ДЗ №2 «Устойчивость сети»

Обходы графа (т.е. посещение вершин и ребер) являются важным инструментом для выявления свойств и анализа различных характеристик графов. Например, с помощью обхода мы можем рассчитать распределение входящих и исходящих степеней графа, найти попарные кратчайшие расстояния между вершинами, его компоненты связности и т.д.

В данном задании вам предстоит провести анализ устойчивости компьютерной сети, в то время как она подвергается кибер-атакам. В частности, мы промоделируем ситуацию, когда все возрастающее количество серверов сети выводится из строя. С математической точки зрения мы будем моделировать компьютерную сеть с помощью неориентированного графа и во время очередного эксперимента будем поочередно удалять из него узлы. В ходе экспериментов мы будем определять устойчивость сети как функцию, отражающую зависимость размера наибольшей компоненты связности от количества удаленных узлов.

Подзадача 1

Так как мы будем анализировать устойчивость неориентированных графов нескольких типов, то прежде всего напишите функции для генерации всех трех перечисленных ниже графов.

Тестовая модель компьютерной сети. Данная модель записана в текстовом файле (1347 узлов и 3112 ребер). Вам необходимо считать его в любую удобную для вас структуру.

ER-граф[1]. Случайный граф, генерируемый с помощью функции, алгоритм которой описан ниже. Функция принимает на вход два параметра: n – количество вершин в графе, p – вероятность того, что произвольное ребро (i,j) будет добавлено в граф. Обратите внимание, что в псевдокоде ниже в строке 3 запись {i,j} означает множество, а не список. Иными словами, если алгоритм обработает значения i=3, j=5, то он уже не будет обрабатывать значения i=5, j=3, так как на элементах множества нет порядка.

Algorithm 1: ER.

Input: Number of nodes n; probability p.
Output: A graph $g = (V, E)$ where $g \in G(n, p)$.

1 $V \leftarrow \{0, 1, \ldots, n-1\}$;
2 $E \leftarrow \emptyset$;
3 **foreach** $\{i, j\} \subseteq V$, *where* $i \neq j$ **do**
4 $a \leftarrow random(0, 1)$; // a is a random real number in $[0, 1)$
5 **if** $a < p$ **then**
6 $E \leftarrow E \cup \{\{i, j\}\}$;
7 **return** $g = (V, E)$;

UPA граф. Далее мы рассмотрим второй способ генерации случайного графа. Ниже приведено описание генерации **ориентированного** графа. Вам необходимо будет внести в него небольшие изменения, чтобы сделать генерацию **неориентированных** графов.

В этом способе граф будет конструироваться итеративно: на каждом шаге будем добавлять в граф новый узел и соединять его ребрами с некоторым подмножеством уже имеющихся вершин. Более формально: для генерации случайного ориентированного графа

[1] Модель Эрдёша-Реньи (венг. Erdős–Rényi) для генерации случайных графов.

необходимо задать два параметра: n – общее количество вершин в создаваемом графе, m (где m<n) – количество вершин, к которым подсоединяется каждая вновь созданная вершина. Заметим, что m фиксировано для всех итераций.

Алгоритм начинает с создания полносвязного ориентированного графа из m вершин. Далее к графу добавляется (n-m) вершин, причем каждая из них соединяется ребром с m случайно выбранными[2] вершинами, уже имеющимися в графе. Так как некоторая вершина может быть выбрана дважды на одной и той же итерации, то, чтобы избежать кратных ребер, алгоритм удаляет подобные дубликаты. Поэтому новая вершина в общем случае может иметь менее чем m соседей.

Полное описание алгоритма представлено ниже.

Algorithm 3: DPA.

Input: Number of nodes n ($n \geq 1$); integer m ($1 \leq m \leq n$).
Output: A directed graph $g = (V, E)$.

