Петрозаводский государственный университет
Институт математики и информационных технологий
Кафедра Прикладной Математики и Кибернетики

01.04.02 - Прикладная математика и информатика

Отчет о практике по научно-исследовательской работе

Отчет по научно-исследовательской практике

Нахождение k-го кратчайшего пути в графе

(промежуточный)

Выполнил:
студент второго курса группы 22204

И. А. Кошкарев _____
подпись

Место прохождения практики:
Кафедра Прикладной Математики и Кибернетики

Период прохождения практики:

Руководитель:
Р. В. Воронов, д.т.н., доцент

_____
подпись

Итоговая оценка

_____
оценка

Петрозаводск — 2023

Содержание

Введение

Актуальность

Вообще говоря, в жизни часто возникает потребность в нахождении самого короткого (или самого дешевого) маршрута где бы то ни было. Примеров привести можно множество: таксисту необходимо знать от навигатора кратчайший путь от одного адреса до другого; логисту нужно построить самый дешевый маршрут для поставки товаров и т.д.

Однако иногда один единственный кратчайший путь может не подходить по каким-либо причинам, которые зачастую тяжело формализовывать: таксист может не захотеть ехать по выбранному пути, потому что если поехать по нему, то пассажир может упустить из виду какую-нибудь очень красивую достопримечательность; логист может не захотеть брать найденный путь, потому что знает, что на каком-то из участков выбранного пути находятся особо злые таможенники.

Таким образом, иногда появляется нужда в предоставлении пользователю не одного пути, а сразу нескольких. Но при этом таких, чтобы они были не сильно короче/дешевле, чем самый короткий/дешевый путь.

Цели

Для того чтобы поставить цели и задачи практики, сначала нужно свести желания пользователей к математической модели.

Многие системы можно представить в виде взвешенного графа. В случае с таксистом в качестве вершин графа можно взять перекрестки города, а в качестве ребер — дороги. Вес ребра можно выбрать огромным количеством способов, поскольку время, затраченное на преодоление определенного участка города, может зависеть от многих факторов: длина дороги, загруженность дороги, математическое ожидание времени стоянки на светофоре перекрестка и прочее. Однако грамотный выбор веса ребра оставим в этой работе за кадром. Главное понимать, что как-то его выбрать точно можно.

Тогда самый короткий путь от одного адреса до другого будет соответствовать самому короткому пути в построенном графе. А какой-то близкий к кратчайшему, но отличающийся от него, путь в городе, будет соответствовать какому-то пути в графе, который по своей длине должен быть в каком-то смысле близок к кратчайшему.

Таким образом, можно поставить следующие цели: изучение, сравнение и реализация эффективных алгоритмов для нахождения k-х кратчайших путей в графе.

Задачи практики

В связи с поставленной целью, задачи практики будет следующими:

1. Изучить существующе методы нахождения k-х путей;
2. Реализовать алгоритмы на одном из языков программирования;
3. Сгенирировать набор тестов для сравнения эффективности алгоритмов;
4. Сравнить методы между собой и дать рекомендации, в каких случаях лучше использовать тот или иной метод.

Постановка математической модели задачи

Решать задачу мы будем в следующем варианте.

Задан взвешенный ориентированный граф $G(V, E)$ и стартовая вершина $s \in V$. Для фиксированной вершины $t \in V$ требуется найти длину k-го кратчайшего пути из вершины s в вершину t.

Определим k-й кратчайший путь в графе из вершины s в вершину t. Рассмотрим множество всех путей в графе из вершины s в вершину v. Упорядочим все эти пути по сумме весов их ребер. k-й порядковый путь и назовем k-м кратчайшим путем.

Заметим, что в данном определении k-го кратчайшего не было отмечено, является ли путь простым. Объясняется это тем, что некоторые представленные алгоритмы будут искать в графе непростые пути, а некоторые – простые.

Сама же задача заключается в нахождении либо первых k кратчайших путей, либо в нахождении одного k-кратчайшего пути (или его длине).

Кроме того, будем считать, что количество вершин $n \leq$ количество ребер m для упрощения оценки времени работы алгоритмов.

О реализациях

Реализация будет приведена в нулевой индексации вершин графа. Кроме того, в качестве стартовой вершины s будем считать вершину с номером 0, а в качестве конечной t — вершину $n - 1$.

