Отчет по исследованию алгоритма топологической сортировки

Величко Кирилл Андреевич

7 декабря 2020 г.

Содержание

1 Введение

Топологическая сортировка (Topological sort) — один из основных алгоритмов на графах, который применяется для решения множества более сложных задач. Задача топологической сортировки графа состоит в следующем: указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Очевидно, что если в графе есть циклы, то такого порядка не существует. Представленный далее алгоритм был придуман Тарьяном в 1976 году.

1.1 Постановка задачи

Пусть задан ориентированный граф $G = (V, E)$, где:

- V — множество вершин графа.
- $E \subset V \times V$ — множество ребер графа.
- $G(V, E)$ — не содержит циклов

Требуется найти такое отображение $\phi : V \to \{1..n\}, uv \in E \to \phi(u) < \phi(v)$

1.2 Неформальное описание алгоритма:

Для каждой вершины V графа $G(V, E)$ мы вызываем алгоритм поиска в глубину. После завершения работы над вершиной мы кладём её в начало связного списка всех вершин. Утверждается, что полученный связный список, является тем самым отображением $\phi : V \to \{1..n\}$, удовлетворяющий условию $uv \in E \to \phi(u) < \phi(v)$

1.3 Формальное описание алгоритма

Ниже приведен псевдокод реализующий алгоритм топологической сортировки:

```
TopologicalSort(G) :
1  Fill(visited, False)
2  V := G.V
3  for v ∈ V
4     if not visited[v]
5        DFS(v)
6  ans.reverse()
7  return ans
DFS(v) :
1  visited[v] := True
2  for vu ∈ E
3     if not visited[u]
4        DFS(u)
5  ans.pushback(v)
```

Алгоритм предполагает, что на вход подаётся ориентированный граф G без циклов.

Пояснения к процедурам

Процедура Fill принимает массив **visited** и устанавливает для всех вершин стандартное значение **False**.

Процедура DFS принимает принимает вершину $v \in V$ и выполняет поиск в глубину из этой вершины.

Процедура Pushback добавляет вершину v в конец массива.

Процедура Reverse выполняет перестановку элементов массива в обратном порядке.

https://github.com/veliKerril/topsort – код, реализующий алгоритм на языке C++

1.4 Применение

Топологическая сортировка применяется в самых разных ситуациях, например при создании параллельных алгоритмов, когда по некоторому описанию алгоритма нужно составить граф зависимостей его операций и, отсортировав его топологически, определить, какие из операций являются независимыми и могут выполняться параллельно (одновременно). Примером использования топологической сортировки может служить создание карты сайта, где имеет место древовидная система разделов. Также топологическая сортировка применяется при обработке исходного кода программы в некоторых компиляторах и IDE, где строится граф зависимостей между сущностями, после чего они инициализируются в нужном порядке, либо выдается ошибка о циклической зависимости.

2 Математическое обоснование алгоритма

2.1 Обоснование корректности алгоритма

Лемма 2.1. G - ациклический ориентированный граф, тогда $uv \in E \to$ leave[u] > leave[v], где leave - массив времени выхода из вершины при обходе в глубину.

Доказательство. Введём следующую терминологию. Вершина $v \in V$ при обходе поиском в глубину называется:

- **белой**, если она еще не была рассмотрена алгоритмом
- **серой**, если она находится в текущем дереве вызовов процедуры DFS
- **черной**, если работа с ней уже закончена

Рассмотрим произвольное ребро (u, v), исследуемое процедурой DFS. При исследовании вершина v не может быть серой, так как серые вершины в процессе работы DFS всегда образуют простой путь в графе, и факт попадания в серую вершину v означает, что в графе есть цикл из серых вершин, что противоречит условию утверждения. Следовательно, вершина v должна быть белой либо черной. Если вершина v — белая, то она становится потомком u, так что leave[u] > leave[v]. Если v — черная, значит, работа с ней уже завершена и значение leave[v] уже установлено. Поскольку мы все еще работаем с вершиной u, значение leave[u] еще не определено, так что, когда это будет сделано, будет выполняться неравенство leave[u] > leave[v]. Следовательно, для любого ребра (u, v) ориентированного ациклического графа выполняется условие leave[u] > leave[v]. □

