Отчёт

Описание тестируемых алгоритмов

DFS

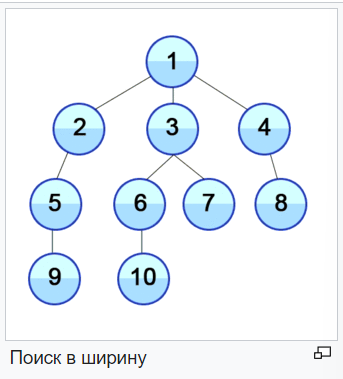
Один из методов обхода графа. Depth-first search – стратегия поиска в глубину: алгоритм устроен так, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

Изображение выглядит как текст, часы

Автоматически созданное описание

BFS

Один из методов обхода графа. Breadth-first search – стратегия поиска в ширину, в процессе обхода мы идём вширь, то есть перед тем как приступить к поиску вершин на расстоянии k + 1, выполняется обход вершин на расстоянии k.



Topological Sort by Kahn

Топологическая сортировка - способ нумерации вершин ориентированного ацикличного графа, при котором каждое ребро ведёт из вершины с меньшим номером в вершину с большим номером. Для одного графа может существовать несколько способов топологической сортировки (например, когда он несвязный), а может не существовать не одного (при наличии циклов).

Алгоритм Кана - один из первых алгоритмов, и наиболее приспособленный к исполнению вручную.

Изображение выглядит как текст

Автоматически созданное описание

Top Sort on DFS

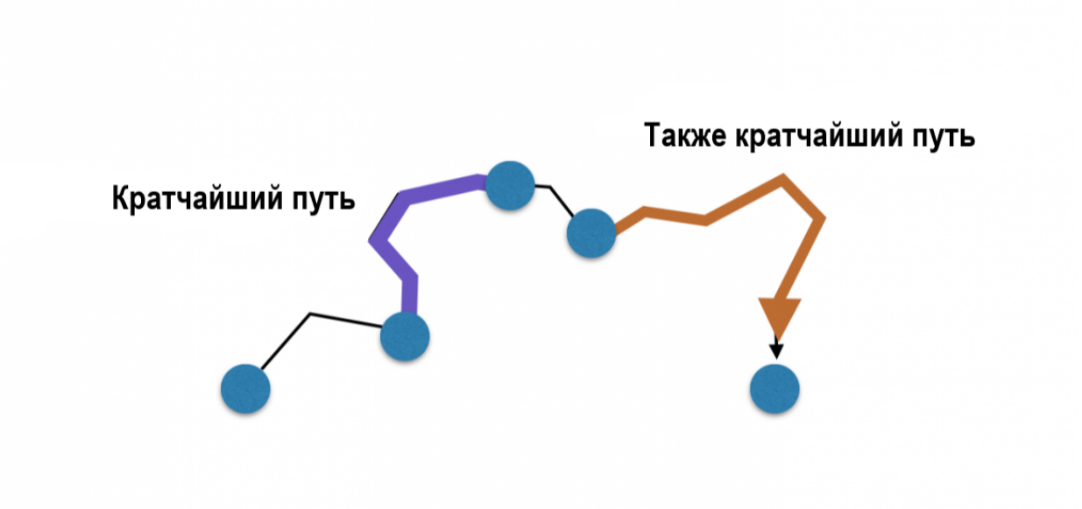
Заметим, что вершину, из которой не ведет ни одно ребро, можно всегда поставить последней. И такая вершина всегда есть в графе без циклов. Из этого сразу следует доказательство: просто будем класть в массив вершину, из которой ничего не ведет и убирать ее из графа. Массив потом надо будет развернуть.

Вершина, из которой мы вышли в DFS-е первой - как раз та, из которой ничего не выходит. Ведь если из нее есть ребро, то только в уже посещенную, но мы не могли выйти из нее, ведь мы берем самую первую из которой мы вышли. Значит, это одна из вершин, в которые мы вошли, и не вышли, и это цикл.

Из этого следует, что достаточно просто брать вершины в порядке выхода из DFS, то есть в конце DFS, например, просто класть эту вершину в конец массива с ответом.

Алгоритм Дейкстры

Алгоритм Дейкстры работает на том основании, что любой подпуть B -> D кратчайшего пути A -> D между вершинами A и D также является кратчайшим путем между вершинами B и D.



Подробное описание реализации алгоритмов – см. комментарии к коду

Анализ результатов тестирования

Во всех таблицах в отчёте представлены средние значения измерений на указанных данных. Точность измерений указана под таблицами.

Тестирование времени

После многократных тестовых запусков было выявлено, что лучшие данные для этих алгоритмов - множество графов, состоящих из одной вершины.

Время работы получилось таким:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function | DFS time, sec | BFS time, sec | Topological sort by Kahn time, sec | Topological sort on DFS time, sec |
| Number of dots |
| 100 | 0.000001 | 0.000001 | 0.000026 | 0.000035 |
| 300 | 0.000001 | 0.000001 | 0.000073 | 0.000097 |
| 500 | 0.000001 | 0.000002 | 0.000117 | 0.000172 |
| 800 | 0.000001 | 0.000002 | 0.000195 | 0.000266 |
| 1200 | 0.000001 | 0.000004 | 0.000283 | 0.000386 |

Отклонение от средней величины: <= 0.0 sec

Предполагаем, что сейчас работа DFS быстрее BFS, так как в нашем варианте реализации BFS создаётся список, размер которого равен количеству вершин, а в DFS используется словарь.

Алгоритм Кана в сначала проходит по всем точкам и находит корни, в данном случае все точки - корни дерева. То есть последовательно проходит по всем точкам, потом выводит их.