Каждый алгоритм будет реализован на языке `C++` в виде функции, которая в качестве параметров принимает `const std::vector<std::pair<int, unsigned long long> > &g`,

являющийся списком смежности графа и `size_t k` — неотрицательное число k, используемое в условии задачи.

Выбор языка объясняется большой скоростью работы языка `C++`, а также своей высокоуровневостью (относительно, например, языка `C`). То есть, с одной стороны, он имеет широкий функционал, а с другой — реализации алгоритмов на языке `C++` будут работать сравнительно быстрее, нежели на многих других популярных языках.

Реализацию алгоритмов можно найти по ссылке:

`https://github.com/4qqqq/kth-path-algorithms`.

Алгоритм Йены

Алгоритм Йены находит список k кратчайших **простых** путей в графе; он был опубликован Цзинь И Йеной в 1971 году.

Для работы алгоритма необходимо наличие алгоритма, который умеет находить обычный кратчайший путь в графе. Чаще всего для этого используют алгоритм Дейкстры.

Ограничения на структуру графа

Для корректной работы алгоритма Йены необходимо, чтобы G не содержал кратных ребер и дуг. Кроме того, в описываемой ниже версиии алгоритм Йены использует внутри своей работы алгоритм Дейкстры, который накладывает ограничение на веса ребер графа: они должны быть неотрицательными.

Алгоритм

На каждом шаге алгоритма у нас будут поддерживаться два списка: список кандидатов и результирующий список (обозначим их как $cand$ и res соответственно). При каждом добавлении очередного пути в res будем обновлять $cand$. Затем будем брать среди кандидатов самый короткий путь и удалять его из $cand$.

В начальный момент работы алгоритма нужно положить в res кратчайший путь от s до t, который можно найти алгоритмом Дейкстры.

Как обновлять $cand$? Заметим, что если в res был добавлен очередной путь p, то каждый новый кандидат c должен иметь какое-то общее начало (префикс) с p (возможно, общее начало будет состоять только из стартовой вершины), после которого p будет отличаться от c по одному ребру.

Получается, новый кандидат должен искаться как кратчайший путь на изначальном графе, из которого удалили ребро, следующее за перебираемым префиксом текущего пути p. Кроме того, так как в данном алгоритме мы ищем только простые пути, нам нужно удалить еще и все вершины текущего префикса пути p, за исключением последней вершины, чтобы случайно не прийти в них еще один раз. Также нужно помнить про то, что во время нахождения пути p мы уже удалили какие-то ребра. Значит, их тоже нужно удалить из текущего графа. На получившемся графе нужно запустить алгоритм поиска кратчайшего пути из последней вершины префикса. Найденный путь и будет новым кандидатом.

Таким образом, вкратце алгоритм будет выглядеть так:

1. Изначально списки res и $cand$ пустые;

2. Находим кратчайший путь из вершины s в вершину t алгоритмом Дейкстры; добавим этот путь в res;

3. На очередной итерации рассматриваем последний добавленный путь p в res. Перебираем префикс пути p, удаляем из изначального графа ребро, следующее за концом префикса (а также ребра, которые были удалены при поиске p) и все его вершины, за исключением последней. Из последней вершины префикса запустим алгоритм Дейкстры. Найденный путь добавим в $cand$, запомнив, какое ребро было удалено;

4. Отсортируем список $cand$ по длине пути. Самый первый кандидат попадет в результирующий список, после чего удалится из списка кандидатов;

5. Алгоритм завершается, когда размер res равен k или если $cand$ оказывается пустым (это будет означать, что всего различных простых путей в графе не больше, чем k).

Отметим, что алгоритм Йены никак не использует тот факт, что все ребра графа имеют неотрицательный вес, однако его использует алгоритм Дейкстры. Тем не менее, алгоритм Йены не подразумевает использование именно алгоритма Дейкстры для поиска кратчайшего пути. Это значит, что используя для его нахождения, например, алгоритм Форда-Беллмана, можно ослабить ограничение на исходный граф: в нем не должно быть циклов отрицательного веса, однако сами ребра могут иметь отрицательный вес.

