**Генератор модели копирования**

horizontal line

# Описание

Изначально фиксируем вероятность выпадения решки монеты P (0,1) и степень регулярности d начального графа. Пусть на итерации t построен граф с вершинами Vt. Добавим к нему новую вершину U. Для этого выбираем случайно вершину Vp из Vt. Далее соединяем новую вершину с вершинами из Vt через d рёбер по следующему алгоритму:

На каждой итерации подбрасываем монетку, и если выпадает решка с вероятностью P, то выпускаем ребро в случайную вершину из Vt. Если орёл с вероятностью 1-P, то выпускаем ребро в i-го соседа вершины Vp, которую мы выбрали изначально.

## Алгоритм

В аргументах функции main задаются параметры:  
Additional\_vertexes - сколько вершин нужно добавить в начальный граф  
Reg\_graph\_vertexes - сколько вершин в начальном графе  
*Degree* - степени регулярности начального графа  
Probability\_of\_coin\_toss - вероятность выпадения решки (соединение со случайной вершиной)

Вызывается функция graph\_generator\_copy, которая в свою очередь создаёт объект графа, генерирует в нем регулярный граф G.generate\_regular\_graph() и финальный граф G.generation(). Основной алгоритм в методе generation() описан вначале. Для хранения соседей вершин используется таблица std::map. Для поиска соседей можно было бы каждый раз пробегать массив рёбер, что было бы эффективно по памяти, но затратно по времени. Таблица занимает больше места, но поиск соседей очень быстрый.

Для создания регулярного графа используется программа [GenReg](http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html). Скачать программу можно [здесь](https://sourceforge.net/projects/genreg/), также есть [документация](http://www.mathe2.uni-bayreuth.de/markus/manual/genreg.html). Выход программы GenReg подаётся на вход генератору. Функция fillmap() заполняет таблицу соседей вершин, функция generate\_regular\_graph() заполняет массив рёбер. Например, команда ./GenReg 4 3 -a stdout | ./copy 250 4 3 0.8 создаёт регулярный граф с 4-мя вершинами и степенью регулярности 3, на основе которого создаётся граф с дополнительными 250-ю вершинами и вероятностью выпадения решки 0.8.

## Параллельность

Алгоритм итерационный, при добавлении каждой вершины требуются предыдущие вершины, поэтому внешний цикл метода generation() нельзя распараллелить. Внутренний цикл пробегает только d ребёр, поэтому накладные расходы будут выше, чем ускорение от параллельности. Можно добавить параллельность в generate\_regular\_graph(), но вершин в начальном регулярном графе мало, поэтому на тестах ускорение не заметно.

## Примеры Параметры компиляции: g++ -Wall Reny.cpp -o Reny -lcgraph -lgvc -fopenmp

./GenReg 16 7 -a stdout | ./copy 10 16 7 0.8 - 1\_copy.png (первая картинка в отчёте)  
./GenReg 16 7 -a stdout | ./copy 10 16 7 0.2 - 2\_copy.png  
./GenReg 16 7 -a stdout | ./copy 50 16 7 0.2 - 3\_copy.png (вторая картинка в отчёте)  
./GenReg 4 3 -a stdout | ./copy 25 4 3 0.2 - 4\_copy.png

На последнем графе видны начальные вершины, из которых выходит много рёбер. Как было сказано в [книге](https://mccme.ru/free-books/dubna/raigor-4.pdf) на стр. 128, данная модель имитирует появление новых сайтов в интернете. Вершины начального графа - это сайты, на которые либо ссылаются, либо с которых копируются данные.