Теорема 2.2. G - ациклический ориентированный граф, тогда $\exists \phi : V \to \{1..n\}, uv \in E \to \phi(u) < \phi(v)$

Доказательство. Определим leave[u] как порядковый номер окраски вершины u в черный цвет в результате работы процедуры DFS. Рассмотрим функцию $\phi = n + 1 -$ leave[u]. Очевидно, что такая функция подходит под критерий функции ϕ из условия теоремы, так как выполнена предыдущая лемма. □

Следствие 2.3. Представленный выше алгоритм топологической сортировки работает корректно.

2.2 Оценка сложности алгоритма

Если структуры **visited**, **G.V**, **G.E**, **ans** устроены как массивы, то сложность процедур следующая:

- Сложность процедуры Fill — $\Theta(V)$.
- Количество итераций цикла **for** — $\Theta(V)$
- Суммарная сложность процедуры DFS — $\Theta(V + E)$ (в сумме по всем итерациям цикла).

- Сложность процедуры pushback — $\Theta(1)$.
- Сложность процедуры reverse — $\Theta(V)$.

Итоговая сложность

$$\Theta(V) + \Theta(V) + \Theta(V+E) + \Theta(V) \cdot \Theta(1) + \Theta(V) = \Theta(V+E)$$

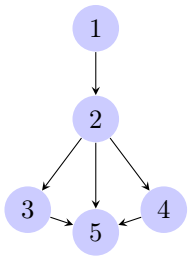
3 Характеристики входных данных

3.1 Структура входных данных алгоритма

Входные данные должны содержать описание графа. Не теряя общности, положим что вершинами графа являются натуральные числа, а сам граф не содержит кратных ребер и петель.

Первая строка содержит два числа n и m — количество вершин и ребер в графе.

Следующие m строк содержат три числа (u, v) — означающие что в графе есть ребро (u, v).




```
5 6
1 2
2 3
2 4
2 5
3 5
4 5
```

Рис. 1: Иллюстрация: Слева дан ациклический ориентированный граф, справа дано его формальное описание

3.2 Генерация входных данных

Числа n и m выбираются из фиксированного диапазона , причем $\max(m) \leqslant \frac{n(n-1)}{2}$. После чего генерируется m чисел из диапазона $[0; \frac{n(n-1)}{2})$. Каждое число соответствует ребру (u, v) по следующему правилу:

$$x \mapsto (x \text{ div } n, x \bmod n)$$

Далее нам необходимо проверить граф на ацикличность, это можно сделать простым обходом в глубину. Для этого нам необходимо проверить, что в дереве обхода нет обратных ребёр.

Формальная процедура генерации выглядит так:

GenGraph(N) :

1 $M \coloneqq \text{GetRandomInt}\left([N^{\frac{1}{2}}, \frac{N(N-1)}{2}]\right)$

2 $E \coloneqq \text{GetRandomSeq}\left([0; \frac{N(N-1)}{2} - 1], M\right)$

3 $Edges \coloneqq \{\varnothing\}$

4 **for** $e \in E$:

5 $\quad E = E \cup \big(\text{e div N}, \text{e mod N}\big)$

6 $V = \{1..n\}$

7 **if** $Acyclic(V, E)$:

8 $\quad Write(N, M)$

9 $\quad$ **for** $u, v \in Edges$:

10 $\quad\quad Write(u, v)$

Ребра ограничены снизу, чтобы граф был не слишком разряжённым и сверху, так как при большем количнстве ребёр граф, очевидно, имеет циклы.