Псевдокод

```
candidates = []
result = [path found by dijkstra(s, t, graph)]

for k = [1...number_of_paths - 1]:
    p = last in result
    prefix = []

    for each edge (u, v, w) in p:
        graph_in_the_moment = graph without (u, v, w)
        and
        without edges which were deleted when p was found
```

```
        new_candidate = prefix + dijkstra(u, t, graph_in_the_moment)

        if new_candidate was not found before:
            push back new_candidate in candidates

    if candidates empty:
        break

    sort candidates by length

    push back candidates[0] in result
    erase candidates[0] from candidates

return result
```

Оценка

Несколько обозначений:

- T — время нахождения кратчайшего пути в графе (в представленной реализации был реализован алгоритм Дейкстры на двоичной куче, работающий за $O(m \log n)$);
- $|P|$ — количество ребер в пути P;
- ans — результирующий список из k кратчайших путей в графе;
- $l = \max_{P \in ans} |P|$;

Нетрудно видеть, что после каждого рассматриваемого пути в список кандидатов добавится $\mathcal{O}(l)$ путей, для каждого из которого запускается алгоритм поиска кратчайшего пути. Всего результирующих путей k, поэтому кандидатов будет $\mathcal{O}(kl)$. Таким образом, при аккуратной реализации алгоритм Йены работает за $\mathcal{O}(klT)$.

Что касается оценки l, то в худшем случае она может быть порядка n (поскольку рассматриваемые пути всегда простые), поэтому верхняя оценка на время работы алгоритма есть $\mathcal{O}(knT)$. Однако в реальности зачастую мы сталкиваемся со случайными графами, где оценка на l может очень сильно варьироваться, засчет чего на практике алгоритм Йены может работать сильно быстрее ожидаемого.

Модифицированный алгоритм Дейкстры

Описываемый ниже алгоритм находит k кратчайших **не обязательно простых** путей.

В целом, Алгоритм Дейкстры используется для нахождения кратчайшего пути в графе, однако его можно модифицировать таким образом, чтобы он находил k-й **непростой** кратчайший путь.

Ограничения на структуру графа

Для корректной работы алгоритма необходимо, чтобы G не содержал ребер отрицательного веса.

Алгоритм

На каждой итерации классического алгоритма во всех вершинах графа хранится длина кратчайшего пути до этой вершины, который мы смогли найти на данный момент. Давайте теперь будем хранить длину не только кратчайшего пути, а длины k кратчайших путей при помощи сбалансированного дерева поиска (в C++ можно использовать `std::set<>`, внутри которого реализовано красно-черное дерево). Тогда в результате работы алгоритма в каждой вершине будут храниться длины k кратчайших путей.

Псевдокод

```
q = empty min multiset
distances = array[n] of multisets
insert (0, s) in q
insert 0 in distances[s]

while q not empty:
   (l, u) = minimal in q
    erase (l, u) from q

    for each outgoing edge (u, v, w) from u:
        insert (l + w, v) in q and distances[v]
        insert l + w in distances[v]
```

```
        if size of distances[v] > k:
            erase maximal in distances[v] from q and distances[v]

return max distances[t]
```

Оценка

Самая труднозатратная часть алгоритма находится во внешнем цикле, откуда мы берем текущее обрабатываемое состояние. Таких состояний может в каждый момент времени быть не более nk, поскольку всего вершин в графе n, а в каждой вершине хранится не более k состояний. С другой стороны, так как мы можем обновлять эти состояния благодаря прохождению по одному из m ребер, то суммарно состояний может оказаться порядка mk.

Таким образом, используя для поддержки состояний сбалансированное дерево поиска или бинарную кучу, имеем время работы алгоритма $\mathcal{O}(mk \log(nk))$.

Стоит отметить два момента.

1. Если в качестве поддержки состояний использовать фибоначчиеву кучу, то удастся добиться асимптотики $\mathcal{O}(mk + nk \log(nk))$. В таком случае алгоритм теоретически будет работать быстрее, чем при использовании обычной двоичной кучи или сбалансированного дерева поиска, однако на практике фибоначчиева куча имеет очень большую скрытую константу работы, из-за чего алгоритм работает не сильно быстрее.

2. Если в качестве поддержки состояний использовать обычный массив, то удастся добиться асимптотики $\mathcal{O}(n^2k+mk)$. В таком случае алгоритм будет работать быстрее на плотных графах, где $m \approx n^2$.

Несмотря на достоинства в некоторых случаях этих двух реализаций, в работе будет приведена реализация на красно-черном дереве (`std::multiset<>`), поскольку она сильно проще реализации на фибоначчиевой куче, а также работает эффективнее, чем реализация на массиве, в случае разряженных графов.

