МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)

ФАКУЛЬТЕТ ИННОВАЦИЙ И ВЫСОКИХ ТЕХНОЛОГИЙ
КАФЕДРА АНАЛИЗ ДАННЫХ

(Специализация «Прикладные информационные технологии
в управлении и бизнесе»)

ОТЫСКАНИЕ ОПТИМАЛЬНЫХ ПАРАМЕТРОВ В МОДЕЛЯХ СЛУЧАЙНЫХ ВЕБ-ГРАФОВ

Магистерская диссертация
студента 793 группы
Жернова Павла Владимировича

Научный руководитель
Райгородский А.М., д.ф.-м.н.

г. Москва
2013

Оглавление

Глава 1

Введение

Теория графов играет огромную роль как в фундаментальной, так и в прикладной математике. Граф — это мощный инструмент, позволяющий моделировать, описывать и упорядочивать самые различные реальные объекты. В данной магистерской диссертации будет рассмотрено активно развивающееся направление теории графов — случайные графы. Эти графы изучаются с вероятностной точки зрения и позволяют описывать нестатические среды и сети, социальные, биологические, транспортные. Описание вышеперечисленных сред необходимо для решения нескольких задач: поиска зависимостей, кластеризации и др.

Нас будет интересовать применение случайных графов к всемирной сети Интернет. Все множество веб-страниц всемирной паутины можно представить в виде вершин графа, которые соединены между собой в том случае, если на одном сайте есть ссылка на другой сайт или наоборот. Полученный таким образом граф называется веб-графом. Веб-графы позволяют формализовать и структурировать Интернет, представив множество сайтов в удобном виде. Такое представление используется в ряде актуальных прикладных задач: поиск информации, вычисление индекса цитируемости, отыскание статей схожей тематики и многих других.

Глава 2

Модели случайных веб-графов

Модели случайных веб-графов [1] позволяют генерировать WWW-подобные графы, которые значительно меньше и проще, чем реальные WWW-графы, однако сохраняют определенные ключевые свойства структуры ребер веба. Такие искусственные графы можно рассматривать как экспериментальную платформу для получения новых подходов к поиску, индексации и т.д. Вершины веб-графа соответствуют веб-страницам, а ребра – гиперссылкам между ними. Веб-графы довольно активно изучались на предмет различных числовых характеристик таких как распределение, диаметр, число связных компонент, макроскопическая структура. Ниже приведем различные модели, призванные описывать реальные веб-графы.

Один из возможных теоретических подходов к модели веб-графа — это математическая концепция случайного графа. Суть этого подхода заключается в том, что веб-граф развивается стохастически. Было предпринято множество попыток смоделировать граф гиперссылок интернета как случайный граф. Наиболее простой и исторически первой является модель

Эрдеша–Реньи.

2.1 Модель Эрдеша–Реньи

Пусть $V_n = \{1, \ldots, n\}$ — множество вершин графа. Именно на них мы и будем строить наш случайный граф. Соединим любые две вершины a и b ребром с вероятностью $p \in [0, 1]$ независимо от всех остальных пар вершин. Другими словами, ребра в графе будут появляться в соответствии со схемой Бернулли, в которой вероятность успеха p и C_n^2 испытаний (нас не интересуют кратные ребра, петли; граф неориентирован). Пусть E — случайное множество ребер, полученное в результате реализации такой схемы. Граф $G = (V_n, E)$ и есть случайный граф в модели Эрдеша–Реньи.

2.2 Модели предпочтительного присоединения

В 90-е годы XX века в своих работах Барабаши и Альберт описали некоторые статистики интернета — веб-графа, вершинами которого являются страницы в интернете, а ребрами — гиперссылки между ними. На самом деле, похожую структуру имеют также большинство других реальных сетей: социальные, биологические, транспортные.

Основные результаты исследования Барабаши и Альберта состоят в следующем:

1. Веб-граф — это «разреженный» граф. У него на n вершинах всего mn ребер, где $m \in \mathbb{Z}$ — некоторая константа. Для сравнения, у полного

графа на n вершинах $C_n^2 \sim n^2$ ребер.

2. Диаметр веб-графа очень мал (5-7, результат 1999 года). Это хорошо известное свойство любой социальной сети, которое принято называть «мир тесен». Например, говорят, что любые 2 человека в мире «знакомы через 5-6 рукопожатий». В интернете это свойство заключается в том, что кликая 5-7 раз по ссылкам можно перейти между любыми двумя страницами. (Если говорить более точно, то в интернете есть только что появившиеся сайты, которые могут быть не связаны с остальными сайтами. Поэтому правильнее сказать, что в интернете есть огромная компонента, диаметр которой мал). Итак, веб-граф обладает интересным свойством — он разрежен, но при этом «тесен».

3. Для веб-графа характерен степенной закон распределения степеней вершин. То есть, вероятность того, что вершина веб-графа имеет степень d, равна $cd^{-\xi}$, где $\xi = 2.1$. Интересно, что этот закон характерен для всех реальных сетей, но у каждой из них своя ξ.

Таким образом, описанная выше модель случайного графа Эрдеша–Реньи плохо описывает реальные веб-графы, поскольку графы, полученные в этой модели, не имеют степенного закона распределения степеней вершин.

Барабаши и Альберт предложили концепцию предпочтительного присоединения: граф строится с помощью случайного процесса, на каждом шаге которого добавляется новая вершина и фиксированное число ребер из новой вершины в уже существующие. При этом вершины с большей степенью приобретают рёбра с большей вероятностью, которая линейно зависит от их степени.

2.3 Модель Боллобаша–Риордана

Общая идея предпочтительного присоединения строго математически формулируется в модели Боллобаша–Риордана. Конструируется набор графов (марковская цепь) G_m^n, $n = 1, 2, \ldots,$ с n вершинами и mn ребрами, где $m \in \mathbb{Z}$ — целое число. Сначала рассмотрим случай $m = 1$. Пусть граф G_1^1 — граф, состоящий из одной вершины и одного ребра (петля). Граф G_1^t получается из графа G_1^{t-1} добавлением вершины t и ребра из вершины t в вершину i, где i выбирается из существующих в графе вершин случайно, согласно следующему распределению вероятностей:

$$P(i = s) = \begin{cases} d_{G_1^{t-1}}(s)/(2t-1), & \text{если } 1 \leqslant s \leqslant t-1, \\ 1/(2t-1), & \text{если } s = t, \end{cases}$$

где $d_{G_1^{t-1}(s)}$ — степень вершины s в графе G_1^t.

Заметим, что распределение вероятностей задано корректно, поскольку:

$$\sum_{i=1}^{t-1} \frac{d(i)}{2t-1} + \frac{1}{2t-1} = \frac{2t-2}{2t-1} + \frac{1}{2t-1} = 1$$

Случайный граф G_1^n построен и он удовлетворяет принципу предпочтительного присоединения. Далее, граф G_m^n строится из графа G_1^{mn} объединением вершин $1, \ldots, m$ в вершину 1 нового графа, объединением вершин $m+1, \ldots, 2m$ в вершину 2 нового графа и так далее. Заметим, что можно аналогичным образом строить ориентированные графы: ребро между вершинами i и j идет из i в j, если $i > j$.

2.4 Модель Боллобаша–Риордана. Статическая модификация

Существует также статическая модификация этой же модели. Статическая она потому, что в ней статическое описание случайности. Итак, зафиксируем на оси абсцисс на плоскости $2n$ точек: $1, \ldots, 2n$. Все точки разобьем на пары, каждую пару соединим дугой, которая лежит в верхней полуплоскости. Получится объект, который назовем *линейной хордовой диаграммой* (*lineared chord diagram или, сокращенно, LCD*). Заметим, что на $2n$ точках можно построить

$$l_n = \frac{(2n)!}{2^n n!}$$

различных LCD. По каждой LCD построим граф на n вершинах и с n ребрами. Алгоритм следующий: двигаемся по оси абсцисс слева направо до тех пор, пока не обнаруживаем правый конец любой дуги. Пусть этот конец имеет номер k_1. Тогда множество $1, \ldots, k_1$ делаем первой вершиной графа. Продолжаем двигаться от $k_1 + 1$ направо до следующего правого конца любой дуги k_2. Второй вершиной графа делаем набор $k_1 + 1, \ldots, k_2$. Далее, аналогично. Всего правых концов n, поэтому мы получим граф на n вершинах. Ребра в графе будем проводить по следующему правилу: две вершины соединяем ребром в том случае, если между соответствующими множествами точек есть дуга, при этом ребра ориентируются справа налево.

Далее, если считать LCD случайной, то есть полагать, что вероятность каждой LCD равна $1/l_n$, то возникают случайные графы. Можно доказать, что в определенном смысле такие графы очень похожи на G_1^n. Графы на n вершинах и с mn ребрами получаем так же, как и ранее.

2.5 Модель Боллобаша–Риордана. Результаты

Модель Боллобаша–Риордана хорошо отражает эмпирические свойства различных реальных графов. Во-первых, справедлива

Теорема 1 *Для любого $k \geqslant 2$ и любого $\varepsilon > 0$*

$$P\left((1-\varepsilon)\frac{\log n}{\log\log n} \leqslant diamG_k^n \leqslant (1+\varepsilon)\frac{\log n}{\log\log n}\right) \to 1, n \to \infty$$

Это означает, что диаметр графа плотно сконцентрирован (по вероятности) около величины $\log n / \log\log n$, что согласуется с результатом 5-6 для 1999 года, потому что в интернете в 1999 году было 10^7 вершин, значит

$$\frac{\log 10^7}{\log\log 10^7} = \frac{7\log 10}{\log 7 + \log\log 10} \approx 6.$$

Во-вторых, в 2001 году была доказана

Теорема 2 *Для любого $k \geqslant 1$ и любого $d \leqslant n^{\left(\frac{1}{15}\right)}$*

$$M\left(\frac{|i = 1, \ldots, n : deg_{G_k^n} i = d|}{n}\right) \sim \frac{2k(k+1)}{(d+k+1)(d+k+2)(d+k+3)}.$$

Поскольку k — константа, выражение в правой части имеет вид $const/d^3$, что и представляет из себя степенной закон. У этой теоремы, однако, есть и неприятные моменты. Во-первых, из-за ограничения $d < n^{(1/15)}$ теорема не годится для практического применения. Во-вторых, степень d в степенном законе в этой теореме равна 3, что расходится с реальными графами, для которых $\xi_{www} = 2.1$. Это означает, что хоть модель Боллобаша–Риордана и отражает некоторые свойства интернета, она должна быть видоизменена, чтобы лучше соответствовать реальности.

2.6 Модель Бакли–Остхуса

Возможный подход к такому видоизменению — это модель, независимо предложенная двумя группами исследователей. Они предложили расширить модель с помощью параметра, называемого *начальная аттрактивность вершины*. Это положительная константа, которая не зависит от степени. Позже Бакли и Остхус предложили явную конструкцию данной модели. Распределение степеней вершин в модели Бакли–Остхуса также подчиняется степенному закону, однако теперь варьируя значение параметра a в определении модели можно изменять значение ξ результирующего графа [2].

Более строго, модель генерирует набор графов $H_{a,m}^n, n = 1, 2, \dots,$ с n вершинами и mn ребрами, где $m \in \mathbb{Z}$ — фиксированное число. Определение $H_{a,1}^n$ повторяет определение G_1^n с одним отличием, заключающимся в том, что вероятность нового ребра, добавляемого в $H_{a,1}^n$ равна

$$P(i = s) = \begin{cases} \frac{d_{H_{a,1}^{t-1}}(s)+a-1}{(a+1)t-1}, & \text{если } 1 \leqslant s \leqslant t-1, \\ \frac{a}{(a+1)t-1}, & \text{если } s = t, \end{cases}$$

Граф $H_{a,m}^n$ получается из графа $H_{a,1}^{mn}$ так же, как и G_m^n получается из G_1^{mn}. Заметим, что при $a = 1$ мы получаем изначальную модель Боллобаша–Риордана G_m^n. Для целых a Бакли и Остхус доказали, что распределение степеней вершин случайного графа в модели соответствует степенному закону с $\xi = 2 + a$.

