

**Московский авиационный институт
(национальный исследовательский университет)**

**Факультет информационных технологий и прикладной
математики**

Кафедра вычислительной математики и программирования

Лабораторная работа №9 по курсу «Дискретный анализ»

Студент: А. А. Боглаев
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-22
Дата:
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №9

Задача: Разработать программу на языке C или C++, реализующую алгоритм согласно условию задачи.

Вариант 4: Поиск кратчайшего пути между парой вершин алгоритмом Дейкстры

Задан взвешенный неориентированный граф, состоящий из n вершин и m ребер. Вершины пронумерованы целыми числами от 1 до n . Необходимо найти длину кратчайшего пути из вершины с номером *start* в вершину с номером *finish* при помощи алгоритма Дейкстры. Длина пути равна сумме весов ребер на этом пути. Граф не содержит петель и кратных ребер.

Формат ввода:

В первой строке заданы $1 \leq n \leq 10^5$ и $1 \leq m \leq 10^5$, $1 \leq start \leq n$ и $1 \leq finish \leq n$. В следующих m строках записаны ребра. Каждая строка содержит три числа – номера вершин, соединенных ребром, и вес данного ребра. Вес ребра – целое число от 0 до 10^9 .

Формат вывода:

Необходимо вывести одно число – длину кратчайшего пути между указанными вершинами. Если пути между указанными вершинами не существует, следует вывести строку No solution.

1 Описание

Требуется реализовать алгоритм Дейкстры для поиска кратчайшего пути между двумя вершинами в неориентированном графе.

Согласно [1]: «Алгоритм Дейкстры решает задачу о кратчайшем пути из одной вершины во взвешенном ориентированном графе $G = (V, E)$ в том случае, когда веса ребер неотрицательны.»

Основным недостатком алгоритма Дейкстры является то, что он не работает с отрицательными весами ребер. Алгоритм Дейкстры является наиболее популярным способом решения данной задачи, так как является более производительным по сравнению с алгоритмом Беллмана-Форда.

Для решения задачи при помощи алгоритма Дейкстры, нужно задать расстояния от начальной вершины для всех остальных вершин. Для начальной вершины это расстояние будет 0 (расстояние до самой себя). Для остальных вершин это расстояние равно бесконечности (или числу, которое будет больше, чем максимально возможный вес ребра в графе).

Далее нужно создать неубывающую очередь с приоритетом для работы с парами (вершина и ее вес). Сначала поместим в очередь начальную вершину. Достанем ее из очереди и проверяем, посетили мы ее или нет. После того, как достали вершину, ее надо удалить из очереди. Далее делаем ослабление ребер (релоксацию), исходящих из этой вершины. Если мы ослабили ребро, его надо добавить в очередь со своим новым весом. Так алгоритм учтет изменения и сделает пересчет для всех ребер графа, которые исходят из добавленной вершины. И так пока не пройдем весь граф. Пока не выполним обход в ширину.

Как только очередь станет пустой, значит алгоритм закончил свою работу и мы нашли кратчайшие пути в графе.

Реализация на очереди с приоритетом имеет сложность: $O((V + E)\log V)$. Такую сложность можно получить при работе с разреженным графом.

Если же граф будет полным, то нужно использовать реализацию на массиве, которая имеет сложность $O(V^2 + E)$.

В данной задаче число ребер не превышает число вершин, поэтому можно воспользоваться реализацией на очереди с приоритетом.

2 Исходный код

Для начала определим тип данных для графа. Граф можно представить как матрицу (матрицу смежности). В языке программирования C++ это вектор векторов из пар (вершина и вес пути до нее).

Также необходимо задать бесконечность для корректной реализации алгоритма.

Для считывания графа из стандартного потока ввода запустим цикл по количеству ребер m и добавим в граф введеную информацию. Стоит отметить, что в условии задачи дан неориентированный граф, значит нужно добавить путь как из $u \rightarrow v$, так и из $v \rightarrow u$.

После, создадим вектор расстояний для каждой вершины и инициализируем его значениями бесконечности.

Важно отметить, что мы используем индексы с 0, поэтому нужно вычесть из каждого значения вершины 1.

Передадим в функцию для алгоритма Дейкстры граф, стартовую и конечную вершины, и вектор расстояний.

В функции алгоритма заменим расстояние от стартовой вершины до самой себя на 0. Создадим очередь с неубывающим порядком, которая будет принимать пару (вес, вершина). Такая структура пары поможет правильно сортировать вершины по весам внутри очереди. Далее добавляем в очередь стартовую вершину с весом 0.

Достаем из очереди вершину и начинаем проход по графу ослабляя ребра ее соседей (делаем релоксацию). И делаем так, пока не дойдем до конечной вершины.