Проблема списка из k непростых путей

В случае, если использовать описанный выше алгоритм для восстановления k кратчайших путей, стоит помнить о следующей проблеме.

Длина k-го кратчайшего **непростого** пути в графе может быть очень большой по количеству ребер. А именно, нетрудно построить пример, когда k-й путь может содержать порядка nk ребер. Это значит, что суммарная длина всех путей по количеству ребер будет порядка nk^2. Поэтому, посмотрев на асимптотическую оценку времени работы алгоритма, можно понять, что этот факт может стать проблемой: оценка сверху на длину по количеству ребер оказывается больше (и даже сильно больше!), чем оценка на время работы алгоритма.

В связи с этим реализация алгоритма возвращает лишь длину k-го кратчайшего пути.

Алгоритм Эппштейна (упрощенный)

Алгоритм Эппштейна находит k-й кратчайший **не обязательно простой** путь.

Этот алгоритм был опубликован Дэвидом Эппштейном 31 марта 1997 года. Вообще говоря, оригинальный алгоритм работает за время $\mathcal{O}(m+n\log n+k)$. Кроме того, алгоритм умеет находить k-й кратчайший путь от заданной вершины до всех остальных за время $\mathcal{O}(m + n\log n + nk)$. Однако, ввиду сложности оригинального алгоритма, в этой работе будет описываться его упрощенная версия, которая, тем не менее, работает значительно быстрее алгоритмов, описанных ранее.

Алгоритм

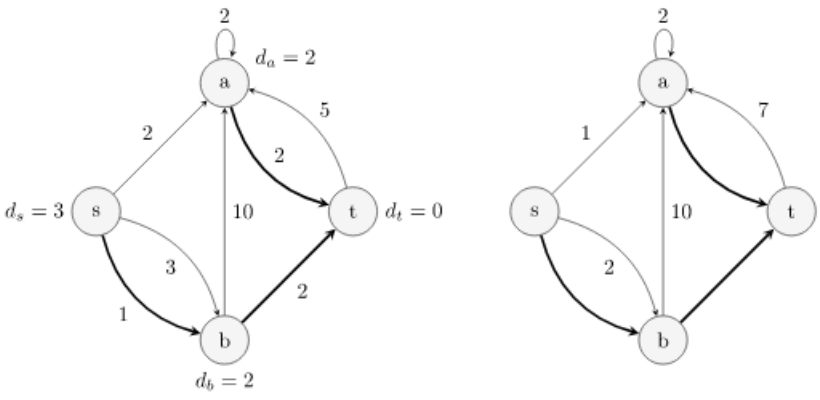
Связь с модифицированным алгоритмом Дейкстры

Отметим, что в статье был описан модифицированный алгоритм Дейкстры, который ищет k-й кратчайший путь за время $\mathcal{O}(mk\log(nk))$. По своей сути, все, что описывается дальше – это дальнейшие модификации этого же алгоритма.

Построение вспомогательного графа

Для простоты из изначального графа G удалим вершины, из которых не достижима t. Теперь алгоритмом Дейкстры (на обратном графе к графу G) найдем расстояние от каждой вершины u до вершины t и обозначим его за d_u. Пусть nxt_u — ребро, по которому идет кратчайший путь от вершины u до вершины t; p_u — вершина, в которую можно попасть из вершины u по ребру nxt_u. Если таких ребер может быть несколько, то возьмем в качестве nxt_u любое из них.

Для каждого ребра $e = (u, v, w)$ введем величину sidetrack(e) $= d_v - d_u + w$.



Пример изначального графа G и значение его *sidetrack*-ребер

Легко видеть, что эта величина всегда неотрицательна. Эта величина показывает, на сколько ребро $(u,\ v,\ w)$ отличается от оптимального ребра nxt_u. Теперь построим граф G', где будут отсутствовать ребра nxt_u, а вес всех остальных ребер e_i заменится на $sidetrack(e_i)$. Кроме того, в графе G' из каждой вершины u будут исходить все ребра, которые исходят из вершин $p_u,\ p_{p_u},\ \ldots,\ t$ с соответствущими концами и $sidetrack$-весами.