2.7 Модель Боллобаша–Боргса–Риордана – Чайеса

В данной модели [3] строится ориентированный граф итеративно, на каждом шаге добавляется одно ребро. На каждом шаге также может быть добавлена одна вершина. Для простоты будем считать, что в графе могут присутствовать множественные ребра и петли.

Более строго, пусть $\alpha, \beta, \gamma, \delta_{in}$ и δ_{out} — неотрицательные действительные числа такие, что $\alpha + \beta + \gamma = 1$. Пусть G_0 — фиксированный начальный ориентированный граф, например, одна вершина без ребер, и пусть t_0 — это число ребер в графе G_0. (В зависимости от параметров может понадобиться положить $t_0 \geqslant 1$, чтобы на первых шагах процесс имел смысл). Положим $G(t_0) = G_0$, то есть в момент времени t граф $G(t)$ имеет ровно t ребер и случайное число $n(t)$ вершин.

Для упрощения описания модели, условимся под фразой «*выбрать* вершину v графа $G(t)$ *в соответствии* с $d_{out} + \delta_{out}$» понимать, что вершина v выбирается таким образом, что $P(v = v_i)$ пропорциональна $d_{out}(v_i) + \delta_{out}$, то есть таким образом, что $P(v = v_i) = (d_{out}(v_i) + \delta_{out})/(t + \delta_{out} n(t))$. Аналогично, под фразой «*выбрать v в соответствии с $d_{in} + \delta_{in}$*» будем понимать, что вершина v выбирается таким образом, что $P(v = v_i) = (d_{in}(v_i) + \delta_{in})/(t + \delta_{in} n(t))$. Здесь $d_{out}(v_i)$ и $d_{in}(v_i)$ — исходящая и входящая степени вершины v_i в графе $G(t)$.

Для $t \geqslant t_0$ граф $G(t+1)$ строится из графа $G(t)$ по следующим правилам:

1. С вероятностью α добавляется новая вершина v вместе с ребром из v в существующую вершину w, где w выбирается в соответствии с

$d_{in} + \delta_{in}$.

2. С вероятностью β добавляется ребро из существующей вершины v в существующую вершину w, где v и w выбираются независимо, v в соответствии с $d_{out} + \delta_{out}$, а w в соответствии с $d_{in} + \delta_{in}$.

3. С вероятностью γ добавляется новая вершина w и ребро из существующей вершины v в вершину w, где v выбирается в соответствии с $d_{out} + \delta_{out}$.

Понятно, что вероятности α, β и γ должны в сумме давать единицу. Чтобы граф не был тривиальным, необходимо также положить $\alpha + \gamma > 0$. Заметим также, что для веб-графа естественно взять $\delta_{out} = 0$, потому что вершины, добавляемые в третьем случае соответствуют веб-страницам, которые просто предоставляют некий контент. Такие страницы никогда не изменяются, они «рождаются» без исходящих ссылок и сохраняют это свойство. Вершины, добавляемые в первом случае соответствуют обычным страницам, ссылки на которые могут быть добавлены позже. Также, чисто математически кажется естественным положить и $\delta_{in} = 0$ наряду с $\delta_{out} = 0$, однако это приводит к модели, в которой каждая страница не из G_0 не будет иметь либо входящих ссылок, либо исходящих, что довольно нереалистично и неинтересно! Ненулевое значение δ_{in} говорит о том, что вершина не является частью веба до тех пор, пока на нее не появятся ссылки, обычно с одного из крупных поисковых сервисов. Эти ссылки с поисковых сервисов естественно рассматривать отдельно от графа, поскольку они имеют другую природу. По той же причине δ_{in} не обязательно должно быть целым числом. Параметр δ_{out} все же включается в модель для симметрии и для большей общности.

Модель допускает наличие в графе петель и кратных ребер, нет оснований для исключения их из графа. Более того, их число оказывается небольшим, поэтому они незначительно влияют на численные эксперименты.

2.8 Модель копирования

Эта модель возникла практически в одно время с моделью Барабаши-Альберта. Ее авторами являются Р. Кумар, П. Рагхаван, С. Раджагопалан, Д. Сивакумар, А. Томкинс и Э. Упфал.

Зафиксируем $\alpha \in (0, 1)$ и $d \geqslant 1, d \in \mathbb{N}$. Граф будем строить итеративно, в качестве начального графа G_0 возьмем d-регулярный граф (граф, у которого степень каждой вершины равна d). Пусть граф с номером t уже построен — это граф $G_t = (V_t, E_t)$, где $V_t = \{u_1, \dots, u_s\}$, а s отличается от t на число вершин начального графа G_0, то есть на $const(d)$. Добавим теперь к графу G_t новую вершину u_{s+1} и d ребер, выходящих из нее. Сделаем это следующим образом: сначала выберем случайную вершину $p \in V_t$ (все вершины V_t равновероятны), затем построим d ребер из u_{s+1} в V_t за d шагов. На каждом шаге с вероятностью α проводим ребро из u_{s+1} в случайную вершину из V_t (все вершины V_t равновероятны), а с вероятностью $1 - \alpha$ проводим ребро из u_{s+1} в i-го соседа вершины p, который всегда найдется, потому что у каждой вершины не менее d соседей.

Глава 3

Практическая часть

Предметом исследования в данной работе является модель Боллобаша–Боргса–Риордана–Чайеса.

3.1 Постановка эксперимента

Рассмотрим степенное распределение степеней вершин графа

$$P = cd^{-\xi},$$

где d — степень вершины графа, P — вероятность встретить вершину степени d в данном графе (то есть отношение количества вершин степени d к общему количеству вершин в графе), c и ξ — некие константы.

Будем считать, что величина ξ является характеристикой графа, по которой можно сравнить несколько графов, то есть эта величина выступает в качестве метрики при сравнении графов. Чем меньше модуль разности между этими величинами у различных графов, тем больше эти графы будем считать похожими друг на друга.

Целью эксперимента является подбор оптимальных параметров α, β, γ, δ_{in} и δ_{out} модели Боллобаша–Боргса–Риордана–Чайеса, при которых величина ξ для модельного графа окажется максимально близкой к величине ξ для реального веб-графа.

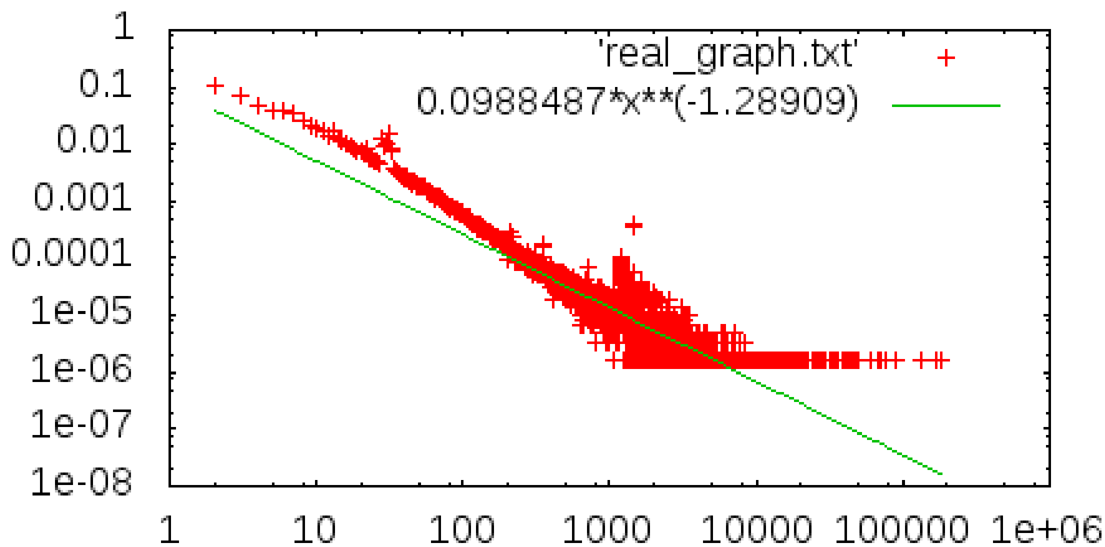
Параметр $\gamma = 1 - \alpha - \beta$, а параметр $\delta_{out} = 0$. Поэтому достаточно перебрать значения параметров α, β и δ_{in}. Значения α и β могут быть от 0 до 1 включительно, однако исключим вырожденные случаи и будем рассматривать значения α и β из множества $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, причём $\alpha + \beta < 1$. Значения параметра δ_{in} рассмотрим из множества $\{10, 20, 30, 40, 50\}$.

В качестве реального веб-графа будем рассматривать срез Интернет-графа за 2005 год, сделанный роботом ведущей российской компании, занимающейся поиском в сети Интернет.

3.2 Результат эксперимента для реального графа

В ходе эксперимента для реального графа получилось значение $\xi = 1.28909$.

Если полученное распределение отобразить на графике с логарифмическими шкалами по обеим осям, отложив степени вершин d по горизонтальной оси и вероятности P по вертикальной оси, то вместе с апроксиммирующей прямой методом наименьших квадратов [4] получаем такой результат:



3.3 Результат эксперимента для модельного графа

Для каждого набора параметров строим модельный граф с временем $t = 100000$ три раза, а потом вычисляем математическое ожидание и дисперсию величины ξ и ищем те значения параметров, при которых среднее значение ξ модельного графа почти совпадёт с значением ξ для реального графа.

В ходе эксперимента были получены следующие результаты:

α	β	γ	δ_{in}	δ_{out}	ξ
0.1	0.1	0.8	10	0	0.932771 ± 0.00728888
0.1	0.1	0.8	20	0	0.974909 ± 0.000788533
0.1	0.1	0.8	30	0	0.969449 ± 0.000961652
0.1	0.1	0.8	40	0	0.961931 ± 0.000463594
0.1	0.1	0.8	50	0	0.931287 ± 0.000423722

α	β	γ	δ_{in}	δ_{out}	ξ
0.1	0.2	0.7	10	0	0.99747 ± 0.000663344
0.1	0.2	0.7	20	0	0.97063 ± 0.00198001
0.1	0.2	0.7	30	0	$0.94963 \pm 0.00000200138$
0.1	0.2	0.7	40	0	1.08471 ± 0.00390443
0.1	0.2	0.7	50	0	1.02931 ± 0.000443945
0.1	0.3	0.6	10	0	1.01148 ± 0.00112183
0.1	0.3	0.6	20	0	0.994045 ± 0.000407881
0.1	0.3	0.6	30	0	1.07286 ± 0.000944741
0.1	0.3	0.6	40	0	1.00004 ± 0.00199681
0.1	0.3	0.6	50	0	0.975628 ± 0.00530568
0.1	0.4	0.5	10	0	1.06259 ± 0.0029934
0.1	0.4	0.5	20	0	1.06301 ± 0.00134939
0.1	0.4	0.5	30	0	1.02225 ± 0.000126111
0.1	0.4	0.5	40	0	1.09805 ± 0.000911361
0.1	0.4	0.5	50	0	1.06724 ± 0.000594735
0.1	0.5	0.4	10	0	1.10449 ± 0.0000661492
0.1	0.5	0.4	20	0	1.09398 ± 0.0000102478
0.1	0.5	0.4	30	0	1.10271 ± 0.000873227
0.1	0.5	0.4	40	0	1.03951 ± 0.000933735
0.1	0.5	0.4	50	0	1.08029 ± 0.00161058
0.1	0.6	0.3	10	0	1.20919 ± 0.000494145
0.1	0.6	0.3	20	0	1.14218 ± 0.000446993
0.1	0.6	0.3	30	0	1.1227 ± 0.00196923
0.1	0.6	0.3	40	0	1.14014 ± 0.00798361
0.1	0.6	0.3	50	0	1.13292 ± 0.00204208

