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. Нас будет интересовать лишь одно направление, которое становится все более актуальным с каждым годом. Это графы, которые изучаются с вероятностной точки зрения. Случайные графы, которые описывают рост различных сетей, – социальных, биологических, транспортных – наиболее современны в этом направлении. В первую очередь, это связано, конечно же, с Интернетом.

Глава 2

Модели случайных веб-графов

Модели случайных веб-графов 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 $cd^{-\gamma}$, где $\gamma = 2.1$. Интересно, что этот закон характерен для всех реальных сетей, но у каждой из них своя γ.

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 степени вершины.

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 ребра 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, $n = 1, 2, \dots$, 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:

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

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 $1, \dots, m$ в вершину 1 нового графа, объединением вершин $m+1, \dots, 2m$ в вершину 2 нового графа и так далее. Замети, 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 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 $k \geq 2$ и любого $\epsilon > 0$*

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

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 $k \geq 1$ и любого $d \leq n^{(\frac{1}{15})}$*

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

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 $\gamma_{www} = 2.1$. 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 γ результирующего графа.

Более строго, модель генерирует набор графов $H^n_{a,m}, n = 1, 2, \dots,$ с n вершинами и mn ребрами, где $m \in \mathbb{Z}$ – фиксированное число. Определение $H^n_{a,1}$ повторяет определение G^n_1 с одним отличием, заключающимся в том, что вероятность нового ребра, добавляемого в $H^n_{a,1}$ равна

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

Граф $H^n_{a,m}$ получается из графа $H^{mn}_{a,1}$ так же, как и G^n_m получается из G^{mn}_1. Заметим, что при $a = 1$ мы получаем изначальную модель Боллобаша-Риордана G^n_m. Для целых a Бакли и Остхус доказали, что распределение степеней вершин случайного графа в модели соответствует степенному закону с $\gamma = 2 + a$.

2.7 Модель Боллобаша–Боргса–Риордана–Чайес

В данной модели 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 $t_0 \geq 1$, 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 $Pr(v = v_i)$ пропорциональна $d_{out}(v_i) + \delta_{out}$, то есть таким образом, что $Pr(v = v_i) = (d_{out}(v_i) + \delta_{out})/(t + \delta_{out}n(t))$. Аналогично, под фразой «*выбрать v в соответствии с $d_{in} + \delta_{in}$*» будем понимать, что вершина v выбирается таким образом, что $Pr(v = v_i) = (d_{in}(v_i) + \delta_{in})/(t + \delta_{in}n(t))$. Здесь $d_{out}(v_i)$ и $d_{in}(v_i)$ – исходящая и входящая степени вершины v_i в графе $G(t)$.

Для $t \geq t_0$ 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 вероятсность 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 существу-

ющей 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 $\delta_{in} = 0$ на ряду 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-Альберт. Ее авторами являются Р. Кумар, П. Рагхаван, С. Раджагопалан, Д. Сивакумар, А. Томкинс и Э. Упфал.

Зафиксируем $\alpha \in (0,1)$ и $d \geq 1, d \in \mathbb{N}$. Граф будем строить итеративно, в качества 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 $V_t = \{u_1, \ldots, u_s\}$, 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

Previous work

A much longer $\LaTeX\,2_\varepsilon$ example was written by Gil [**?**].

Глава 4

Результаты

In this section we describe the results.

4.1 Выводы

We worked hard, and achieved very little.

Литература

[1] Степанов В. Е. О вероятности связности случайного графа $g_m(t)$ // Теория вероятностей и ее применения. 1970. Т. 15. № 1. С. 55–67.

[2] Степанов В. Е. Фазовый переход в случайных графах // Теория вероятностей и ее применения. 1970. Т. 15. № 2. С. 187–203.

[3] Степанов В. Е. Структура случайных графов $g_n(x|h)$ // Теория вероятностей и ее применения. 1972. Т. 17. № 3. С. 227–242.

[4] Колчин В. Ф. Случайные графы. М.: Физматлит, 2004.

[5] Bollobas B. Random Graphs. Cambridge: Cambridge Univ. Press, 2001.

[6] Алон Н., Спенсер Дж. Вероятностный метод. М: Бином. Лаборатория знаний, 2007.

[7] Janson S., Luczak T., Rucinski A. Random graphs. N.Y.: Wiley, 2000.

[8] Маргулис Г. А. Вероятностные характеристики графов с большой связностью // Проблемы передачи информации. 1974. Т. 10. С. 101–108.

[9] Karp R. The transitive closure of a random digraph // Random structures and algorithms. 1990. V. 1. P. 73–94.

[10] Карлин С. Основы теории случайных процессов. М: Мир, 1971.

[11] Barabasi L.-A., Albert R. Emergence of scaling in random networks // Science. 1999. V. 286. P. 509–512.