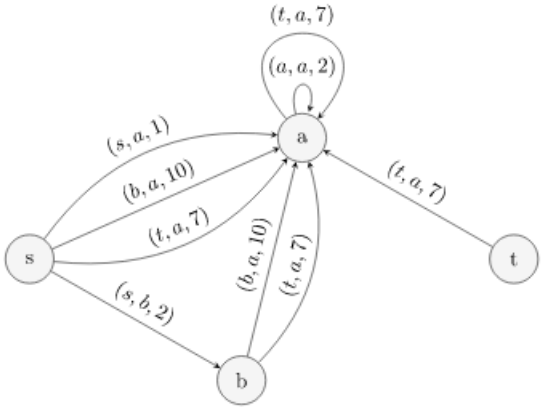
Так же, назовем $sidetrack$-ребром такое ребро, которое не является nxt_u для любой вершины u.

Заметим, что между графами G и G' есть биекция в том смысле, что какой бы путь, начинающийся в вершине u и заканчивающийся в вершине t мы ни взяли в G, можно однозначно сопоставить некоторую последовательность ребер из графа G'. И наоборот — любому пути из графа G' (к которому добавили все nxt_v ребра с нулевым весом) можно сопоставить путь в графе G; более того, если выбранный путь в графе G' имеет длину l, то соответствующий ему путь в графе G имеет длину $l + d_u$.

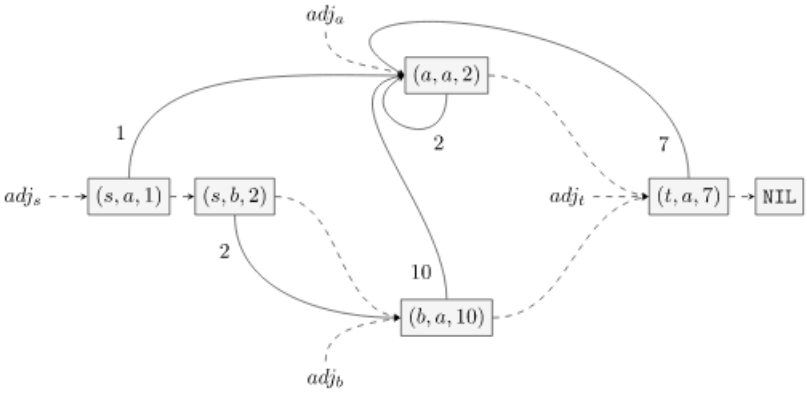
Итак, вкратце опишем построение графа G':

1. Найдем значения d_u, p_u, nxt_u и $sidetrack(e)$ алгоритмом Дейкстры. $sidetrack$-ребра добавим в граф G'.

2. Обозначим список ребер, исходящих из вершины u как adj_u. Рассмотрим ребра $e_i = (p_u,\ v,\ w) \in adj_{p_u}$. Добавим в G' ребра $(u,\ v,\ w)$. Сразу отметим, что нет смысла их добавлять явно. Вместо этого можно поддерживать персистентный список. Тогда мы сможем копировать список adj_{p_u} за $\mathcal{O}(1)$.



Полученный G' из того же примера



Тот же G', но с визуализацией персистентных списков

k-pop алгоритм

Ввиду описанной выше биекции между G и G', если мы найдем k-й кратчайший путь в графе G', то он будет соответствовать k-му кратчайшему пути в графе G.

Находить его длину будем с помощью k-pop алгоритма:

```
q = empty min heap
push (d[s], s) on q

repeat k - 1 times:
    (l, u) = pop q
    // found a path of length l
    for each outgoing edge (u, v, w) from u:
        push (l + w, v) on q

return first(pop q)
```

После выполнения k-pop алгоритма значение, лежащее первым в q, окажется длиной k-го кратчайшего пути до вершины t в графе G'.

Сразу дадим оценку этому алгоритму. Всего мы делаем $\mathcal{O}(k)$ итераций, в каждой из которой совершается столько операций, сколько есть исходящих ребер из текущей вершины. Исходящих ребер может быть вплоть до $\mathcal{O}(m)$. Поэтому куча q может иметь размер до mk.

Таким образом, итоговая оценка времени работы алгоритма есть $\mathcal{O}(mk \log(mk))$. Именно такую оценку мы уже получали в модифицированном алгоритме Дейкстры! Тогда в чем был смысл всего этого?