α	β	γ	δ_{in}	δ_{out}	ξ
0.1	0.7	0.2	10	0	1.23367 ± 0.00132783
0.1	0.7	0.2	20	0	1.18782 ± 0.00428949
0.1	0.7	0.2	30	0	1.27585 ± 0.00275136
0.1	0.7	0.2	40	0	1.22912 ± 0.00039264
0.1	0.7	0.2	50	0	1.17065 ± 0.00339632
0.1	0.8	0.1	10	0	1.26342 ± 0.00249616
0.1	0.8	0.1	20	0	1.29528 ± 0.00196035
0.1	0.8	0.1	30	0	1.25078 ± 0.00301471
0.1	0.8	0.1	40	0	1.23568 ± 0.00087426
0.1	0.8	0.1	50	0	1.25477 ± 0.000962454
0.2	0.1	0.7	10	0	1.23841 ± 0.000770949
0.2	0.1	0.7	20	0	1.19279 ± 0.000768557
0.2	0.1	0.7	30	0	1.189 ± 0.000427635
0.2	0.1	0.7	40	0	1.17462 ± 0.000312942
0.2	0.1	0.7	50	0	1.22531 ± 0.00285168
0.2	0.2	0.6	10	0	1.27213 ± 0.00168208
0.2	0.2	0.6	20	0	1.26298 ± 0.0000707486
0.2	0.2	0.6	30	0	1.2835 ± 0.00280229
0.2	0.2	0.6	40	0	1.26348 ± 0.000511625
0.2	0.2	0.6	50	0	1.20518 ± 0.00345746
0.2	0.3	0.5	10	0	1.23412 ± 0.00104592
0.2	0.3	0.5	20	0	1.2875 ± 0.000527015
0.2	0.3	0.5	30	0	1.28566 ± 0.00430134
0.2	0.3	0.5	40	0	1.22661 ± 0.000687818
0.2	0.3	0.5	50	0	1.28648 ± 0.000353701

α	β	γ	δ_{in}	δ_{out}	ξ
0.2	0.4	0.4	10	0	1.30134 ± 0.000357783
0.2	0.4	0.4	20	0	1.29683 ± 0.000250697
0.2	0.4	0.4	30	0	1.33242 ± 0.00190293
0.2	0.4	0.4	40	0	$1.30397 \pm 0.00000382383$
0.2	0.4	0.4	50	0	1.28801 ± 0.00389082
0.2	0.5	0.3	10	0	1.36286 ± 0.000763779
0.2	0.5	0.3	20	0	1.30001 ± 0.00136646
0.2	0.5	0.3	30	0	1.33997 ± 0.00108664
0.2	0.5	0.3	40	0	1.37349 ± 0.0039013
0.2	0.5	0.3	50	0	1.33394 ± 0.000192949
0.2	0.6	0.2	10	0	1.4149 ± 0.00720377
0.2	0.6	0.2	20	0	1.42447 ± 0.00134053
0.2	0.6	0.2	30	0	1.38158 ± 0.000679881
0.2	0.6	0.2	40	0	1.39088 ± 0.000561193
0.2	0.6	0.2	50	0	1.34371 ± 0.000489887
0.2	0.7	0.1	10	0	1.43066 ± 0.000642419
0.2	0.7	0.1	20	0	1.46199 ± 0.0000889685
0.2	0.7	0.1	30	0	1.46693 ± 0.00109336
0.2	0.7	0.1	40	0	1.44535 ± 0.00547142
0.2	0.7	0.1	50	0	1.47933 ± 0.00261129
0.3	0.1	0.6	10	0	1.44605 ± 0.00110126
0.3	0.1	0.6	20	0	1.49866 ± 0.00207915
0.3	0.1	0.6	30	0	1.4928 ± 0.00290065
0.3	0.1	0.6	40	0	1.47628 ± 0.00233522
0.3	0.1	0.6	50	0	1.47639 ± 0.00105536

α	β	γ	δ_{in}	δ_{out}	ξ
0.3	0.2	0.5	10	0	1.5047 ± 0.00204238
0.3	0.2	0.5	20	0	1.51964 ± 0.000521645
0.3	0.2	0.5	30	0	1.48895 ± 0.00135505
0.3	0.2	0.5	40	0	1.53566 ± 0.000545929
0.3	0.2	0.5	50	0	1.47804 ± 0.00143119
0.3	0.3	0.4	10	0	1.52457 ± 0.000237678
0.3	0.3	0.4	20	0	1.52941 ± 0.000652224
0.3	0.3	0.4	30	0	1.51453 ± 0.00176311
0.3	0.3	0.4	40	0	1.55945 ± 0.0000687918
0.3	0.3	0.4	50	0	1.53562 ± 0.00641355
0.3	0.4	0.3	10	0	1.60746 ± 0.000783873
0.3	0.4	0.3	20	0	1.53671 ± 0.00185931
0.3	0.4	0.3	30	0	1.59394 ± 0.00932542
0.3	0.4	0.3	40	0	1.56606 ± 0.000836987
0.3	0.4	0.3	50	0	1.56114 ± 0.00178809
0.3	0.5	0.2	10	0	1.65333 ± 0.00334823
0.3	0.5	0.2	20	0	1.63995 ± 0.0030323
0.3	0.5	0.2	30	0	1.67205 ± 0.00117298
0.3	0.5	0.2	40	0	1.65376 ± 0.000025201
0.3	0.5	0.2	50	0	1.608 ± 0.000610703
0.3	0.6	0.1	10	0	1.71611 ± 0.000653836
0.3	0.6	0.1	20	0	1.67483 ± 0.00165382
0.3	0.6	0.1	30	0	1.67222 ± 0.00446685
0.3	0.6	0.1	40	0	1.67199 ± 0.000164499
0.3	0.6	0.1	50	0	1.70552 ± 0.000339063

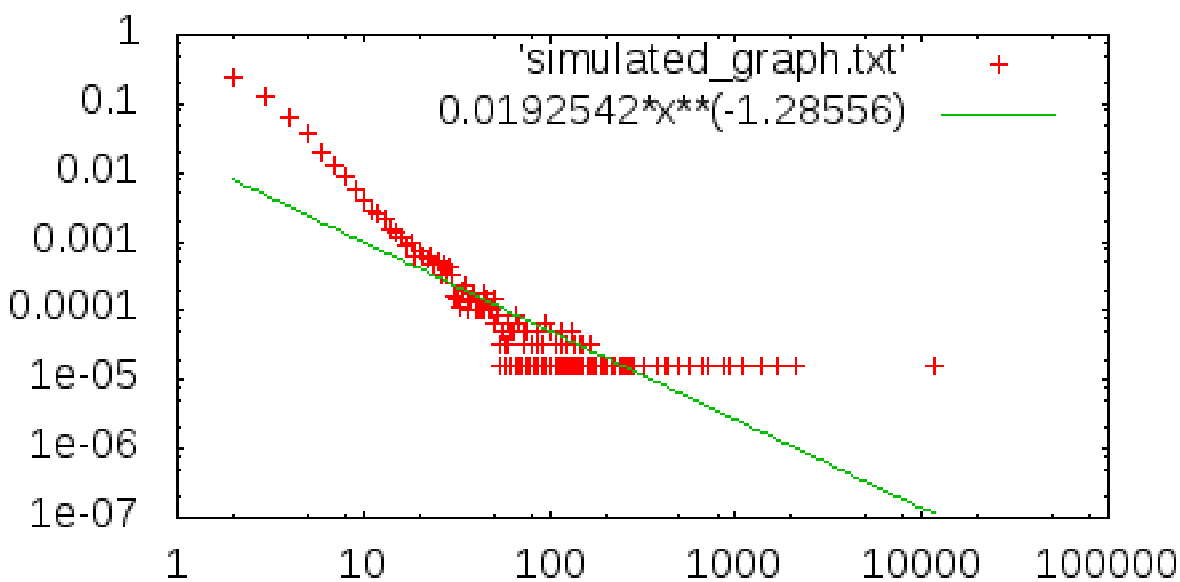
α	β	γ	δ_{in}	δ_{out}	ξ
0.4	0.1	0.5	10	0	1.85665 ± 0.000804998
0.4	0.1	0.5	20	0	1.82345 ± 0.00610725
0.4	0.1	0.5	30	0	1.78459 ± 0.000654807
0.4	0.1	0.5	40	0	1.81684 ± 0.00544487
0.4	0.1	0.5	50	0	1.82816 ± 0.00448046
0.4	0.2	0.4	10	0	1.84631 ± 0.00114318
0.4	0.2	0.4	20	0	1.84171 ± 0.00335591
0.4	0.2	0.4	30	0	1.83696 ± 0.000794708
0.4	0.2	0.4	40	0	1.80721 ± 0.000486551
0.4	0.2	0.4	50	0	1.81096 ± 0.00400177
0.4	0.3	0.3	10	0	1.88461 ± 0.00351277
0.4	0.3	0.3	20	0	1.91742 ± 0.00201697
0.4	0.3	0.3	30	0	1.86504 ± 0.00102953
0.4	0.3	0.3	40	0	1.86436 ± 0.000598827
0.4	0.3	0.3	50	0	1.87625 ± 0.000134616
0.4	0.4	0.2	10	0	1.9842 ± 0.0052373
0.4	0.4	0.2	20	0	1.93133 ± 0.000686224
0.4	0.4	0.2	30	0	1.93827 ± 0.000770509
0.4	0.4	0.2	40	0	1.91178 ± 0.00226971
0.4	0.4	0.2	50	0	1.92069 ± 0.00348812
0.4	0.5	0.1	10	0	1.96767 ± 0.00120533
0.4	0.5	0.1	20	0	1.9868 ± 0.00369677
0.4	0.5	0.1	30	0	1.94129 ± 0.00209144
0.4	0.5	0.1	40	0	1.99606 ± 0.00168267
0.4	0.5	0.1	50	0	1.96347 ± 0.00595658

α	β	γ	δ_{in}	δ_{out}	ξ
0.5	0.1	0.4	10	0	2.33274 ± 0.00494496
0.5	0.1	0.4	20	0	2.21657 ± 0.0109728
0.5	0.1	0.4	30	0	2.25624 ± 0.00343628
0.5	0.1	0.4	40	0	2.24872 ± 0.00235612
0.5	0.1	0.4	50	0	2.23092 ± 0.00780588
0.5	0.2	0.3	10	0	2.32951 ± 0.00889997
0.5	0.2	0.3	20	0	2.26997 ± 0.00380407
0.5	0.2	0.3	30	0	2.31947 ± 0.00044122
0.5	0.2	0.3	40	0	2.28689 ± 0.00363055
0.5	0.2	0.3	50	0	2.31509 ± 0.000536671
0.5	0.3	0.2	10	0	2.31756 ± 0.00153393
0.5	0.3	0.2	20	0	2.32366 ± 0.000276932
0.5	0.3	0.2	30	0	2.38659 ± 0.00256113
0.5	0.3	0.2	40	0	2.24827 ± 0.00450243
0.5	0.3	0.2	50	0	2.38144 ± 0.0081087
0.5	0.4	0.1	10	0	2.34105 ± 0.000838409
0.5	0.4	0.1	20	0	2.46757 ± 0.00180566
0.5	0.4	0.1	30	0	2.40895 ± 0.000978078
0.5	0.4	0.1	40	0	2.45435 ± 0.000863965
0.5	0.4	0.1	50	0	2.40079 ± 0.0028487
0.6	0.1	0.3	10	0	2.83816 ± 0.0010161
0.6	0.1	0.3	20	0	2.86843 ± 0.00103467
0.6	0.1	0.3	30	0	2.80655 ± 0.0119543
0.6	0.1	0.3	40	0	2.83118 ± 0.0092681
0.6	0.1	0.3	50	0	2.82788 ± 0.00852022

