРХИТЕКТУРА ПОДСИСТЕМЫ “ALTERNATE PROPERTY COMPUTER”

Подсистема “alternate\_property\_computer” отвечает за вычисление траекторий того или иного свойства во время развития сети. Архитектура проекта составлена так чтобы возможно было с минимальными изменениями добавлять новых менеджеров процессов, новых свойств случайных сетей и новых рандомизаторов

процесса развитии сети. Ниже представлена таблица модулей подсистемы.



Ниже подчеркнуты особенности подсистемы.

 Возможность парализации на вычислительных кластерах

 Возможность парализации в многопотчной среде

 Возможность управления работы в однопроцессном режиме

 Возможность добавления новых менеджеров процессов

 Возможность добавления новых Характеристик для получения траекторий

 Использования библиотеки Boost.program\_options, которая облегчает добавления новых аргументов программы.

 Использования библиотеки Boost.MPI (Facade над технологией MPI) которая дает возможность транспортировки объектов, для которых реализован интерфейс Boost.Serialization (Boost.Graph, контейнери stl)

 Использования “Mediator” шаблона проектирования, которая облегчает

управления подсистемой.

 Использования “ Abstract Factory ” шаблона проектирования, которая облегчает добавление новых рандомизаторов и Характеристик.

АРХИТЕКТУРА ПОДСИСТЕМЫ “GRAPH”

Подсистема “graph” предоставляет библиотеку для работы с гафоми. Изначально для работы с графоми была выбрана библиотека Boost.Graph, но во время проектирования были выявлены проблемы и неисправности этой библиотеки, которые блокировали дальнейшее проектирования:

 Копирующий конструктор не делал deep-copy, из-за этого изначальный граф портился

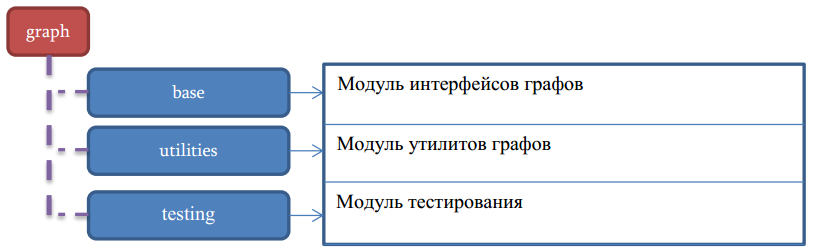
 Производительность некоторых очень важных методов была не удовлетворяющая поставленной нормы

 Файл хранящий граф этого типа очень велик даже для небольших графов

 Для больших графов процесс использовал очень много памяти RAM

Из-за этого было решено создать свой собственный тип для хранения графов который бы удовлетворял поставленным условиям. Ниже представлена таблица

модулей подсистемы.



Ниже подчеркнуты особенности подсистемы.

 Автоматическим или механическим способом выбора внутреннего контейнера информации о графе

 Возможность добавление нового контейнера

 Независимость внешнего интерфейса от выбранного контейнера

 Использование динамических битовых последовательностей в некоторых из внутренних контейнеров

 Возможность синхронизации графов, не смотря на разность в использование внутренних контейнеров

 В зависимости от условий исследования выбор вылечены внутреннего типа вершин

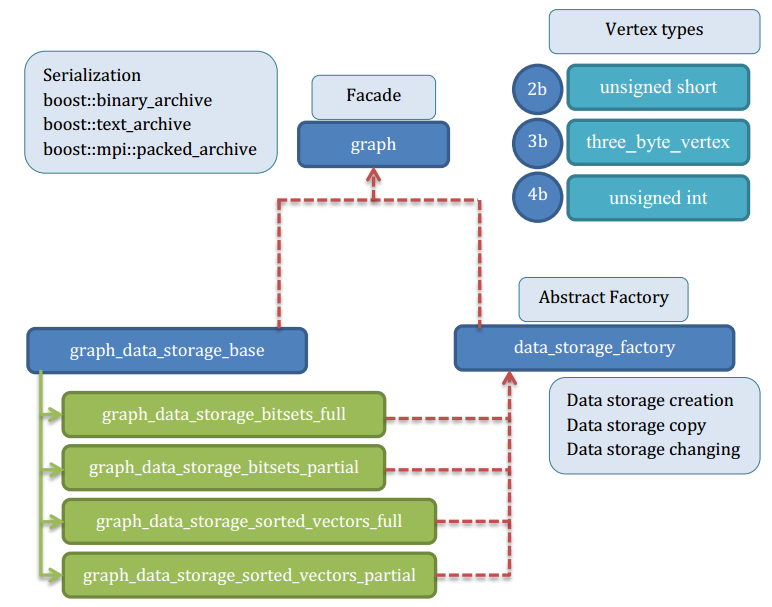
 С помощью атрибутов компилятора был сконструирован новый тип вершин с венечной 3 байт.