Основной алгоритм

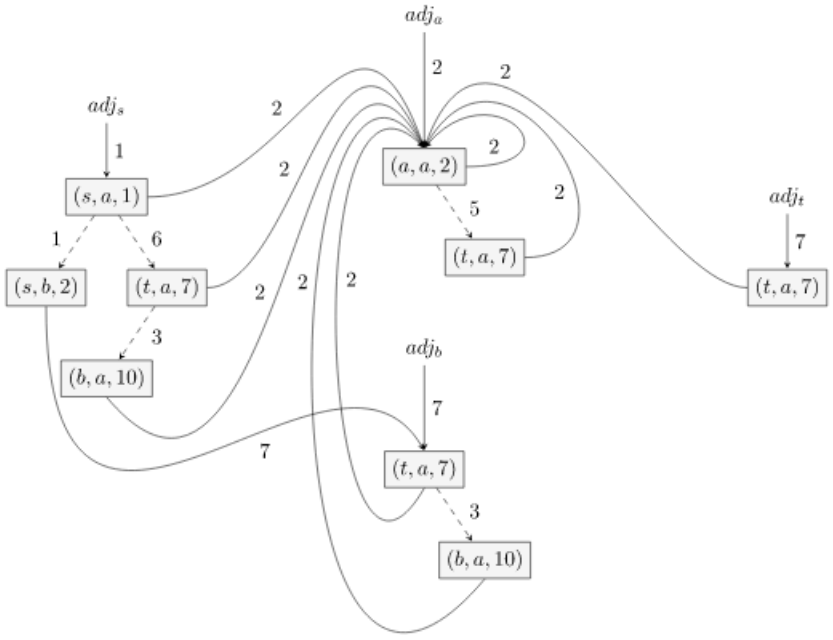
Проблема предыдущего алгоритма заключается в том, что из вершин, которые мы просматриваем, может исходить очень много ребер (вплоть до m). Если бы это значение было ограничено каким-то числом c, то $k - pop$ алгоритм работал бы за $O(ck \log(ck))$. Можем ли мы как-то создать это ограничение?

Давайте сделаем граф G'' из графа G' следующим образом.

adj_u в графе G' был персистентным списком. А давайте попробуем представить adj_u как **двоичную** персистентную кучу на минимум! Тогда adj_u станет кучей на минимум, в которой хранятся *sidetrack*-значения ребер графа G'. Кроме того, давайте в этой куче предадим вес ребрам. Вес ребра между двумя вершинами будет равно разнице значений, находящимся в них.

Но самое главное то, что мы теперь adj_u будем рассматривать не просто как информацию о вершине u, а как замена этой вершины. Другими словами, все вершины и ребра кучи adj_u будут вставлены в граф вместо вершины u.

Теперь из каждой вершины adj_u проведем соответствующие ребра в другие adj_v с весом, равным соответствующей значению вершины в adj_v.



Полученный G'' из G'

На получившемся графе G'' запустим $k - pop$ алгоритм. Заметим, что с его помощью все еще можно легко находить длину k-го кратчайшего пути в изначальном графе G.

Оценка

Итак, теперь структура графа G'' устроена так, что из каждой вершины исходит не более трех ребер (не более двух ребер внутри кучи adj_u и ровно одно ребро, входящее в вершину другой кучи). Таким образом, k-рор алгоритм работает за $\mathcal{O}(3k \log(3k)) = \mathcal{O}(k \log k)$!

Итак, оценим время работы всего алгоритма:

- Нахождение значений d_u и $sidetrack$ работает за $\mathcal{O}(m \log m)$ или $\mathcal{O}(m + n \log n)$ в зависимости от реализации алгоритма Дейкстры;
- Построение графа G'' работает за $\mathcal{O}(m \log m)$, поскольку у нас всего m ребер и каждое из них нужно добавить в персистентную кучу за время $\mathcal{O}(\log m)$;
- k-рор алгоритм занимает $k \log k$ времени.

Таким образом, весь алгоритм работает за $\mathcal{O}(m \log m + k \log k)$.

На самом деле, существуют и дальнейшие оптимизации этого алгоритма, представленные в оригинальной статье Дэвида Эппштейна, которые оптимизируют алгоритм до $\mathcal{O}(n + m + k)$ без учета алгоритма Дейкстры. С другой стороны, упрощенная версия алгоритма проще в реализации и понимании, но при этом не сильно проигрывает в производительности.