α	β	γ	δ_{in}	δ_{out}	ξ
0.6	0.2	0.2	10	0	2.80257 ± 0.00680838
0.6	0.2	0.2	20	0	2.77229 ± 0.00907522
0.6	0.2	0.2	30	0	2.90865 ± 0.00321538
0.6	0.2	0.2	40	0	2.8512 ± 0.000240225
0.6	0.2	0.2	50	0	2.74585 ± 0.00341822
0.6	0.3	0.1	10	0	2.85822 ± 0.00700646
0.6	0.3	0.1	20	0	2.88147 ± 0.00071349
0.6	0.3	0.1	30	0	2.7895 ± 0.00457419
0.6	0.3	0.1	40	0	3.01195 ± 0.00406189
0.6	0.3	0.1	50	0	3.00996 ± 0.000130839
0.7	0.1	0.2	10	0	3.43458 ± 0.0102724
0.7	0.1	0.2	20	0	3.4369 ± 0.03133
0.7	0.1	0.2	30	0	3.48409 ± 0.0210833
0.7	0.1	0.2	40	0	3.63702 ± 0.00643128
0.7	0.1	0.2	50	0	3.64236 ± 0.019093
0.7	0.2	0.1	10	0	3.51785 ± 0.000243311
0.7	0.2	0.1	20	0	3.41559 ± 0.0126742
0.7	0.2	0.1	30	0	3.52958 ± 0.00600853
0.7	0.2	0.1	40	0	3.47119 ± 0.0235151
0.7	0.2	0.1	50	0	3.44423 ± 0.013753
0.8	0.1	0.1	10	0	3.95465 ± 0.00389398
0.8	0.1	0.1	20	0	4.07212 ± 0.0165141
0.8	0.1	0.1	30	0	4.24585 ± 0.00697883
0.8	0.1	0.1	40	0	4.16692 ± 0.00618292
0.8	0.1	0.1	50	0	4.2845 ± 0.00206282

Как видно из приведённой выше таблицы, наилучшие значение параметров модели получились $\alpha = 0.2, \beta = 0.4, \gamma = 0.4, \delta_{in} = 50, \delta_{out} = 0$. При этих параметрах у модельного графа $\xi = 1.28801 \pm 0.00389082$. Таким образом, значение ξ у модельного графа отличается от значения ξ для реального графа всего на 0.00108132.

Если аналогичным образом изобразить на графике распределение степеней вершин для модельного графа с наилучшими параметрами и провести апроксиммирующую прямую методом наименьших квадратов, то мы получим:



Глава 4

Программа

Для моделирования веб-графов в соответствии с моделью Боллобаша–Боргса–Риордана–Чайеса был реализован программный продукт, позволяющий считывать реальный ориентированный веб-граф, моделировать графы с различными значениями параметров, сравнивать реальный и модельный графы по распределению степеней вершин и находить те значения параметров, при которых модельный граф наилучшим образом отражает некоторые характеристики реального веб-графа.

4.1 Модуль Graph

Программный модуль *Graph* позволяет хранить ориентированный граф с заданным количеством вершин и рёбрами между ними, изменять и узнавать количество вершин в графе, добавлять новые вершины и рёбра, узнавать о наличии либо отсутствии ребра между заданными двумя вершинами, а также вычислять показатель одной из важных характеристик графа — распределения степеней вершин.

Каждый объект графа инкапсулирует целочисленную переменную — количество вершин в графе, а также множество упорядоченных пар целых чисел. В каждой паре первое число означает номер вершины, из которой выходит ребро графа, а второе число означает номер вершины, в которую входит ребро графа. Все пары образуют множество рёбер графа.

Конструктор по умолчанию создаёт пустой граф, то есть граф, в котором нет ни одной вершины и ни одного ребра, а конструктор копирования полностью копирует ориентированный граф из другого объекта, сохраняя все вершины и рёбра между ними. Деструктор удаляет имеющийся граф.

Функция $SetVertexCount$ принимает в качестве параметра целое число, являющееся новым значением количества вершин графа, и устанавливает в графе данное количество вершин. Функция $GetVertexCount$ позволяет узнать текущее количество вершин графа. Обе функции работают за константное время.

Функция $AddEdge$ принимает в качестве параметров два целых числа — номера вершин графа, между которыми требуется добавить ребро в граф. Первый параметр — это номер вершины графа, из которой выходит ребро, а второй параметр — номер вершины, в которую входит ребро. Вершины нумеруются с нуля. Функция создаёт упорядоченную пару из этих двух чисел и добавляет созданную пару в множество рёбер графа. Функция $GetEdge$ принимает точно такие же параметры и проверяет, имеется ли в графе ребро, выходящее из вершины, номер которой указан в первом параметре, в вершину, номер которой указанан во втором параметре. Для этого функция создаёт упорядоченную пару чисел и ищет её в множестве вершин графа. Если такая пара находится, возвращается истина, иначе — ложь. Обе функции работают за логарифмическое время от количества рёбер в графе.

Функция $EstimateXi$ оценивает характеристику распределения степеней вершин графа. Степенью вершины считается суммарное количество входящих в вершину рёбер и исходящих из вершины рёбер. Вероятностью встретить вершину заданной степени является отношение количества вершин заданной степени к общему количеству вершин графа. Предполагается, что распределение степеней вершин графа имеет следующий вид:

$$P = cd^{-\xi},$$

где d — степень вершины, P — вероятность встретить вершину степени d в исследуемом графе, c и ξ — некие константы, причём c нас интересовать не будет, а вот ξ как раз и является той величиной, которую вычисляет обсуждаемая функция для имеющегося графа. Величина ξ позволяет сравнивать разные графы, то есть выступает в роли некоторой метрики: чем меньше разница между этими величинами у разных графов, тем более похожими мы их будем считать.

Для вычисления степеней всех вершин в фукнции $EstimateXi$ создаётся вектор длиной, равной количеству вершин в графе, и заполняется нулями. В каждой ячейке этого вектора будет храниться степень вершины графа (номер ячейки вектора равен номеру вершины графа). После этого перебираются все рёбра графа из множества рёбер графа, у каждого ребра определяются номера вершин, из которых выходит ребро и в которых входит ребро, а потом значения соответствующих ячеек вектора увеличиваются на единицу.

Для вычисления вероятностей встретить вершину с заданной степенью сначала строится отображение из степени вершины графа в количество вершин графа с такой степенью, которое хранится в контейнере map. Мы проходим по всем вершинам графа, для каждой вершины с помощью ранее вычисленного вектора определяем её степень, а дальше проверяем наличие в контейнере map соответствующей пары. Если в контейнере для данной степени уже есть количество таких вершин, то мы просто увеличиваем это количество на единицу, а если такой степени ещё не было в контейнере, то мы добавляем туда новую пару с данной степенью и количеством один. После построения такого отображения мы можем вычислить вероятность встретить вершину с соответствующей степенью.

Поскольку распределение степеней вершин графа $P = cd^{-\xi}$ имеет показательный вид, прологарифмируем его и получим

$$\ln P = \ln c - \xi \ln d$$

Полученную прямую можно найти с помощью метода наименьших квадратов, для этого введём следующие обозначения:

$$y_t = a + bx_t + \varepsilon_t$$

$$y_t = \ln P$$

$$a = \ln c$$

$$b = \xi$$

$$x_t = -\ln d$$

По формулам из метода наименьших квадратов

$$\hat{b} = \frac{Cov(x, y)}{Var(x)} = \frac{\overline{xy} - \overline{x}\overline{y}}{\overline{x^2} - (\overline{x})^2}$$

$$\hat{a} = \overline{y} - b\overline{x}$$

При реализации этой формулы в программе не учитываются изолированные вершины, то есть вершины степени нуль, поскольку степени вершин должны быть прологарифмированы, а логарифм нуля не существует. Кроме того вероятность встретить вершину определённой степени может оказаться близкой к нулю и из-за погрешностей вычислений чисел с плавающей точкой не получится вычислить логарифм данного числа. Поэтому вершины такой степени, вероятность встретить которые менее 10^{-9} также не учитываются. При вычислении средних значений используется количество учтённых в подсчёте вершин.

Для построения графика зависимости вероятности встретить вершину определённой степени от степеней вершин можно добавить одну строчку в эту функцию, чтобы вывести все точки для графика в файл, а потом построить график, например, с помощью программы *gnuplot*. Значение константы c можно было бы не вычислять, но для наглядности график можно дополнить прямой, полученной методом наименьших квадратов, поэтому требуется вычислить значение c.

Результатом работы данной функции является вычисленное значение константы ξ, которое и возвращается.

4.2 Модуль RealGraph

Программный модуль *RealGraph* позволяет прочитать входные данные реального веб-графа. В данном модуле реализован класс *RealGraph*, который пронаследован от класса Graph, описанного выше.

Конструктор по умолчанию создаёт пустой граф вызовом конструктора по умолчанию для базового класса, а конструктор копирования полностью копирует ориентированный граф из другого объекта, сохраняя все вершины и рёбра между ними, также с помощью вызова конструктора копирования для базового класса. Деструктор удаляет имеющийся граф.

Функция *LoadRealGraph* принимает в качестве параметра строку, в которой записано имя входного файла с реальным веб-графом. Входной файл должен быть следующего формата: каждая строка содержит несколько полей, разделённых пробелами. Первые два поля каждой строки — это префикс хоста и наименование хоста, которые игнорируются данной функцией. В третьем поле содержится идентификатор вершины графа, а все поля, начиная с четвёртого и до конца строки, содержат идентификаторы вершин графа, из которых выходят рёбра, входящие в вершину, указанную в третьм поле строки. Число -1 в третьем поле означает внешнюю вершину графа для возможности указания рёбер, идущих из вершин графа наружу или приходящих в граф извне. Подобные строки игнорируются данной функцией.

С помощью простого скрипта на языке *Python* была вычислена максимальная длина среди всех строк входного файла, а потом был создан буфер в один миллион символов (с запасом), чтобы в него точно поместилась любая строка входного файла.

Файл открывается и читается построчно. Строка записывается в буфер, в ней дважды ищется символ пробела, чтобы пропустить первые два поля, а потом читается число — идентификатор вершины. Для чтения чисел из строки используется функция *strtok* из стандартной библиотеки *cstring*. Если этот идентификатор оказывается равным -1, то строка пропускается и мы переходим к обработке следующей строки, иначе читаются все остальные числа до конца строки в цикле.

Все идентификаторы вершин сохраняются в множестве, а рёбра в виде упорядоченных пар идентификаторов вершин — в векторе. Первым числом пары является идентификатор вершины, из которой выходит ребро (четвёртое поле строки и далее), а вторым числом пары является идентификатор вершины, в которую входит ребро (третье поле строки).

После чтения файла у нас образуется множество различных идентификаторов вершин, которые нам нужно обойти и перенумеровать от 0 до N, где N — количество вершин графа. Для этого мы создаём контейнер map и для каждого идентификатора записываем в него порядковый номер этого идентификатора при обходе множества итератором.

Теперь остаётся установить количество вершин в графе равным мощности множества различных идентификаторов вершин с помощью вызова функции базового класса, а также пройтись по вектору пар идентификаторов, для каждого индентификатора из пары с помощью отображения узнать его порядковый номер (номер вершины графа) и добавить соответствующее ребро в граф с помощью вызова функции базового класса.