[12] Barabasi L.-A., Albert R., Jeong H. Scale-free characteristics of random networks: the topology of the world-wide web // Physica A. 2000. V. 281. P. 69–77.

[13] Albert R., Jeong H., Barabasi L. A. Diameter of the world-wide web // Nature. 1999. V. 401. P. 130–131.

[14] Bollobas B., Riordan O. Mathematical results on scale-free random graphs. Handbook of graphs and networks. Weinheim: Wiley-VCH. 2003. P. 1–34.

[15] Райгородский А. М. Экстремальные задачи теории графов и анализ данных. М.–Ижевск: НИЦ «РХД», 2009.

[16] Stoimenow A. Enumeration of chord diagrams and an upper bound for Vassiliev invariants // J. Knot Theory Ramifications. 1998. V. 7. N. 1. P. 93–114.

[17] Bollobas B., Riordan O. The diameter of a scale-free random graph // Combinatorica. 2004. V. 24. N. 1. P. 5–34.

[18] Bollobas B., Riordan O., Spencer J., Tusnady G. The degree sequence of a scale-free random graph process // Random Structures Algorithms. 2001. V. 18. N. 3. P. 279–290.

[19] Kumar R., Raghavan P., Rajagopalan S., Sivakumar D., Tomkins A., Upfal E. Stochastic models for the web graph // Proc. 41st Symposium on Foundations of Computer Science. 2000.

Приложение A

Исходный код программы

```
/*** convert_graph.cpp ***/
#include <cstring>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <set>

const int MAX_BUFFER_SIZE = 1000000;

int main() {
  std::map<int, int> ids;
  std::ifstream index_ifs("id_index.txt");
  std::string str;
  int index = 0;
  while (!index_ifs.eof()) {
    getline(index_ifs, str);
    if (index_ifs.good()) {
      std::stringstream ss;
      ss << str;
      int id;
      ss >> id;
      ids[id] = index;
```

```
      ++index;
    }
  }
  std::set<int> uniq_ids;
  std::ifstream links_ifs("links.txt");
  char buffer[MAX_BUFFER_SIZE];
  while (!links_ifs.eof()) {
    getline(links_ifs, str);
    if (links_ifs.good()) {
      strcpy(buffer, str.c_str());
      int id_one = atoi(strtok(buffer, " \n"));
      std::map<int, int>::iterator vertex_in_iter = ids.find(id_one);
      if (vertex_in_iter != ids.end()) {
        int vertex_in = vertex_in_iter->second;
        char* id_next = NULL;
        while (id_next = strtok(NULL, " \n")) {
          int id_two = atoi(id_next);
          std::map<int, int>::iterator vertex_out_iter = ids.find(id_two);
          if (vertex_out_iter != ids.end()) {
            uniq_ids.insert(id_one);
            uniq_ids.insert(id_two);
            int vertex_out = vertex_out_iter->second;
            std::cout << vertex_out << ' ' << vertex_in << std::endl;
          }
        }
      }
    }
  }
  for (std::set<int>::iterator i = uniq_ids.begin(); i != uniq_ids.end(); ++i) {
    std::cerr << (*i) << std::endl;
  }
  return 0;
}
```

```
#/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 EstimateGamma();
 protected:
  int vertex;
  std::set<std::pair<int, int> > edges;
};

#endif
```

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

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::EstimateGamma() {
  // Evaluate vertex degrees
  std::vector<int> vertex_degree(GetVertexCount(), 0);
  //std::vector<int> vertex_in_degree(GetVertexCount(), 0);
  //std::vector<int> vertex_out_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];
  //++vertex_out_degree[edge->first];
  //++vertex_in_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];
  }
}
std::vector<double> probability;
for (int index = 0; index < vertex_degree.size(); ++index) {
  double P = static_cast<double>(count_vertex_degree[vertex_degree[index]]);
  P /= GetVertexCount();
  probability.push_back(P);
}

// P = c * d^(-gamma)
// ln(P) = ln(c) - gamma * ln(d)
// y_t = a + b * x_t + epsilon_t
// y_t = ln(P)
// a = ln(c)
// b = gamma
// x_t = -ln(d)
double xy = 0;
double x = 0;
double y = 0;
double xx = 0;
int count = 0;
for (int index = 0; index < vertex_degree.size(); ++index) {
  const double EPS = 0.000000001;
```

```
    if (vertex_degree[index] >= 150 && vertex_degree[index] <= 350 && probability[index]
      double dx = -log(static_cast<double>(vertex_degree[index]));
      double dy = log(probability[index]);
      xy += dx * dy;
      x += dx;
      y += dy;
      xx += dx * dx;
      ++count;
    }
  }

  xy /= count;
  x /= count;
  y /= count;
  xx /= count;
  double gamma = (xy - x * y) / (xx - x * x); // Ordinary least squares (OLS)
  return gamma;
}
```