Сравнение алгоритмов

Прежде чем сравнивать между собой алгоритмы, стоит еще раз указать на то, что Алгоритм Йены решает не ту же задачу, что и алгоритмы Дейкстры и Эппштейна: первый находит k-й **простой** путь, в то время как вторые два находят k-й **необязательно простой** путь (в нашей работе – его длину). Поэтому напрямую сравнивать эти алгоритмы сравнивать не совсем корректно. Тем не менее, для наглядности, данные о сравнении алгоритмов будут помещены в одну табличку:

Алгоритм	**Время**	**Память**
Йены	$\mathcal{O}(klm \log m)$ или $\mathcal{O}(kl(m + n \log m))$	$\mathcal{O}(m + kl^2)$
Дейкстры	$\mathcal{O}(mk \log(nk))$ или $\mathcal{O}(mk + nk \log(nk))$	$\mathcal{O}(m + nk)$
Эппштейна (У.)	$\mathcal{O}(m \log m + k \log k)$	$\mathcal{O}(m)$

Так же, алгоритмы было замерено время работы и на наборе **случайных** тестов на удаленном сервере со следующими харакетиристиками:

- Ubuntu 22.04.2 LTS;
- Процессор: Intel Xeon Processor (Icelake) [4 ядра по 8 потоков, 2GHZ];
- ОЗУ: 16 ГБ.

Каждый исполняемый файл был скомпилирован из файла с раширением `.cpp` следующими ключами компиляции: `-static -fno-optimize-sibling-calls -fno-strict-aliasing -DONLINE_JUDGE -lm -s -x c++ -O3 -o`.

Набор тестов состоял из пяти групп:

1. *xs* ($n = 100$, $k = 100$, $m \leq 300$);
2. *s* ($n = 1000$, $k = 1000$);
3. *m* ($n = 5000$, $k = 5000$, $m \leq 3 \cdot 10^4$);
4. *l* ($n = 10^5$, $k = 10^5$, $m \leq 3 \cdot 10^5$);
5. *xl* ($n = 10^6$, $k = 10^6$, $m \leq 3 \cdot 10^6$).

Кроме того, каждая группа распредялалась на два типа: тесты, в которых допустимы кратные ребра и петли, и тесты, в которых они не допустимы. Это разделение сделано для того, чтобы можно было корректно тестировать алгоритм Йены.

Алгоритм	**Средн. время (в мс)**					**Макс. время (в мс)**				
	xs	*s*	*m*	*l*	*xl*	*xs*	*s*	*m*	*l*	*xl*
Йены	10.6	2601	—	—	—	13	6412	—	—	—
Дейкстры	5.8	5036	—	—	—	6	11778	—	—	—
Эппштейна (У.)	< 1	9.1	45.2	668	11708	< 1	41	104	804	15837

Заключение

В ходе работы были выполнены все поставленные задачи; тем самым, была достигнута цель — реализация и сравнение существующих алгоритмов для нахождения k-х кратчайших путей.

Осталось лишь дать рекомендации, когда нужно использовать тот или иной алгоритм. Как легко видеть из данных таблиц, алгоритм Эппштейна (даже в упрощенной версии) всегда выигрывает модифицированный алгоритм Дейкстры, притом что оба эти алгоритма решают одну и ту же задачу. Таким образом, заместо алгоритма Дейкстры можно всегда использовать алгоритм Эппштейна.

С алгоритмом Йены получается ситуация немного интереснее. Данный алгоритм решает несколько другую задачу — он ищет только простые пути, в отличие от двух других описанных алгоритмов, ищущих необязательно простые пути. Поэтому напрямую говорить о сравнении эффективности алгоритма Йены с алгоритмом Дейкстры и Эппштейна не совсем корректно.

Таким образом, если необходимо искать обязательно простой k-й путь, то нужно использовать алгоритм Йены. Если же условие простоты пути отсутствует, то следует использовать алгоритм Эппштейна.

Список источников

- D.Eppstein Finding the k Shortest Paths: // D Eppstein - SIAM Journal on computing, 1998 - SIAM. URL: `https://www.ics.uci.edu/~eppstein/pubs/Epp-SJC-98.pdf`. (Дата выпуска: 31.03.1997)

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. Алгоритмы: Построение и анализ [2005]