4.3 Модуль SimulatedGraph

Программный модуль *SimulatedGraph* позволяет смоделировать веб-граф по модели Боллобаша–Боргса–Риордана–Чайеса. В данном модуле реализован класс *SimulatedGraph*, который пронаследован от класса Graph, описанного выше. Класс позволяет устанавливать заданные значения параметров и узнавать установленные значения параметров, выбирать вершины обеими описанными в модели способами и генерировать граф для любого заданного значения параметра времени генерации.

Каждый объект графа инкапсулирует параметры модели $\alpha, \beta, \gamma, \delta_{in}, \delta_{out}$, а также векторы $in\_numerator$ и $out\_numerator$ для хранения значений числителей вероятностей для выбора соответствующих вершин при моделировании.

Конструктор по умолчанию создаёт пустой граф вызовом конструктора по умолчанию для базового класса и заполняет параметры модели значениями по умолчанию, а конструктор копирования полностью копирует ориентированный граф из другого объекта, сохраняя все вершины и рёбра между ними, также с помощью вызова конструктора копирования для базового класса и копирует установленные параметры модели. Деструктор удаляет имеющийся граф.

Функции $SetAlpha$, $SetBeta$ и $SetDeltaIn$ принимают в качестве параметра дробное число и устанавливают значение α, β и δ_{in} соответственно равным заданному числу. Поскольку $\gamma = 1 - \alpha - \beta$ и $\delta_{out} = 0$, то эти значения устанавливаются автоматически, а γ автоматически пересчитывается при установке нового значения α и/или нового значения β.

Функции $GetAlpha$, $GetBeta$, $GetGamma$, $GetDeltaIn$, $GetDeltaOut$ возвращают текущее установленное значение параметра α, β, γ, δ_{in} и δ_{out} соответственно.

Функция $ChooseVertexAccordingToIn$ выбирает одну из вершин графа по описанному в модели принципу. В нулевой ячейке вектора $in\_numerator$ хранится значение числителя вероятности, с которой следует выбрать нулевую вершину, а в произвольной i-ой ячейке хранится сумма числителей вероятностей, с которыми следует выбрать вершины от нулевой до i-ой включительно. Поэтому в последней ячейке этого вектора по сути хранится число, которое можно считать длиной некоторого отрезка, на который требуется кинуть случайную точку, а потом узнать, в какой ячейке записано минимальное число, больше выпавшего случайной точкой. Функция использует генератор псевдослучайных чисел для получения случайного числа в нужном диапазоне, а далее с помощью алгортма $upper\_bound$ находит нужную вершину (для корректной работы этой функции значение последней ячейки вектора временно увеличивается на 1, а потом возвращается обратно).

Функция $ChooseVertexAccordingToOut$ работает аналогично функции $ChooseVertexAccordingToIn$, только работает с вектором $out\_numerator$.

Функция $GenerateGraph$ получает в качестве параметра время генерации модельного графа, а затем моделирует граф по модели Боллобаша–Боргса–Риордана–Чайеса.

Сначала функция очищает векторы $in\_numerator$ и $out\_numerator$ и создаёт граф, состоящий из одной вершины с петлёй, а также записывает в векторы начальные значение вероятностей $1 + \delta_{in}$ и $1 + \delta_{out}$.

Далее функция выполняет цикл заданное в параметре функции время. На каждой итерации определяется псевдослучайное число от 0 до 1 включительно. Дальше реализуется один из трёх случаев: с вероятностью α выбирается вершина функцией $ChooseVertexAccordingToIn$, добавляется новая вершина в граф и новое ребро из добавленной вершины в выбранную, с вероятностью β выбираются две вершины графа (первая вершина выбирается функцией $ChooseVertexAccordingToOut$, вторая вершина выбирается функцией $ChooseVertexAccordingToIn$) и проводится ребро из первой во вторую, с вероятностью γ выбирается вершина графа функцией $ChooseVertexAccordingToOut$, добавляется новая вершина в граф и проводится ребро из выбранной вершины в добавленную. В каждом из трёх случаев векторы $in\_numerator$ и $out\_numerator$ изменяются в соответствии с добавленными вершинами и рёбрами (для новой вершины добавляется ячейка в вектор с начальным значением вероятности, после добавления ребра во все ячейки нужного вектора, начиная с номера вершины, у которой добавилось ребро, все значения увеличиваются на единицу, потому что поменялась степень вершины, а значит и кумулятивные суммы).

Полученный в результате граф может быть оценён функцией базового класса $EstimateXi$ для сравнения с реальным графом.

4.4 Модуль Main

Программный модуль $Main$ предназначен для подбора наилучших значений параметров модели Боллобаша–Боргса–Риордана–Чайеса. В начале модуля задаются константные массивы перебираемых значений параметров, а также некоторые важные константы (время генерирования модельного графа, начальная инициализация генератора датчика псевдослучайных чисел, количество повторений моделирования графа с одними и теми же параметрами, точность вычислений значений параметров с плавающей точкой).

Функция $main$ инициализирует датчик псевдослучайных чисел, создаёт объект реального графа и функцией $LoadRealGraph$ загружает граф из входного файла, затем функцией $EstimateXi$ оценивает показатель степенного распределения вероятностей степеней вершин и выводит это значение в стандартный поток вывода. После этого инициализируются значения переменных для наилучших значений параметров и в циклах перебираются все возможные сочетания параметров модели с учётом условия $\alpha+\beta < 1-\varepsilon$.

На каждой итерации внутреннего цикла создаётся объект модельного графа, функциями $SetAlpha$, $SetBeta$ и $SetDeltaIn$ устанавливаются текущие значения параметров, а затем функцией $GenerateGraph$ моделируется граф. Затем у полученного графа функцией $EstimateXi$ вычисляется показатель степенного распределения степеней вершин. Все эти действия повторяются заданное количество раз, а потом вычисляется математическое ожидание и дисперсия величины ξ, чтобы она меньше зависела от случайностей. Если полученное среднее значение ξ оказалось лучше ранее найденного, то текущие параметры модели сохраняются в качестве наилучших, а также выводится информация о текущих результатах в стандартный поток вывода.

В конце выводится информация о наилучших значениях параметров в стандартный поток вывода, а также информация о том, насколько сильно ξ для реального графа отличается от этой же величины для смоделированного графа при полученных наилучших параметрах.

Глава 5

Выводы

В данной работе исследовалась модель Боллобаша–Боргса–Риордана–Чайеса, варьировались параметры этой модели. Для каждого набора параметров модели генерировались случайные веб-графы и сравнивались по своим свойствам с реальным графом.

На основе проделанной работы можно сделать вывод о том, что модель случайного веб-графа Боллобаша–Боргса–Риордана–Чайеса позволяет моделировать веб-графы, которые близки по свойствам к реальным. А именно, позволяет генерировать графы, у которых параметр ξ из распределения степеней вершин $P = cd^{-\xi}$ совпадает с этим же параметром реального графа. Понятно, что указанный параметр отражает лишь часть свойств веб-графа, поэтому в будущей работе планируется рассмотреть другие критерии сравнения схожести графов.

Литература

[1] Райгородский А. М. Модели случайных графов и их применения.

[2] Maxim Zhukovskiy, Dmitry Vinogradov, Yuri Pritykin, Liudmila Ostroumova, Evgeniy Grechnikov, Gleb Gusev, Pavel Serdyukov, Andrei Raigorodskii. Empirical Validation of the Buckley–Osthus Model for the Web Host Graph: Degree and Edge Distributions. Yandex, 16 Leo Tolstoy St., Moscow, 119021 Russia, August 14, 2012.

[3] Bela Bollobas, Christian Borgs, Jennifer Chayes, Oliver Riordan. "Directed Scale-free Graph". Proc. 14th ACM-SIAM Symposium on Discrete Algorithms, 132-139, 2003.

[4] http://ru.wikipedia.org/wiki/Метод_наименьших_квадратов

[5] Kulwadee Somboonviwat, Shinji Suzuki, Masashi Toyoda, Masaru Kitsuregawa. Characterization of the Thai Hostgraph.