Для вывода результата нужно проверить, если вес последней вершины равен бесконечности, то мы не можем до нее дойти. Иначе нужно вывести результат.

```
1 | #include <bits/stdc++.h>
2 |
3 | const int64_t INF = 1e18;
4 | using graph = std::vector<std::vector<pair<int, int64_t>>>>;
5 | using g_item = std::pair<int64_t, int>;
6 |
7 |
8 | void dijkstra(const graph &g, int u, int f, std::vector<int64_t> &d) {
9 |     d[u] = 0;
10 |     std::priority_queue<g_item, std::vector<g_item>, std::greater<g_item>> pq;
11 |     pq.push(std::make_pair(0, u));
12 |
13 |     while (!pq.empty()) {
14 |         g_item current = pq.top();
15 |         pq.pop();
16 |
17 |         u = current.second;
18 |     }
```

```

19     if (u == f) {
20         break;
21     }
22
23     if (current.first > d[u]) {
24         continue;
25     }
26
27     for (size_t i = 0; i < g[u].size(); i++) {
28         int v = g[u][i].first;
29         int64_t w = g[u][i].second;
30
31         if (d[u] + w < d[v]) {
32             d[v] = d[u] + w;
33             pq.push(std::make_pair(d[v], v));
34         }
35     }
36 }
37
38 if (d[f] != INF) {
39     std::cout << d[f] << std::endl;
40 } else {
41     std::cout << "No solution\n";
42 }
43 }
44
45
46 int main() {
47     int n, m, start, finish;
48     std::cin >> n >> m >> start >> finish;
49     graph g(n);
50
51     for (int i = 0; i < m; i++) {
52         int u, v;
53         int64_t w;
54         std::cin >> u >> v >> w;
55         u--;
56         v--;
57         g[u].push_back(std::make_pair(v, w));
58         g[v].push_back(std::make_pair(u, w));
59     }
60
61     std::vector<int64_t> d(n, INF);
62     start--;
63     finish--;
64
65     dijkstra(g, start, finish, d);
66     return 0;
67 }

```

3 Консоль

```
alex@wega:~/$ cat test.txt
5 6 1 5
1 2 2
1 3 0
3 2 10
4 2 1
3 4 4
4 5 5
alex@wega:~/$ ./main <test.txt
8
alex@wega:~/$ cat tests/test_1.t
20 25 13 6
1 19 12
9 20 84
6 10 38
4 12 91
11 14 29
3 20 79
4 20 12
11 20 41
8 17 66
11 13 17
5 11 64
19 20 8
1 3 63
9 14 85
7 10 9
7 10 67
10 12 5
5 10 28
16 18 50
6 13 44
17 18 70
7 18 58
4 5 89
11 15 46
14 20 84
alex@wega:~/$ ./main <tests/test_1.t
44
```

```
alex@wega:~/$ ./main
3 1 2 3
1 2 5
No solution
```

4 Тест производительности

В тесте производительности сравним два алгоритма обхода графов: алгоритм Дейкстры и Беллмана-Форда. В тестах не будет ребер с отрицательным весом.

Тестировать будем на графах с количеством вершин: 10, 20, 100, 1000, 10000. Количество ребер будет примерно равно количеству вершин.

```
alex@wega:~/$ ./wrapper.sh
Stage #4 Benchmarking...
Dijkstra: 0.007 ms
Bellman-Ford: 0.005 ms
alex@wega:~/$ ./wrapper.sh
Stage #4 Benchmarking...
Dijkstra: 0.007 ms
Bellman-Ford: 0.009 ms
alex@wega:~/$ ./wrapper.sh
Stage #4 Benchmarking...
Dijkstra: 0.010 ms
Bellman-Ford: 0.542 ms
alex@wega:~/$ ./wrapper.sh
Stage #4 Benchmarking...
Dijkstra: 0.766 ms
Bellman-Ford: 21.985 ms
alex@wega:~/$ ./wrapper.sh
Stage #4 Benchmarking...
Dijkstra: 2.887 ms
Bellman-Ford: 2487.852 ms
```

Как видно, алгоритм Дейкстры является более производительным по сравнению с алгоритмом Беллмана-Форда. Такой результат получается потому, что алгоритм Дейкстры использует жадный подход для поиска наименьшего пути. Жадный подход позволяет решать задачу более эффективно. Мы стараемся взять ребро с наименьшим весом при помощи очереди приоритетов. Алгоритм Беллмана-Форда будет проходить по графу, пока не пройдет $n - 1$ итерацию, что является неэффективным при работе с большими графами.

5 Выводы

Выполнив девятую лабораторную работу по курсу «Дискретный анализ», я ознакомился с графами и изучил способы представления графов в компьютере. Также я познакомился с базовыми алгоритмами поиска кратчайших путей в графе (алгоритм Дейкстры и алгоритм Беллмана-Форда).

Алгоритм Дейкстры, основанный на жадном подходе, эффективно находит кратчайшие пути от одной вершины до всех остальных в графах с неотрицательными весами рёбер. Его временная сложность составляет $O((V + E)\log V)$ при использовании очереди с приоритетом, что делает его подходящим для разреженных графов. Однако алгоритм не может корректно обрабатывать графы с отрицательными весами.

С другой стороны, алгоритм Беллмана-Форда, хотя и менее эффективный с временной сложностью $O(V * E)$, способен работать с графами, содержащими отрицательные веса, и даже обнаруживать отрицательные циклы. Это делает его более универсальным инструментом для решения задач, где возможны такие условия.

Графы широко используются в нашей повседневной жизни. Например, построение кратчайшего маршрута до дома. Или выбор места строительства нового торгового центра, нужно построить в быстро-доступном для всех месте.

Список литературы

- [1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание*. — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))
- [2] *Алгоритм Дейкстры - Алгоритмика*
URL: <https://ru.algorithmica.org/cs/shortest-paths/dijkstra/> (дата обращения: 10.12.2024).
- [3] *Алгоритм Беллмана-Форда - Хабр*
URL: <https://habr.com/ru/companies/otus/articles/484382/> (дата обращения: 10.12.2024).