4 Вычислительный эксперимент

4.1 Цели

- Установить эмпирическую зависимость времени исполнения программы-алгоритма при росте параметра N от 5 вершин до 100;
- Полученные результаты сравнить с теоретическими оценками;

4.2 Методология

Для каждого $n \in [5; 100]$ кратного 5 сгенерировать 300 тестовых случаев, и сравнить время работы программы на параметрах n и m с теоретическими следующим образом:

- Рассмотреть медианное значение $\mathcal{T}(n)$
- Рассмотреть медианное отношение $\frac{T(n+m)}{n+m}$
- Рассмотреть медианное отношение $\frac{T(2n)}{n}$

Здесь $\mathcal{T}(n)$ — среднее время работы программы для теста с n вершинами, $\mathcal{T}(n+m)$ — время работы программы для теста с n вершинами и m рёбрами.

5 Результаты

Численные результаты представлены в таблице 1

N	$\mathcal{T}(n)$	$\frac{T(N+M)}{N+M}$	$\frac{T(2N)}{N}$
5	1.0	0.11	2.0
10	2.0	0.14	2.0
15	2.3	0.18	1.826
20	4.0	0.11	1.25
25	4.2	0.11	1.666
30	5.0	0.11	1.6
35	5.0	0.09	1.9
40	6.0	0.11	1.68
45	6.0	0.10	2
50	7.0	0.11	1.85
55	7.2	0.11	
60	8.0	0.10	
65	9.0	0.11	
70	9.5	0.09	
75	10.0	0.10	
80	10.1	0.10	
85	11.0	0.10	
90	12.0	0.11	
95	12.5	0.14	
100	13.0	0.10	
Среднее арифметическое		**0.11**	**1.78**
Медиана		**0.11**	**1.84**

Таблица 1: Результаты эксперимента

Зависимость $\mathcal{T}(n)$ представлена на рисунке 2:

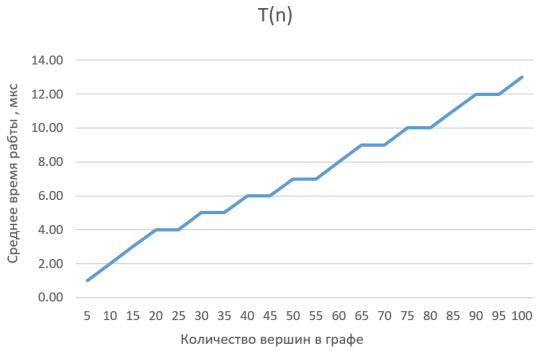


Рис. 2: $\mathcal{T}(n)$

Зависимость $\frac{T(n+m)}{n+m}$ представлена на рисунке 3:

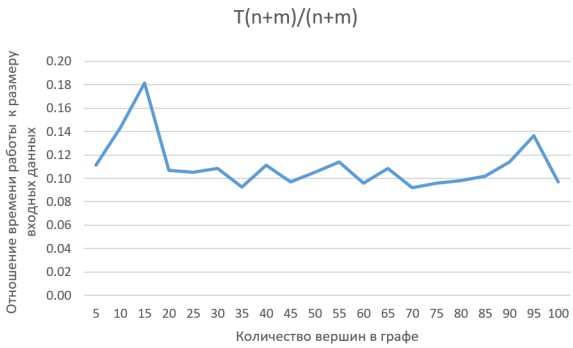


Рис. 3: $\frac{T(n+m)}{n+m}$

5.1 Выводы

Эмпирически полученная зависимость $\mathcal{T}(n+m)$, несмотря на небольшие отклонения, в среднем отличается от теоретически полученной оценки на константу 0.11. А медиана отношения $\frac{T(2N)}{T(N)}$ равна 1.84, что недалеко от теоретического значения 2. Возможно, это вызвано малыми значениями n. Отсюда можно сделать вывод, что данный алгоритм на самом деле имеет сложность работы $\Theta(V+E)$.

6 Список литературы

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. Алгоритмы: Построение и анализ [2005]
- Бьерн Страуструп. Язык программирования C++. - 3е Издание [1985]