Приложение A

Исходный код программы

```
#/usr/bin/env gnuplot

set terminal png size 480,240
set logscale xy

set output 'real_graph.png'
plot 'real_graph.txt', 0.0988487*x**(-1.28909)

set output 'simulated_graph.png'
plot 'simulated_graph.txt', 0.0192542*x**(-1.28556)
```

```
#/usr/bin/env python

### max_line_length.py ###

def main():
    max_line_length = 0
    for row in open('links.txt', 'r'):
        if len(row) > max_line_length:
            max_line_length = len(row)
    print max_line_length

if __name__ == '__main__':
    main()
```

```
/*** graph.h ***/
#ifndef __GRAPH_H__
#define __GRAPH_H__

#include <set>
#include <utility>

class Graph {
 public:
  Graph();
  Graph(const Graph& graph);
  ~Graph();
  void SetVertexCount(int vertex_count);
  int GetVertexCount();
  void AddEdge(int first_vertex, int second_vertex);
  bool GetEdge(int first_vertex, int second_vertex);
  double EstimateXi();
 protected:
  int vertex;
  std::set<std::pair<int, int> > edges;
};

#endif
```

```
/*** graph.cpp ***/
#include "graph.h"
#include <algorithm>
#include <cmath>
#include <map>
#include <vector>
#include <iostream>

Graph::Graph() : vertex(0) {
}

Graph::Graph(const Graph& graph) : vertex(graph.vertex), edges(graph.edges) {
}

Graph::~Graph() {
}

void Graph::SetVertexCount(int vertex_count) {
  vertex = vertex_count;
}

int Graph::GetVertexCount() {
  return vertex;
}

void Graph::AddEdge(int first_vertex, int second_vertex) {
  edges.insert(std::make_pair(first_vertex, second_vertex));
}

bool Graph::GetEdge(int first_vertex, int second_vertex) {
  return edges.find(std::make_pair(first_vertex, second_vertex)) != edges.end();
}

double Graph::EstimateXi() {
  // Evaluate vertex degrees
  std::vector<int> vertex_degree(GetVertexCount(), 0);
  for (std::set<std::pair<int, int> >::iterator edge = edges.begin();
       edge != edges.end(); ++edge) {
```

```
    ++vertex_degree[edge->first];
    ++vertex_degree[edge->second];
  }

  // Evaluate probabilities
  std::map<int, int> count_vertex_degree;
  for (int index = 0; index < vertex_degree.size(); ++index) {
    int degree = vertex_degree[index];
    std::map<int, int>::iterator iter = count_vertex_degree.find(degree);
    if (iter == count_vertex_degree.end()) {
      count_vertex_degree[degree] = 1;
    } else {
      ++count_vertex_degree[degree];
    }
  }

  // P = c * d^(-xi)
  // ln(P) = ln(c) - xi * ln(d)
  // y_t = a + b * x_t + epsilon_t
  // y_t = ln(P)
  // a = ln(c)
  // b = xi
  // x_t = -ln(d)

  double xy = 0;
  double x = 0;
  double y = 0;
  double xx = 0;
  int count = 0;

  for (std::map<int, int>::iterator iter = count_vertex_degree.begin();
       iter != count_vertex_degree.end(); ++iter) {
    double degree = iter->first;
    double probability = iter->second;
    probability /= GetVertexCount();
    const double EPS = 1e-9;
    if (degree > 1 && probability >= EPS) {
      double dx = -log(degree);
```

```
      double dy = log(probability);
      xy += dx * dy;
      x += dx;
      y += dy;
      xx += dx * dx;
      ++count;

      // Uncomment this line to output points to build a plot
      // std::cerr << degree << ' ' << probability << std::endl;
    }
  }

  xy /= count;
  x /= count;
  y /= count;
  xx /= count;

  // Ordinary least squares (OLS)
  double xi = (xy - x * y) / (xx - x * x);

  // Uncomment these lines to output constant to build a line on plot
  // double c = y - xi * x;
  // std::cout << "c = " << exp(c) << std::endl;

  return xi;
}
```

```
/*** simulated_graph.h ***/
#ifndef __SIMULATED_GRAPH_H__
#define __SIMULATED_GRAPH_H__

#include "graph.h"
#include <vector>

class SimulatedGraph : public Graph {
 public:
  SimulatedGraph();
  SimulatedGraph(const SimulatedGraph& simulated_graph);
  ~SimulatedGraph();
  void SetAlpha(double a);
  void SetBeta(double b);
  void SetDeltaIn(double d_in);
  double GetAlpha();
  double GetBeta();
  double GetGamma();
  double GetDeltaIn();
  double GetDeltaOut();
  int ChooseVertexAccordingToIn();
  int ChooseVertexAccordingToOut();
  void GenerateGraph(int time);
 private:
  double alpha;
  double beta;
  double gamma;
  double delta_in;
  double delta_out;
  std::vector<double> in_numerator;
  std::vector<double> out_numerator;
};

#endif
```

```
/*** simulated_graph.cpp ***/
#include "simulated_graph.h"
#include <algorithm>
#include <cstdlib>
#include <set>

SimulatedGraph::SimulatedGraph()
     : Graph()
     , alpha(0.1), beta(0.2), gamma(0.7)
     , delta_in(0.0), delta_out(0.0) {
}

SimulatedGraph::SimulatedGraph(const SimulatedGraph& simulated_graph)
  : Graph(simulated_graph)
  , alpha(simulated_graph.alpha)
  , beta(simulated_graph.beta)
  , gamma(simulated_graph.gamma)
  , delta_in(simulated_graph.delta_in)
  , delta_out(simulated_graph.delta_out) {
}

SimulatedGraph::~SimulatedGraph() {
}

void SimulatedGraph::SetAlpha(double a) {
  alpha = a;
  gamma = 1.0 - alpha - beta;
}

void SimulatedGraph::SetBeta(double b) {
  beta = b;
  gamma = 1.0 - alpha - beta;
}

void SimulatedGraph::SetDeltaIn(double d_in) {
  delta_in = d_in;
}
```

```
double SimulatedGraph::GetAlpha() {
  return alpha;
}

double SimulatedGraph::GetBeta() {
  return beta;
}

double SimulatedGraph::GetGamma() {
  return gamma;
}

double SimulatedGraph::GetDeltaIn() {
  return delta_in;
}

double SimulatedGraph::GetDeltaOut() {
  return delta_out;
}

int SimulatedGraph::ChooseVertexAccordingToIn() {
  // Choose floating point random number in 0 to vertex count inclusively
  double random_point =
      static_cast<double>(rand()) / RAND_MAX * in_numerator.back();

  // Adjust the last number for propper use of upper_bound
  in_numerator[in_numerator.size() - 1] += 1.0;

  // Choose a vertex according to in
  int current_vertex =
      std::upper_bound(in_numerator.begin(), in_numerator.end(), random_point)
      - in_numerator.begin();

  // Turn the last number back again
  in_numerator[in_numerator.size() - 1] -= 1.0;

  // Return chosen vertex number
  return current_vertex;
```

```
}

int SimulatedGraph::ChooseVertexAccordingToOut() {
  // Choose floating point random number in 0 to vertex count inclusively
  double random_point =
      static_cast<double>(rand()) / RAND_MAX * out_numerator.back();

  // Adjust the last number for propper use of upper_bound
  out_numerator[out_numerator.size() - 1] += 1.0;

  // Choose a vertex according to out
  int current_vertex =
      std::upper_bound(out_numerator.begin(), out_numerator.end(), random_point)
      - out_numerator.begin();

  // Turn the last number back again
  out_numerator[out_numerator.size() - 1] -= 1.0;

  // Return chosen vertex number
  return current_vertex;
}

void SimulatedGraph::GenerateGraph(int time) {
  // Prepare vectors for probability numerators
  in_numerator.clear();
  out_numerator.clear();

  // Start with 1 vertex and a loop
  SetVertexCount(1);
  AddEdge(0, 0);
  in_numerator.push_back(1 + GetDeltaIn());
  out_numerator.push_back(1 + GetDeltaOut());

  // Time counter
  for (int t = 1; t < time; ++t) {
    // Generate floating point random number in 0 to 1 inclusively
    double random_point = static_cast<double>(rand()) / RAND_MAX;
```

```
if (random_point <= GetAlpha()) { // With alpha probability
  // Choose existing vertex according to in, add a new vertex
  // and an edge from the new vertex to the chosen vertex
  int existing_vertex = ChooseVertexAccordingToIn();
  int new_vertex = GetVertexCount();
  SetVertexCount(GetVertexCount() + 1);
  AddEdge(new_vertex, existing_vertex);

  // Adjust probability numerators
  in_numerator.push_back(in_numerator.back() + GetDeltaIn());
  out_numerator.push_back(out_numerator.back() + 1.0 + GetDeltaOut());
  for (int index = existing_vertex; index < in_numerator.size(); ++index) {
    in_numerator[index] += 1.0;
  }
} else if (random_point <= GetAlpha() + GetBeta()) {// With beta probability
  // Choose two existing vertices according to out and in
  // and add an edge between them
  int existing_vertex_one = ChooseVertexAccordingToOut();
  int existing_vertex_two = ChooseVertexAccordingToIn();
  AddEdge(existing_vertex_one, existing_vertex_two);

  // Adjust probability numerators
  for (int index = existing_vertex_one;
       index < out_numerator.size(); ++index) {
    out_numerator[index] += 1.0;
  }
  for (int index = existing_vertex_two;
       index < in_numerator.size(); ++index) {
    in_numerator[index] += 1.0;
  }
} else { // With gamma probability
  // Choose existing vertex according to out, add a new vertex
  // and an edge from the existing vertex to the new one
  int existing_vertex = ChooseVertexAccordingToOut();
  int new_vertex = GetVertexCount();
  SetVertexCount(GetVertexCount() + 1);
  AddEdge(existing_vertex, new_vertex);
```

```
      // Adjust probability numerators
      in_numerator.push_back(in_numerator.back() + 1.0 + GetDeltaIn());
      out_numerator.push_back(out_numerator.back() + GetDeltaOut());
      for (int index = existing_vertex; index < out_numerator.size(); ++index) {
        out_numerator[index] += 1.0;
      }
    }
  }
}
```

```
/*** real_graph.h ***/
#ifndef __REAL_GRAPH_H__
#define __REAL_GRAPH_H__

#include "graph.h"
#include <string>

class RealGraph : public Graph {
 public:
  RealGraph();
  RealGraph(const RealGraph& real_graph);
  ~RealGraph();
  void LoadRealGraph(const std::string& links);
};

#endif
```

```
/*** real_graph.cpp ***/
#include "real_graph.h"
#include <cstring>
#include <cstdlib>
#include <fstream>
#include <map>
#include <set>
#include <vector>

RealGraph::RealGraph() : Graph() {
}

RealGraph::RealGraph(const RealGraph& real_graph) : Graph(real_graph) {
}

RealGraph::~RealGraph() {
}

void RealGraph::LoadRealGraph(const std::string& links) {
  const int MAX_BUFFER_SIZE = 1000000; // Max input line length in the file
  std::string str;
  std::set<int> uniq_ids;
  std::vector<std::pair<int, int> > id_edges;
  std::ifstream links_ifs(links.c_str());
  char buffer[MAX_BUFFER_SIZE];

  // Read input file and save all the ids in the uniq_ids set
  // and all the edges in the id_edges vector. Skip host names and -1 ids
  while (!links_ifs.eof()) {
    getline(links_ifs, str);
    if (links_ifs.good()) {
      const char* line = strchr(strchr(str.c_str(), ' ') + 1, ' ') + 1;
      strcpy(buffer, line);
      int id_in = atoi(strtok(buffer, " \n"));
      if (id_in != -1) {
        char* id_next = NULL;
        while (id_next = strtok(NULL, " \n")) {
          int id_out = atoi(id_next);
```

```
            uniq_ids.insert(id_in);
            uniq_ids.insert(id_out);
            id_edges.push_back(std::make_pair(id_out, id_in));
          }
        }
      }
    }

    // Renumber all the ids in 0 to vertex count
    std::map<int, int> ids;
    int vertex_count = 0;
    for (std::set<int>::iterator id = uniq_ids.begin();
         id != uniq_ids.end(); ++id) {
      ids[*id] = vertex_count;
      ++vertex_count;
    }

    // Set vertex count and edges with renumbered vertices
    SetVertexCount(vertex_count);
    for (std::vector<std::pair<int, int> >::iterator edge = id_edges.begin();
         edge != id_edges.end(); ++edge) {
      int vertex_out = ids[edge->first];
      int vertex_in = ids[edge->second];
      AddEdge(vertex_out, vertex_in);
    }
  }
```

```
/*** main.cpp ***/
#include "simulated_graph.h"
#include "real_graph.h"
#include <cmath>
#include <cstdlib>
#include <iostream>

const int SIMULATED_TIME = 100000;
const int RANDOM_SEED = 729531;
const int REPEATINGS = 3;
const double EPS = 1e-3;

// Model parameters to choose from
const double alpha[] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
const double beta[] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
const double delta_in[] = {10, 20, 30, 40, 50};

int main() {
  // Set the random seed
  srand(RANDOM_SEED);

  // Load real graph from the tt3.0 input file
  RealGraph real_graph;
  real_graph.LoadRealGraph("tt3.0");

  // Estimate real graph xi
  double real_graph_xi = real_graph.EstimateXi();
  std::cout << "real_graph_xi = " << real_graph_xi << std::endl;

  // Prepare variables for best parameter values
  double difference_in_xi = -1.0;
  double best_alpha = -1.0;
  double best_beta = -1.0;
  double best_delta_in = -1.0;
  double best_simulated_graph_xi = -1.0;
  double best_simulated_graph_variance_xi = -1.0;

  // Loop by parameter values
```

```
for (int a_i = 0; a_i < sizeof(alpha) / sizeof(alpha[0]); ++a_i) {
  for (int b_i = 0; b_i < sizeof(beta) / sizeof(beta[0]); ++b_i) {
    if (alpha[a_i] + beta[b_i] < 1.0 - EPS) {
      for (int d_i = 0; d_i < sizeof(delta_in) / sizeof(delta_in[0]); ++d_i) {
        // Prepare mean xi and variance of xi
        double simulated_graph_mean_xi = 0.0;
        double simulated_graph_variance_xi = 0.0;

        // Generate simulated graph REPEATINGS times and take the mean xi
        for (int index = 0; index < REPEATINGS; ++index) {
          // Create simulated graph with the chosen parameter values
          SimulatedGraph simulated_graph;
          simulated_graph.SetAlpha(alpha[a_i]);
          simulated_graph.SetBeta(beta[b_i]);
          simulated_graph.SetDeltaIn(delta_in[d_i]);
          simulated_graph.GenerateGraph(SIMULATED_TIME);

          // Estimate simulated graph xi, mean xi and variance of xi
          double simulated_graph_xi = simulated_graph.EstimateXi();
          simulated_graph_mean_xi += simulated_graph_xi;
          simulated_graph_variance_xi +=
              simulated_graph_xi * simulated_graph_xi;
        }

        // Evaluate mean and variance of xi
        simulated_graph_mean_xi /= REPEATINGS;
        simulated_graph_variance_xi /= REPEATINGS;
        simulated_graph_variance_xi -=
            simulated_graph_mean_xi * simulated_graph_mean_xi;

        // Save parameter values for the best case
        // (difference in real graph xi and simulated graph xi is minimal)
        if (difference_in_xi < 0 ||
            fabs(real_graph_xi - simulated_graph_mean_xi) <
            difference_in_xi) {
          difference_in_xi =
              fabs(real_graph_xi - simulated_graph_mean_xi);
          best_alpha = alpha[a_i];
```

```
          best_beta = beta[b_i];
          best_delta_in = delta_in[d_i];
          best_simulated_graph_xi = simulated_graph_mean_xi;
          best_simulated_graph_variance_xi = simulated_graph_variance_xi;
        }

        // Output current results
        std::cout << "alpha = " << alpha[a_i]
                  << ", beta = " << beta[b_i]
                  << ", delta_in = " << delta_in[d_i]
                  << ": simulated_graph_xi = " << simulated_graph_mean_xi
                  << " +- " << simulated_graph_variance_xi
                  << std::endl;
      }
    }
  }
}

// Output best results
std::cout << "Best parameters:" << std::endl
          << "alpha = " << best_alpha
          << ", beta = " << best_beta
          << ", delta_in = " << best_delta_in
          << ": simulated_graph_xi = " << best_simulated_graph_xi
          << ": +- " << best_simulated_graph_variance_xi
          << std::endl
          << "Difference in xi = " << difference_in_xi << std::endl;

// Success
return 0;
}
```