1 $V \leftarrow \{0, 1, \ldots, m-1\}$; // Start a graph on m nodes
2 $E \leftarrow \{(i,j) : i,j \in V, i \neq j\}$; // Make the graph complete
3 **for** $i \leftarrow m$ **to** $n-1$ **do**
4 $totindeg = \sum_{j \in V} indeg(j)$; // sum of the in-degrees of existing nodes
5 $V' \leftarrow \emptyset$;
6 Choose randomly m nodes from V and add them to V', where the probability of choosing node j is $(indeg(j)+1)/(totindeg+|V|)$; // The m nodes may not be distinct; hence $|V'| \leq m$
7 $V \leftarrow V \cup \{i\}$; // new node i is added to set V
8 $E \leftarrow E \cup \{(i,j) : j \in V'\}$; // connect the new node to the randomly chosen nodes
9 **return** $g = (V, E)$;

Обратите внимание, что программирование эффективной реализации алгоритма DPA может оказаться сложней, чем кажется на первый взгляд. В частности, если реализовывать строку 6 с помощью обычного цикла, то обработка графа с 28000 вершин на обычном компьютере может занять до 30 минут.

Для того, чтобы избежать подобной проблемы, вы можете изучить и использовать следующий класс DPATrial. Класс состоит из двух методов:

- `__init__(num_nodes)` – создает объект класса, соответствующий полносвязному графу с num_nodes вершинами,
- `run_trial(num_nodes)` – производит серию из num_nodes случайных испытаний (строки 4-6 алгоритма). Возвращает множество вершин, являющихся соседями вновь созданной.

В предоставленном коде класс DPATrials поддерживает список номеров вершин, в котором каждый номер повторяется несколько раз. Если количество повторений каждой вершины будет пропорционально желаемому распределению вероятностей, то вызов random.choice() будет возвращать вершину с нужной вероятностью.

Напишите код функции UPA на основе кода DPA. Нетрудно заметить, что в тот момент, когда DPA добавляет в граф ориентированное ребро, алгоритм UPA должен добавить неориентированное. Заметим также, что в случае неориентированного графа степень новой вершины уже не будет нулевой, что повышает ее шансы быть выбранной. В качестве класса для проведения случайных испытаний вы можете использовать класс UPATrial.

[2] Среди m случайно выбранных вершин могут быть повторяющиеся.

Подзадача №2

Анализ устойчивости компьютерной сети начнем со случая, когда узлы сети выводятся из строя в случайном порядке. После этого мы сравним устойчивость реальной компьютерной сети с устойчивостью ER и UPA-графов примерно такого же размера.

Начните с того, что определите вероятность p, при которой алгоритм ER генерирует граф примерно с таким же количеством ребер, как и в считанной из текстового файла компьютерной сети. После этого найдите значение параметра m, чтобы сеть, сгенерированная алгоритмом UPA, также имела сравнимый размер. Помните, что все три анализируемых графа должны иметь одинаковое количество вершин и примерно равное количество ребер. Запишите полученные значения двух параметров.

Далее напишите код двух функций. Первая - `compute_resilience`. Она должна принимать на вход граф и список вершин. Функция удаляет вершины из графа одну за одной в том порядке, в котором они заданы в списке, и подсчитывает размер наибольшей компоненты связности графа. На выход функция выдает подсчитанные размеры наибольших компонент.

Вторая функция – `random_order`. Она берет на вход граф и возвращает список его вершин в каком-то случайном порядке.

Далее для каждого из трех графов (компьютерная сеть, ER, UPA) сгенерируйте случайную последовательность атакуемых узлов с помощью `random_order` и рассчитайте массивы устойчивости с помощью функции `compute_resilience`.

Изобразите все три списка в виде графика (на одной картинке). По оси абсцисс откладывается количество удаленных вершин графа (от нуля до размера графа), а по оси ординат – размер наибольшей компоненты связности, оставшейся после того, как соответствующее количество вершин было удалено из графа. Подпишите все графики, чтобы было видно, к чему они относятся.

Подзадача №3

Предположим, что в компьютерной сети из строя выводится существенная часть серверов. Мы будем называть такую сеть устойчивой, если размер наибольшей компоненты связности примерно (отклонение не более 25%) равен количеству оставшихся в графе узлов на протяжении всего времени, как сервера выводились из строя.

Проанализируйте данные, полученные в подзадаче №2. Какие из трех графов являются устойчивыми при удалении первых 20% вершин? Приведите расчеты.