 Производительность сравнимая с производительностью библиотеки

Boost.Graph

 Экономия файловой и RAM памети по сравнению с Boost.Graph, что в свою очередь улутшает скорость транспортировки в среде MPI

 Код подсистемы написан с большой строгостью к соблюдении стиля программирования, которая повышает возможность к расширению и читабельности кода.



Выше представлена диаграмма классов подсистемы “graph”. Ниже переведен разбор их назначений:

 graph – Фасад над внутренним контейнером

 graph\_data\_storage\_base – Базовый класс внутренних контейнеров

 graph\_data\_storage\_bitsets\_full – Внутренний контейнер, хоронящий полные битовые последовательности.

 graph\_data\_storage\_bitsets\_partial - внутренний контейнер, хоронящий частичные битовые последовательности

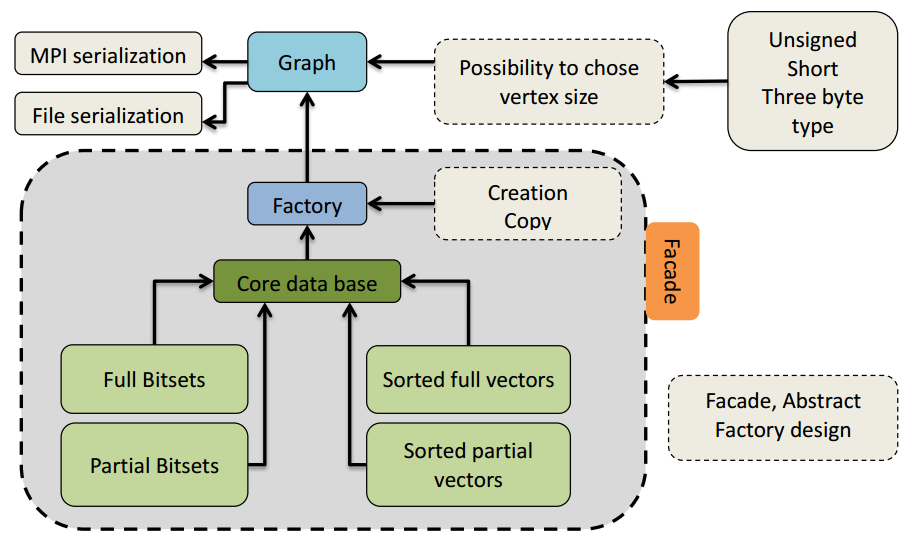
 graph\_data\_storage\_sorted\_vectors\_full - внутренний контейнер, хоронящий отсортированные полные векторы

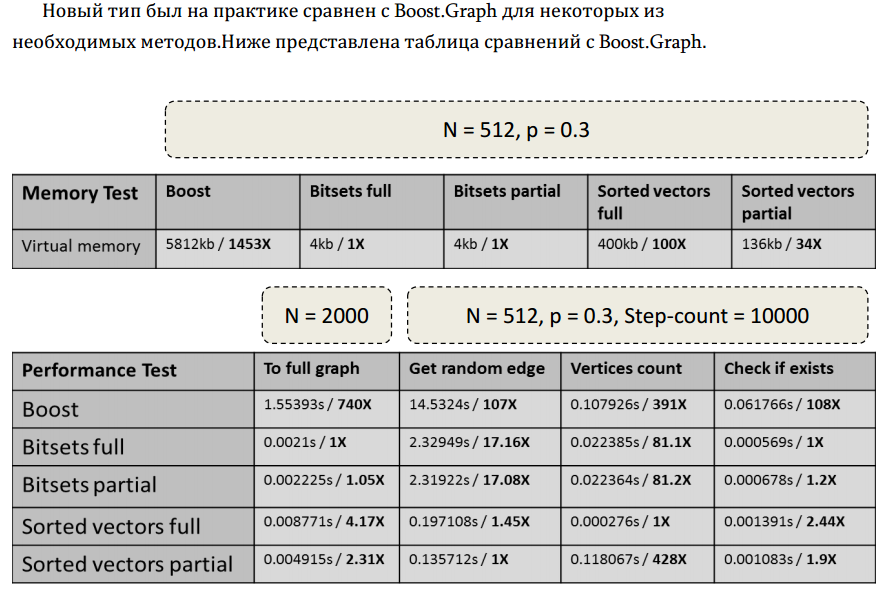
 graph\_data\_storage\_sorted\_vectors\_partial - внутренний контейнер, хоронящий отсортированные частичные векторы

 data\_storage\_factory – Фабрика над внутренними контейнерами

 three\_byte\_vertex – тип вершин с памятью 3 байт

Ниже представлена логическая функциональная схема типа.