Приложение В

Результат работы программы

```
real_graph_xi = 1.28909
alpha = 0.1, beta = 0.1, delta_in = 10: simulated_graph_xi = 0.932771 +- 0.00728888
alpha = 0.1, beta = 0.1, delta_in = 20: simulated_graph_xi = 0.974909 +- 0.000788533
alpha = 0.1, beta = 0.1, delta_in = 30: simulated_graph_xi = 0.969449 +- 0.000961652
alpha = 0.1, beta = 0.1, delta_in = 40: simulated_graph_xi = 0.961931 +- 0.000463594
alpha = 0.1, beta = 0.1, delta_in = 50: simulated_graph_xi = 0.931287 +- 0.000423722
alpha = 0.1, beta = 0.2, delta_in = 10: simulated_graph_xi = 0.99747 +- 0.000663344
alpha = 0.1, beta = 0.2, delta_in = 20: simulated_graph_xi = 0.97063 +- 0.00198001
alpha = 0.1, beta = 0.2, delta_in = 30: simulated_graph_xi = 0.94963 +- 2.00138e-06
alpha = 0.1, beta = 0.2, delta_in = 40: simulated_graph_xi = 1.08471 +- 0.00390443
alpha = 0.1, beta = 0.2, delta_in = 50: simulated_graph_xi = 1.02931 +- 0.000443945
alpha = 0.1, beta = 0.3, delta_in = 10: simulated_graph_xi = 1.01148 +- 0.00112183
alpha = 0.1, beta = 0.3, delta_in = 20: simulated_graph_xi = 0.994045 +- 0.000407881
alpha = 0.1, beta = 0.3, delta_in = 30: simulated_graph_xi = 1.07286 +- 0.000944741
alpha = 0.1, beta = 0.3, delta_in = 40: simulated_graph_xi = 1.00004 +- 0.00199681
alpha = 0.1, beta = 0.3, delta_in = 50: simulated_graph_xi = 0.975628 +- 0.00530568
alpha = 0.1, beta = 0.4, delta_in = 10: simulated_graph_xi = 1.06259 +- 0.0029934
alpha = 0.1, beta = 0.4, delta_in = 20: simulated_graph_xi = 1.06301 +- 0.00134939
alpha = 0.1, beta = 0.4, delta_in = 30: simulated_graph_xi = 1.02225 +- 0.000126111
alpha = 0.1, beta = 0.4, delta_in = 40: simulated_graph_xi = 1.09805 +- 0.000911361
alpha = 0.1, beta = 0.4, delta_in = 50: simulated_graph_xi = 1.06724 +- 0.000594735
alpha = 0.1, beta = 0.5, delta_in = 10: simulated_graph_xi = 1.10449 +- 6.61492e-05
alpha = 0.1, beta = 0.5, delta_in = 20: simulated_graph_xi = 1.09398 +- 1.02478e-05
alpha = 0.1, beta = 0.5, delta_in = 30: simulated_graph_xi = 1.10271 +- 0.000873227
alpha = 0.1, beta = 0.5, delta_in = 40: simulated_graph_xi = 1.03951 +- 0.000933735
alpha = 0.1, beta = 0.5, delta_in = 50: simulated_graph_xi = 1.08029 +- 0.00161058
```

```
alpha = 0.1, beta = 0.6, delta_in = 10: simulated_graph_xi = 1.20919 +- 0.000494145
alpha = 0.1, beta = 0.6, delta_in = 20: simulated_graph_xi = 1.14218 +- 0.000446993
alpha = 0.1, beta = 0.6, delta_in = 30: simulated_graph_xi = 1.1227 +- 0.00196923
alpha = 0.1, beta = 0.6, delta_in = 40: simulated_graph_xi = 1.14014 +- 0.00798361
alpha = 0.1, beta = 0.6, delta_in = 50: simulated_graph_xi = 1.13292 +- 0.00204208
alpha = 0.1, beta = 0.7, delta_in = 10: simulated_graph_xi = 1.23367 +- 0.00132783
alpha = 0.1, beta = 0.7, delta_in = 20: simulated_graph_xi = 1.18782 +- 0.00428949
alpha = 0.1, beta = 0.7, delta_in = 30: simulated_graph_xi = 1.27585 +- 0.00275136
alpha = 0.1, beta = 0.7, delta_in = 40: simulated_graph_xi = 1.22912 +- 0.00039264
alpha = 0.1, beta = 0.7, delta_in = 50: simulated_graph_xi = 1.17065 +- 0.00339632
alpha = 0.1, beta = 0.8, delta_in = 10: simulated_graph_xi = 1.26342 +- 0.00249616
alpha = 0.1, beta = 0.8, delta_in = 20: simulated_graph_xi = 1.29528 +- 0.00196035
alpha = 0.1, beta = 0.8, delta_in = 30: simulated_graph_xi = 1.25078 +- 0.00301471
alpha = 0.1, beta = 0.8, delta_in = 40: simulated_graph_xi = 1.23568 +- 0.00087426
alpha = 0.1, beta = 0.8, delta_in = 50: simulated_graph_xi = 1.25477 +- 0.000962454
alpha = 0.2, beta = 0.1, delta_in = 10: simulated_graph_xi = 1.23841 +- 0.000770949
alpha = 0.2, beta = 0.1, delta_in = 20: simulated_graph_xi = 1.19279 +- 0.000768557
alpha = 0.2, beta = 0.1, delta_in = 30: simulated_graph_xi = 1.189 +- 0.000427635
alpha = 0.2, beta = 0.1, delta_in = 40: simulated_graph_xi = 1.17462 +- 0.000312942
alpha = 0.2, beta = 0.1, delta_in = 50: simulated_graph_xi = 1.22531 +- 0.00285168
alpha = 0.2, beta = 0.2, delta_in = 10: simulated_graph_xi = 1.27213 +- 0.00168208
alpha = 0.2, beta = 0.2, delta_in = 20: simulated_graph_xi = 1.26298 +- 7.07486e-05
alpha = 0.2, beta = 0.2, delta_in = 30: simulated_graph_xi = 1.2835 +- 0.00280229
alpha = 0.2, beta = 0.2, delta_in = 40: simulated_graph_xi = 1.26348 +- 0.000511625
alpha = 0.2, beta = 0.2, delta_in = 50: simulated_graph_xi = 1.20518 +- 0.00345746
alpha = 0.2, beta = 0.3, delta_in = 10: simulated_graph_xi = 1.23412 +- 0.00104592
alpha = 0.2, beta = 0.3, delta_in = 20: simulated_graph_xi = 1.2875 +- 0.000527015
alpha = 0.2, beta = 0.3, delta_in = 30: simulated_graph_xi = 1.28566 +- 0.00430134
alpha = 0.2, beta = 0.3, delta_in = 40: simulated_graph_xi = 1.22661 +- 0.000687818
alpha = 0.2, beta = 0.3, delta_in = 50: simulated_graph_xi = 1.28648 +- 0.000353701
alpha = 0.2, beta = 0.4, delta_in = 10: simulated_graph_xi = 1.30134 +- 0.000357783
alpha = 0.2, beta = 0.4, delta_in = 20: simulated_graph_xi = 1.29683 +- 0.000250697
alpha = 0.2, beta = 0.4, delta_in = 30: simulated_graph_xi = 1.33242 +- 0.00190293
alpha = 0.2, beta = 0.4, delta_in = 40: simulated_graph_xi = 1.30397 +- 3.82383e-06
alpha = 0.2, beta = 0.4, delta_in = 50: simulated_graph_xi = 1.28801 +- 0.00389082
alpha = 0.2, beta = 0.5, delta_in = 10: simulated_graph_xi = 1.36286 +- 0.000763779
alpha = 0.2, beta = 0.5, delta_in = 20: simulated_graph_xi = 1.30001 +- 0.00136646
alpha = 0.2, beta = 0.5, delta_in = 30: simulated_graph_xi = 1.33997 +- 0.00108664
alpha = 0.2, beta = 0.5, delta_in = 40: simulated_graph_xi = 1.37349 +- 0.0039013
alpha = 0.2, beta = 0.5, delta_in = 50: simulated_graph_xi = 1.33394 +- 0.000192949
alpha = 0.2, beta = 0.6, delta_in = 10: simulated_graph_xi = 1.4149 +- 0.00720377
alpha = 0.2, beta = 0.6, delta_in = 20: simulated_graph_xi = 1.42447 +- 0.00134053
alpha = 0.2, beta = 0.6, delta_in = 30: simulated_graph_xi = 1.38158 +- 0.000679881
alpha = 0.2, beta = 0.6, delta_in = 40: simulated_graph_xi = 1.39088 +- 0.000561193
alpha = 0.2, beta = 0.6, delta_in = 50: simulated_graph_xi = 1.34371 +- 0.000489887
alpha = 0.2, beta = 0.7, delta_in = 10: simulated_graph_xi = 1.43066 +- 0.000642419
alpha = 0.2, beta = 0.7, delta_in = 20: simulated_graph_xi = 1.46199 +- 8.89685e-05
```

```
alpha = 0.2, beta = 0.7, delta_in = 30: simulated_graph_xi = 1.46693 +- 0.00109336
alpha = 0.2, beta = 0.7, delta_in = 40: simulated_graph_xi = 1.44535 +- 0.00547142
alpha = 0.2, beta = 0.7, delta_in = 50: simulated_graph_xi = 1.47933 +- 0.00261129
alpha = 0.3, beta = 0.1, delta_in = 10: simulated_graph_xi = 1.44605 +- 0.00110126
alpha = 0.3, beta = 0.1, delta_in = 20: simulated_graph_xi = 1.49866 +- 0.00207915
alpha = 0.3, beta = 0.1, delta_in = 30: simulated_graph_xi = 1.4928 +- 0.00290065
alpha = 0.3, beta = 0.1, delta_in = 40: simulated_graph_xi = 1.47628 +- 0.00233522
alpha = 0.3, beta = 0.1, delta_in = 50: simulated_graph_xi = 1.47639 +- 0.00105536
alpha = 0.3, beta = 0.2, delta_in = 10: simulated_graph_xi = 1.5047 +- 0.00204238
alpha = 0.3, beta = 0.2, delta_in = 20: simulated_graph_xi = 1.51964 +- 0.000521645
alpha = 0.3, beta = 0.2, delta_in = 30: simulated_graph_xi = 1.48895 +- 0.00135505
alpha = 0.3, beta = 0.2, delta_in = 40: simulated_graph_xi = 1.53566 +- 0.000545929
alpha = 0.3, beta = 0.2, delta_in = 50: simulated_graph_xi = 1.47804 +- 0.00143119
alpha = 0.3, beta = 0.3, delta_in = 10: simulated_graph_xi = 1.52457 +- 0.000237678
alpha = 0.3, beta = 0.3, delta_in = 20: simulated_graph_xi = 1.52941 +- 0.000652224
alpha = 0.3, beta = 0.3, delta_in = 30: simulated_graph_xi = 1.51453 +- 0.00176311
alpha = 0.3, beta = 0.3, delta_in = 40: simulated_graph_xi = 1.55945 +- 6.87918e-05
alpha = 0.3, beta = 0.3, delta_in = 50: simulated_graph_xi = 1.53562 +- 0.00641355