Подзадача №4

В оставшихся трех подзадачах мы будем рассматривать случай, когда узлы сети удаляются не в случайном порядке, а с учетом структуры графа. Простое правило подобных таргетированных атак заключается в том, чтобы выводить из строя узел с максимальным количеством связей (вершина с максимальной степенью). Изучите код функции targeted_order, который в цикле делает следующие действия:

- Находит узел с максимальной степенью. Если таковых несколько, то выбирает любой.
- Удаляет данный узел и все инцидентные ребра из графа.

Обратите внимание, что алгоритм удаляет вершины из графа и на каждом шаге определяет максимальную входящую степень среди тех вершин, которые остались. Функция должна возвращать номера вершин в том порядке, в котором они удалялись из графа[3].

После изучения кода вы можете заметить, что он не очень эффективен. В частности, очень много ненужной работы делается всякий раз, когда мы ищем узел с максимальной степенью. Рассмотрим более эффективную версию этого метода (будем называть его `fast_targeted_order`). Вот его псевдокод:

Algorithm 1: FastTargetedOrder.

Input: Graph $g = (V, E)$, with $V = \{0, 1, \ldots, n-1\}$.
Output: A (ordered) list L of the nodes in V in decreasing order of their degrees.

```
for k ← 0 to n − 1 do
    DegreeSets[k] ← {u ∈ V : degree(u) = k};
L ← [];                                   // L is initialized to an empty list
i ← 0;
for k ← n − 1 downto 0 do
    while DegreeSets[k] ≠ ∅ do
        Let u be an arbitrary element in DegreeSets[k];
        DegreeSets[k] ← DegreeSets[k] − {u};
        foreach neighbor v of u do
            d ← degree(v);
            DegreeSets[d] ← DegreeSets[d] − {v};
            DegreeSets[d − 1] ← DegreeSets[d − 1] ∪ {v};
        L[i] ← u;
        i ← i + 1;
        Remove node u from g;
return L;
```

Алгоритм создает список множеств `degree_sets`, в котором k-тый элемент содержит множество узлов со степенью k. Далее метод перебирает все элементы `degree_sets` в порядке убывания степени k. Легко заметить, что, когда он встречает непустое множество, все его элементы имеют максимальную степень. Далее метод поочередно удаляет все вершины из этого множества и параллельно обновляет множества из `degree_sets`.

В данной подзадаче вам необходимо реализовать оба алгоритма, а затем проанализировать их время работы на UPA-графе размера n с m=5. Ваш анализ должен быть как аналитическим, так и эмпирическим и включать в себя следующее:

- Определите асимптотическое наихудшее время работы обоих алгоритмов (`targeted_order` и `fast_targeted_order`) в зависимости от n.
- Нарисуйте графики зависимости времени выполнения этих алгоритмов от n.

Так как количество узлов в UPA-графе всегда меньше 5n (из-за выбора m=5) ваша оценка в виде O-нотации должна быть выражена как функция от n. Вы также можете считать, что все операции с множествами в `fast_targeted_order` выполняются за O(1).

Далее определите сложность алгоритмов эмпирически – для этого выполните оба алгоритма на UPA-графах с размерами range(10, 1000, 10) при m=5 и используйте утилиты

[3] Обратите внимание, что функция хоть и может реально уничтожать граф, но она не должна это делать с его единственной копией, т.к. она понадобится в дальнейшем для анализа устойчивости. Если вы реализуете «деструктивный» алгоритм, то выполняйте его на копии графа.

из библиотеки time, чтобы рассчитать фактическое время выполнения алгоритмов. Затем изобразите полученные значения на графиках.

Подзадача №5

Продолжим анализ устойчивости компьютерной сети. Исследуем ее поведение в том случае, когда узлы атакуются в зависимости от количества инцидентных им ребер.

Используя функцию `targeted_order` (или `fast_tageted_order`, если вы выполнили подзадачу №4), постройте последовательности атакуемых узлов для всех трех графов. Далее вычислите их устойчивость с помощью функции `compute_resilience`. Наконец отобразите все значения на графике (три линии на одном рисунке). Обязательно добавьте легенду.

Подзадача №6

Изучите данные, полученные в подзадаче №5. Какие из трех графов являются устойчивыми при удалении первых 20% вершин?

Сдаваемые результаты.

- Комментированный код с указанием конкретного вклада каждого участника проекта.
- Примеры выполнения кода.
- Ответы на все вопросы и все графики, которые требовалось построить в каждой подзадаче.