АРХИТЕКТУРА ПОДСИСТЕМЫ “UNIT TESTS”

Подсистема “unit\_tests” предоставляет мошный инструмент для тестирования и предохранениа фунцкианальности системы.

* Архитектура подсистемы нет отличаетса от остальных.
* В качестве библиотеки был выбран google-tests (<https://github.com/google/googletest>), который являеста более продвинутым и изяшншым из всех восможных.

ГЛАВА 6: Рукаводство для пользователя

### <https://github.com/NarekAt/DiplomaLocal/blob/master/README.md>

### Installation

##### Steps for configuration of development environment:

* GCC version must be 4.6 or above (if it is possible install 4.8)
* Download sources, extract somewhere (let's assume folder name is extracted\_dir)<http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4.tar.gz>
* Change current directory to extracted\_dir
* ./configure --enable-shared --enable-sharedlibs=gcc
* make && sudo make install (this will install mpich3 to /usr/local/lib, /usr/local/bin etc.), note: All versions of mpi must be uninstalled before installation with this instruction.
* Download boost source (do not install with apt-get, it is needed to compile it using mpi settings)<http://sourceforge.net/projects/boost/files/boost/1.53.0/>. Then set BOOST\_LIBS environment variable to point on boost installation directory.
* Read <http://www.boost.org/doc/libs/1_53_0/doc/html/mpi/getting_started.html#mpi.config>
* When building boost, edit user-config.jam file and add following line: using mpi : /usr/local/bin/mpic++ ;(don't miss up white spaces around ':' and ';').
* Make sure that all previous versions of mpich3 are uninstalled.
* Proceed to building and installing of boost.

##### Steps for external libs installation

* From central place, cd install/compile directory and run load\_external\_libs.sh script. This will automatically configure OGDF and libxl packages.
* The installation also requires following packages to be installed: Eigen (<https://eigen.tuxfamily.org/dox/GettingStarted.html>), igraph (<http://igraph.org/c/>).

### Building system

* From central place, cd subsystem for which you want to (re)-generate executable and run make or make debug, for release and debug builds correspondingly. The location of generated executable is $(pwd)/lib/ directory.

### Testing system (Unit tests)

./unit\_tests/run\_tests.sh – For running all existing unit\_tests

### Practical Usage

##### Graph Generation

graph\_generator.exe –M [random\_graph\_model] -N [number\_of\_vertices] -p [probability] –E [number\_of\_edges] –f [output\_file] –t [storage\_core\_type]

Note: number\_of\_edges and probability are mutually exclusive properties 1st should be specified only in case of Block-Hierarchical model and 2nd for Erdos-Reyni model.

##### Alternate Property Computer and Community Detection

./scripts/apc\_runner.sh [configuration file]

###### Configuration input file prototype

<?xml version="1.0"?>

<config>

<graphs>

<graph-path></graph-path>

</graphs>

<community-graph></community-graph>

<properties>

<property></property>

<property></property>

<property></property>

<property></property>

</properties>

<calculate-distribution></calculate-distribution>

<calcuate-average></calcuate-average>

<max-parallel-proc-count></max-parallel-proc-count>

</config>

##### Property Average Calculation

./scripts/avg\_calc.sh

Calculates and stores averages of properties given in config.xml under avg\_results folder. To enable this feature user must set calcuate-average option in configuration file as true(or 1)

##### Graph Converter

graph\_converter.exe [flag] [first\_file\_name] [second\_file\_name]

Note: [flag] is 'b', then [first\_file\_name] must be boost graph file name, [second\_file\_name] must be standard graph file name.If [flag] is 's', then [first\_file\_name] must be standard graph file name, [second\_file\_name] must be boost graph file name.

ИСТОЧНИКИ И ЛИТЕРАТУРА

1. Albert R, Barab´asi A-L Statistical mechanics of complex networks. Rev. Mod. Phys. 74: 47–97. (2002)
2. M. Newman - Networks: An Introduction (2010)
3. <http://www.cc.gatech.edu/~bader/papers/GraphCT-TPDS2013.pdf> (2012)
4. 3. *В.А. Аветисов, А.Х. Бикулов, О. А. Васильев, С. К. Нечаев, А. В Чертович.*  О некоторых физических приложений случайных иерархических матриц. ЖЭТФ 136(3) 566 (2009); *V. A. Avetisov, A. Kh. Bikulov, O. A. Vasilyev, S. K. Nechev, and A. V. Chertovich,* JETP, 109(3) 485 (2009).
5. Fast unfolding of communities in large networks: <http://arxiv.org/abs/0803.0476> (2013)
6. Introduction to Algorithms, 3rd Edition: <http://www.amazon.com/Introduction-Algorithms-3rd-MIT-Press/dp/0262033844>