alpha = 0.3, beta = 0.4, delta_in = 10: simulated_graph_xi = 1.60746 +- 0.000783873
alpha = 0.3, beta = 0.4, delta_in = 20: simulated_graph_xi = 1.53671 +- 0.00185931
alpha = 0.3, beta = 0.4, delta_in = 30: simulated_graph_xi = 1.59394 +- 0.00932542
alpha = 0.3, beta = 0.4, delta_in = 40: simulated_graph_xi = 1.56606 +- 0.000836987
alpha = 0.3, beta = 0.4, delta_in = 50: simulated_graph_xi = 1.56114 +- 0.00178809
alpha = 0.3, beta = 0.5, delta_in = 10: simulated_graph_xi = 1.65333 +- 0.00334823
alpha = 0.3, beta = 0.5, delta_in = 20: simulated_graph_xi = 1.63995 +- 0.0030323
alpha = 0.3, beta = 0.5, delta_in = 30: simulated_graph_xi = 1.67205 +- 0.00117298
alpha = 0.3, beta = 0.5, delta_in = 40: simulated_graph_xi = 1.65376 +- 2.5201e-05
alpha = 0.3, beta = 0.5, delta_in = 50: simulated_graph_xi = 1.608 +- 0.000610703
alpha = 0.3, beta = 0.6, delta_in = 10: simulated_graph_xi = 1.71611 +- 0.000653836
alpha = 0.3, beta = 0.6, delta_in = 20: simulated_graph_xi = 1.67483 +- 0.00165382
alpha = 0.3, beta = 0.6, delta_in = 30: simulated_graph_xi = 1.67222 +- 0.00446685
alpha = 0.3, beta = 0.6, delta_in = 40: simulated_graph_xi = 1.67199 +- 0.000164499
alpha = 0.3, beta = 0.6, delta_in = 50: simulated_graph_xi = 1.70552 +- 0.000339063
alpha = 0.4, beta = 0.1, delta_in = 10: simulated_graph_xi = 1.85665 +- 0.000804998
alpha = 0.4, beta = 0.1, delta_in = 20: simulated_graph_xi = 1.82345 +- 0.00610725
alpha = 0.4, beta = 0.1, delta_in = 30: simulated_graph_xi = 1.78459 +- 0.000654807
alpha = 0.4, beta = 0.1, delta_in = 40: simulated_graph_xi = 1.81684 +- 0.00544487
alpha = 0.4, beta = 0.1, delta_in = 50: simulated_graph_xi = 1.82816 +- 0.00448046
alpha = 0.4, beta = 0.2, delta_in = 10: simulated_graph_xi = 1.84631 +- 0.00114318
alpha = 0.4, beta = 0.2, delta_in = 20: simulated_graph_xi = 1.84171 +- 0.00335591
alpha = 0.4, beta = 0.2, delta_in = 30: simulated_graph_xi = 1.83696 +- 0.000794708
alpha = 0.4, beta = 0.2, delta_in = 40: simulated_graph_xi = 1.80721 +- 0.000486551
alpha = 0.4, beta = 0.2, delta_in = 50: simulated_graph_xi = 1.81096 +- 0.00400177
alpha = 0.4, beta = 0.3, delta_in = 10: simulated_graph_xi = 1.88461 +- 0.00351277
alpha = 0.4, beta = 0.3, delta_in = 20: simulated_graph_xi = 1.91742 +- 0.00201697
alpha = 0.4, beta = 0.3, delta_in = 30: simulated_graph_xi = 1.86504 +- 0.00102953
alpha = 0.4, beta = 0.3, delta_in = 40: simulated_graph_xi = 1.86436 +- 0.000598827
```

```
alpha = 0.4, beta = 0.3, delta_in = 50: simulated_graph_xi = 1.87625 +- 0.000134616
alpha = 0.4, beta = 0.4, delta_in = 10: simulated_graph_xi = 1.9842 +- 0.0052373
alpha = 0.4, beta = 0.4, delta_in = 20: simulated_graph_xi = 1.93133 +- 0.000686224
alpha = 0.4, beta = 0.4, delta_in = 30: simulated_graph_xi = 1.93827 +- 0.000770509
alpha = 0.4, beta = 0.4, delta_in = 40: simulated_graph_xi = 1.91178 +- 0.00226971
alpha = 0.4, beta = 0.4, delta_in = 50: simulated_graph_xi = 1.92069 +- 0.00348812
alpha = 0.4, beta = 0.5, delta_in = 10: simulated_graph_xi = 1.96767 +- 0.00120533
alpha = 0.4, beta = 0.5, delta_in = 20: simulated_graph_xi = 1.9868 +- 0.00369677
alpha = 0.4, beta = 0.5, delta_in = 30: simulated_graph_xi = 1.94129 +- 0.00209144
alpha = 0.4, beta = 0.5, delta_in = 40: simulated_graph_xi = 1.99606 +- 0.00168267
alpha = 0.4, beta = 0.5, delta_in = 50: simulated_graph_xi = 1.96347 +- 0.00595658
alpha = 0.5, beta = 0.1, delta_in = 10: simulated_graph_xi = 2.33274 +- 0.00494496
alpha = 0.5, beta = 0.1, delta_in = 20: simulated_graph_xi = 2.21657 +- 0.0109728
alpha = 0.5, beta = 0.1, delta_in = 30: simulated_graph_xi = 2.25624 +- 0.00343628
alpha = 0.5, beta = 0.1, delta_in = 40: simulated_graph_xi = 2.24872 +- 0.00235612
alpha = 0.5, beta = 0.1, delta_in = 50: simulated_graph_xi = 2.23092 +- 0.00780588
alpha = 0.5, beta = 0.2, delta_in = 10: simulated_graph_xi = 2.32951 +- 0.00889997
alpha = 0.5, beta = 0.2, delta_in = 20: simulated_graph_xi = 2.26997 +- 0.00380407
alpha = 0.5, beta = 0.2, delta_in = 30: simulated_graph_xi = 2.31947 +- 0.00044122
alpha = 0.5, beta = 0.2, delta_in = 40: simulated_graph_xi = 2.28689 +- 0.00363055
alpha = 0.5, beta = 0.2, delta_in = 50: simulated_graph_xi = 2.31509 +- 0.000536671
alpha = 0.5, beta = 0.3, delta_in = 10: simulated_graph_xi = 2.31756 +- 0.00153393
alpha = 0.5, beta = 0.3, delta_in = 20: simulated_graph_xi = 2.32366 +- 0.000276932
alpha = 0.5, beta = 0.3, delta_in = 30: simulated_graph_xi = 2.38659 +- 0.00256113
alpha = 0.5, beta = 0.3, delta_in = 40: simulated_graph_xi = 2.24827 +- 0.00450243
alpha = 0.5, beta = 0.3, delta_in = 50: simulated_graph_xi = 2.38144 +- 0.0081087
alpha = 0.5, beta = 0.4, delta_in = 10: simulated_graph_xi = 2.34105 +- 0.000838409
alpha = 0.5, beta = 0.4, delta_in = 20: simulated_graph_xi = 2.46757 +- 0.00180566
alpha = 0.5, beta = 0.4, delta_in = 30: simulated_graph_xi = 2.40895 +- 0.000978078
alpha = 0.5, beta = 0.4, delta_in = 40: simulated_graph_xi = 2.45435 +- 0.000863965
alpha = 0.5, beta = 0.4, delta_in = 50: simulated_graph_xi = 2.40079 +- 0.0028487
alpha = 0.6, beta = 0.1, delta_in = 10: simulated_graph_xi = 2.83816 +- 0.0010161
alpha = 0.6, beta = 0.1, delta_in = 20: simulated_graph_xi = 2.86843 +- 0.00103467
alpha = 0.6, beta = 0.1, delta_in = 30: simulated_graph_xi = 2.80655 +- 0.0119543
alpha = 0.6, beta = 0.1, delta_in = 40: simulated_graph_xi = 2.83118 +- 0.0092681
alpha = 0.6, beta = 0.1, delta_in = 50: simulated_graph_xi = 2.82788 +- 0.00852022
alpha = 0.6, beta = 0.2, delta_in = 10: simulated_graph_xi = 2.80257 +- 0.00680838
alpha = 0.6, beta = 0.2, delta_in = 20: simulated_graph_xi = 2.77229 +- 0.00907522
alpha = 0.6, beta = 0.2, delta_in = 30: simulated_graph_xi = 2.90865 +- 0.00321538
alpha = 0.6, beta = 0.2, delta_in = 40: simulated_graph_xi = 2.8512 +- 0.000240225
alpha = 0.6, beta = 0.2, delta_in = 50: simulated_graph_xi = 2.74585 +- 0.00341822
alpha = 0.6, beta = 0.3, delta_in = 10: simulated_graph_xi = 2.85822 +- 0.00700646
alpha = 0.6, beta = 0.3, delta_in = 20: simulated_graph_xi = 2.88147 +- 0.00071349
alpha = 0.6, beta = 0.3, delta_in = 30: simulated_graph_xi = 2.7895 +- 0.00457419
alpha = 0.6, beta = 0.3, delta_in = 40: simulated_graph_xi = 3.01195 +- 0.00406189
alpha = 0.6, beta = 0.3, delta_in = 50: simulated_graph_xi = 3.00996 +- 0.000130839
alpha = 0.7, beta = 0.1, delta_in = 10: simulated_graph_xi = 3.43458 +- 0.0102724
```

```
alpha = 0.7, beta = 0.1, delta_in = 20: simulated_graph_xi = 3.4369 +- 0.03133
alpha = 0.7, beta = 0.1, delta_in = 30: simulated_graph_xi = 3.48409 +- 0.0210833
alpha = 0.7, beta = 0.1, delta_in = 40: simulated_graph_xi = 3.63702 +- 0.00643128
alpha = 0.7, beta = 0.1, delta_in = 50: simulated_graph_xi = 3.64236 +- 0.019093
alpha = 0.7, beta = 0.2, delta_in = 10: simulated_graph_xi = 3.51785 +- 0.000243311
alpha = 0.7, beta = 0.2, delta_in = 20: simulated_graph_xi = 3.41559 +- 0.0126742
alpha = 0.7, beta = 0.2, delta_in = 30: simulated_graph_xi = 3.52958 +- 0.00600853
alpha = 0.7, beta = 0.2, delta_in = 40: simulated_graph_xi = 3.47119 +- 0.0235151
alpha = 0.7, beta = 0.2, delta_in = 50: simulated_graph_xi = 3.44423 +- 0.013753
alpha = 0.8, beta = 0.1, delta_in = 10: simulated_graph_xi = 3.95465 +- 0.00389398
alpha = 0.8, beta = 0.1, delta_in = 20: simulated_graph_xi = 4.07212 +- 0.0165141
alpha = 0.8, beta = 0.1, delta_in = 30: simulated_graph_xi = 4.24585 +- 0.00697883
alpha = 0.8, beta = 0.1, delta_in = 40: simulated_graph_xi = 4.16692 +- 0.00618292
alpha = 0.8, beta = 0.1, delta_in = 50: simulated_graph_xi = 4.2845 +- 0.00206282
Best parameters:
alpha = 0.2, beta = 0.4, delta_in = 50: simulated_graph_xi = 1.28801: +- 0.00389082
Difference in xi = 0.00108132
```