```
/*** 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() {
  double random_point =
      static_cast<double>(rand()) / RAND_MAX * in_numerator.back();
  in_numerator[in_numerator.size() - 1] += 1.0;
  int current_vertex =
      std::upper_bound(in_numerator.begin(), in_numerator.end(), random_point)
      - in_numerator.begin();
  in_numerator[in_numerator.size() - 1] -= 1.0;
  return current_vertex;
}

int SimulatedGraph::ChooseVertexAccordingToOut() {
  double random_point =
      static_cast<double>(rand()) / RAND_MAX * out_numerator.back();
  out_numerator[out_numerator.size() - 1] += 1.0;
  int current_vertex =
      std::upper_bound(out_numerator.begin(), out_numerator.end(), random_point)
      - out_numerator.begin();
```

```
  out_numerator[out_numerator.size() - 1] -= 1.0;
  return current_vertex;
}

void SimulatedGraph::GenerateGraph(int time) {
  in_numerator.clear();
  out_numerator.clear();
  SetVertexCount(1);
  AddEdge(0, 0);
  in_numerator.push_back(1 + GetDeltaIn());
  out_numerator.push_back(1 + GetDeltaOut());
  for (int t = 1; t < time; ++t) {
    double random_point = static_cast<double>(rand()) / RAND_MAX;
    if (random_point <= GetAlpha()) {
      int existing_vertex = ChooseVertexAccordingToIn();
      int new_vertex = GetVertexCount();
      SetVertexCount(GetVertexCount() + 1);
      AddEdge(new_vertex, existing_vertex);
      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()) {
      int existing_vertex_one = ChooseVertexAccordingToOut();
      int existing_vertex_two = ChooseVertexAccordingToIn();
      AddEdge(existing_vertex_one, existing_vertex_two);
      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 {
      int existing_vertex = ChooseVertexAccordingToOut();
      int new_vertex = GetVertexCount();
```

```
        SetVertexCount(GetVertexCount() + 1);
        AddEdge(existing_vertex, new_vertex);
        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(int vertex_count, const std::string& links);
};

#endif
```

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

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

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

RealGraph::~RealGraph() {
}

void RealGraph::LoadRealGraph(int vertex_count, const std::string& links) {
  SetVertexCount(vertex_count);
  std::ifstream links_ifs(links.c_str());
  while (!links_ifs.eof()) {
    int vertex_one;
    int vertex_two;
    links_ifs >> vertex_one >> vertex_two;
    if (links_ifs.good()) {
      AddEdge(vertex_one, vertex_two);
    }
  }
}
```

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

const int REAL_VERTEX_COUNT = 538638;
const int SIMULATED_TIME = 10000;
const int RANDOM_SEED = 729531;

const double alpha[] = {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1};
const double beta[] = {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
const double delta_in[] = {0, 1, 10, 50, 100, 200, 500, 1000};

int main() {
  srand(RANDOM_SEED);
  RealGraph real_graph;
  real_graph.LoadRealGraph(REAL_VERTEX_COUNT, "converted_links.txt");
  double real_graph_gamma = real_graph.EstimateGamma();
  std::cout << "real_graph_gamma = " << real_graph_gamma << std::endl;
  return 0; // debug
  double difference_in_gamma = -1.0;
  double best_alpha = -1.0;
  double best_beta = -1.0;
  double best_delta_in = -1.0;
  double best_simulated_graph_gamma = -1.0;
  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) {
        for (int d_i = 0; d_i < sizeof(delta_in) / sizeof(delta_in[0]); ++d_i) {
          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);
          double simulated_graph_gamma = simulated_graph.EstimateGamma();
          if (difference_in_gamma < 0 ||
```

```
                fabs(real_graph_gamma - simulated_graph_gamma) <
                difference_in_gamma) {
              difference_in_gamma =
                  fabs(real_graph_gamma - simulated_graph_gamma);
              best_alpha = alpha[a_i];
              best_beta = beta[b_i];
              best_delta_in = delta_in[d_i];
              best_simulated_graph_gamma = simulated_graph_gamma;
            }
            std::cout << "alpha = " << alpha[a_i]
                      << ", beta = " << beta[b_i]
                      << ", delta_in = " << delta_in[d_i]
                      << ": simulated_graph_gamma = " << simulated_graph_gamma
                      << std::endl;
          }
        }
      }
    }
    std::cout << "Best parameters:" << std::endl
              << "alpha = " << best_alpha
              << ", beta = " << best_beta
              << ", delta_in = " << best_delta_in
              << ": simulated_graph_gamma = " << best_simulated_graph_gamma
              << std::endl
              << "Difference in gamma = " << difference_in_gamma << std::endl;
    return 0;
}
```