Топологическая сортировка, которая использует DFS, из случайной вершины начинает спускаться вниз, и так как графы состоят из одной вершины, работа алгоритма завершается, из-за этих вызовов топологическая сортировка на DFS работает медленнее в данном случае.

Далее худшими данными будет граф, в котором каждая вершина соединена с каждой другой.

Время получилось таким

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function | DFS time, sec | BFS time, sec | Topological sort by Kahn time, sec | Topological sort on DFS time, sec |
| Number of dots |
| 100 | 0.000363 | 0.000565 | 0.001123 | 0.000343 |
| 300 | 0.003117 | 0.004995 | 0.009109 | 0.003074 |
| 500 | 0.009410 | 0.013582 | 0.026856 | 0.009316 |
| 800 | 0.025796 | 0.033652 | 0.070872 | 0.024975 |
| 1200 | 0.058072 | 0.074392 | 0.162351 | 0.057561 |

Отклонение от средней величины: <= 8.291 sec. Так как среди плохих случаев есть такие, которые работают значительно медленнее, чем остальные (из-за краевых случаев отклонение велико)

Так как в алгоритме Кана мы проходим по каждой вершине, алгоритм начинает работать в разы медленнее, чем остальные алгоритмы.

Интересная и необъяснимая деталь: топологическая сортировка на DFS работает быстрее, чем сам DFS, хотя должно быть наоборот, так как в топ сорте есть проверка всех вершин на то, были они посещены или нет.

Алгоритм BFS так же работает медленнее чем DFS. Это происходит по той же причине.

Возможно, влияет работа deque.

Так же мы рассмотрели случайно сгенерированный граф

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function | DFS time, sec | BFS time, sec | Topological sort by Kahn time, sec | Topological sort on DFS time, sec |
| Number of dots |
| 100 | 0.000153 | 0.000214 | 0.000397 | 0.000140 |
| 300 | 0.001209 | 0.001866 | 0.003253 | 0.001224 |
| 500 | 0.003746 | 0.005125 | 0.009280 | 0.003739 |
| 800 | 0.012600 | 0.015861 | 0.027084 | 0.012670 |
| 1200 | 0.032061 | 0.036615 | 0.062623 | 0.033357 |

Отклонение от средней величины: <= 3.123 sec. Так как данные сгенерированы случайно, и может случиться так, что будет сгенерирован плохой случай, алгоритм будет работать дольше, чем в обычной ситуации.

Тут у нас уже топологическая сортировка на DFS и сам DFS чередуются по скорости.

По тем же самым причинам BFS и алгоритм Кана работают медленнее.

Тестирование памяти

После многократных тестовых запусков было выявлено, что лучшие данные для наших алгоритмов – также множество графов состоящих из одной вершины

Вот сколько памяти было использовано:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function | DFS, bytes | BFS, bytes | Topological sort by Kahn, bytes | Topological sort on DFS, bytes |
| Number of dots |
| 100 | 24 | 24 | 2796 | 2796 |
| 300 | 24 | 24 | 8396 | 8396 |
| 500 | 24 | 24 | 13996 | 13996 |
| 800 | 24 | 24 | 22396 | 22396 |
| 1200 | 24 | 24 | 33596 | 33596 |

Отклонение от средней величины: <= 10 bytes

Так как у нас DFS и BFS выводят массив из одной вершины, то их вес состоит из одной вершины (их вес в выводе равен 24)

Алгоритмы топологической сортировки проходят по всем вершинам и выводят одинаковые ответы.

Далее, на тестовых запусках было выявлено, что худшие данные - если каждая вершина соединена с каждой.

Тут затраты памяти получились такими

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function | DFS, bytes | BFS, bytes | Topological sort by Kahn, bytes | Topological sort on DFS, bytes |
| Number of dots |
| 100 | 2796 | 2796 | 0 | 2796 |
| 300 | 8396 | 8396 | 0 | 8396 |
| 500 | 13996 | 13996 | 0 | 13996 |
| 800 | 22396 | 22396 | 0 | 22396 |
| 1200 | 33596 | 33596 | 0 | 33596 |

Отклонение от средней величины: <= 10 bytes

Топологическая сортировка алгоритмом Кана выводит пустой массив, так как он находит циклы в графе.

Остальные алгоритмы выводят все вершины.

Так же мы рассмотрели случайно сгенерированный граф:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function | DFS, bytes | BFS, bytes | Topological sort by Kahn, bytes | Topological sort on DFS, bytes |
| Number of dots |
| 100 | 2796 | 2796 | 0 | 2796 |
| 300 | 8396 | 8396 | 0 | 8396 |
| 500 | 13996 | 13996 | 0 | 13996 |
| 800 | 22396 | 22396 | 0 | 22396 |
| 1200 | 33596 | 33596 | 0 | 33596 |

Отклонение от средней величины: <= 10 bytes

Тут все точно также: у нас создается случайный граф, и в нем есть цикл.

Остальные точно так же выводят все вершины.

Измерения времени проводились с помощью функции time.time()

Кол-во запусков в тестировании времени: 1000. Также изначально выполняется три холостых прогона(время выполнения которых не учитывается)

Измерения памяти проводились с помощью функции sys.getsizeof(test\_func)

Кол-во запусков в тестировании памяти: 1, так как объём памяти зависит только от размера входных данных – на больших графах большой объём потребляемой памяти. Поэтому достаточно вычислить объём памяти один раз для различных графов.

Отклонения(точность вычисленных значений времени и памяти) были посчитаны в коде:

