**Алгоритм Дейкстры**

**Алгоритм Дейкстры**

**Алгоритм Дейкстры** назван в честь голландского ученого Эдсгера Дейкстры (Edsger Dijkstra). Алгоритм был предложен в 1959 году для нахождения кратчайших путей от одной вершины до всех остальных в ориентированном взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса.

Рассмотрим две модели хранения взвешенного графа в памяти. В первой модели (матрица весов, аналог матрицы смежности) будем считать, что вес ребра из вершины i в вершину j равен w[i][j], то есть в матрице w хранятся веса ребра для любых двух вершин. Если из вершины i в вершину j нет ребра, то w[i][j]==INF для некоторого специального значения константы INF. Значение INF следует выбирать исходя из задачи, например, если речь идет о расстояниях между какими-либо населенными пунктами Земли, то можно выбрать значение INF равным 109 километров.

Алгоритм Дейкстры относится к так называемым «жадным» алгоритмам. Пусть расстояние от начальной вершины start до вершины i хранится в массиве dist[i]. Начальные значения dist[start]=0, dist[i]=INF для всех остальных вершин i. То есть в самом начале алгоритму известен путь из вершины start до вершины start длины 0, а до остальных вершин кратчайшие пути неизвестны. Между тем алгоритм будет постепенно улучшать значения в массиве dist, в результате получит кратчайшие расстояния до всех вершин.

Основная идея для улучшения называется «релаксацией ребра». Пусть из вершины i в вершину j есть ребро веса w[i][j], при этом выполнено неравенство dist[i] + w[i][j] < dist[j]. То есть можно построить маршрут из начальной вершины до вершины i и добавить к нему ребро из i в j, и суммарная стоимость такого маршрута будет меньше, чем известная ранее стоимость маршрута из начальной вершины в вершину j. Тогда можно улучшить значение dist[j], присвоив dist[j] = dist[i] + w[i][j].

В алгоритме Дейкстры вершины красятся в два цвета, будем говорить, что вершина «неокрашенная» или «окрашенная». Изначально все вершины неокрашенные. Если алгоритм Дейкстры покрасил вершину i, то это означает, что найденное значение dist[i] является наилучшим возможным и в последствии не будет улучшаться, то есть значение dist[i] является кратчайшим расстоянием от начальной вершины до вершины i. Если же вершина не покрашена, то величина dist[i] для такой вершины i равна кратчайшему пути из вершины start до вершины i, который проходит только по покрашенным вершинам (за исключением самой вершины i).

На каждом шаге алгоритма Дейкстры красится одна новая вершина. В качестве такой вершины выбирается неокрашенная вершина i с наименьшим значением D[i]. Затем рассматриваются все ребра, исходящие из вершины i, и производится релаксация этих ребер, то есть улучшаются расстояния до вершин, смежных с i.

Алгоритм заканчивается, когда на очередном шаге не останется неокрашенных вершин или если расстояние до всех неокрашенных вершин будет равно INF (то есть эти вершины являются недостижимыми).

Запишем алгоритм Дейкстры. Пусть n — число вершин в графе, вершины пронумерованы от 0 до n−1. Номер начальной вершины — start и веса ребер хранятся в матрице w.

**Реализация на языке C++**

**const int INF = 1000000000;**

**vector <int> dist(n, INF);**

**dist[start] = 0;**

**vector <bool> used(n);**

**int min\_dist = 0;**

**int min\_vertex = start;**

**while (min\_dist < INF)**

**{**

**int i = min\_vertex;**

**used[i] = true;**

**for (int j = 0; j < n; ++j)**

**if (dist[i] + w[i][j] < dist[j])**

**dist[j] = dist[i] + w[i][j];**

**min\_dist = INF;**

**for (int j = 0; j < n; ++j)**

**if (!used[j] && dist[j] < min\_dist)**

**{**

**min\_dist = dist[j];**

**min\_vertex = j;**

**}**

**}**

**Реализация на языке Python**

**INF = 10 \*\* 10**

**dist = [INF] \* n**

**dist[start] = 0**

**used = [False] \* n**

**min\_dist = 0**

**min\_vertex = start**

**while min\_dist < INF:**

**i = min\_vertex**

**used[i] = True**

**for j in range(n):**

**if dist[i] + w[i][j] < dist[j]:**

**dist[j] = dist[i] + w[i][j]**

**min\_dist = INF**

**for j in range(n):**

**if not used[j] and dist[j] < min\_dist:**

**min\_dist = dist[j]**

**min\_vertex = j**

Массив used будет хранить информацию о том, была ли покрашена вершина. Сначала инициализируются массивы dist и used. Затем запускается внешний цикл алгоритма, который выбирает неокрашенную вершину с минимальным расстоянием, номер этой вершины хранится в переменной min\_vertex, а расстояние до этой вершины — в переменной min\_dist. Если же min\_dist оказывается равно INF, то значит все неокрашенные вершины являются недостижимыми и алгоритм заканчивает свою работу. Иначе найденная вершина окрашивается и после этого релаксируются все ребра, исходящие из этой вершины.

Данный алгоритм имеет сложность O(n2), так как внешний цикл может быть выполнен до n раз, внутри него содержится два цикла, каждый из которых также выполняется n раз.

Для восстановления ответа, то есть для нахождения пути из начальной вершины до всех остальных, необходимо построить дерево кратчайших путей. Это дерево будет состоять из тех ребер, которые были успешно срелаксированы в результате исполнения алгоритма. То есть если происходит релаксация ребра из i в j, то теперь кратчайший маршрут из вершины start до вершины j должен проходить через вершину i и затем содержать ребро i-j. Тем самым вершина i становится предшественником вершины j на кратчайшем пути из начальной вершины до вершины j.

Рассмотрим реализацию алгоритм Дейкстры с восстановлением ответа на графе, хранимым в виде списка смежности на языке C++. Ребро из вершины i в вершину j веса wt будет хранить в виде пары (j, wt), список ребер, исходящих из вершины i будет храниться в векторе w[i]. То есть списки смежности w будут объявлены так:

**vector <vector <pair <int, int > > > w;**

Реализация считывания ребер графа (из номеров вершин будем вычитать число 1 для нумерации с нуля, рассматриваем ориентированный граф, то есть не дублируем ребро):

**for (k = 0; k < m; ++k)**

**{**

**int i, j, wt;**

**cin >> i >> j >> wt;**

**w[i - 1].push\_back(make\_pair(j - 1, wt));**

**}**

Тогда при обработки вершины i вместо перебора всех других вершин мы рассматриваем только ребра, исходящие из данной вершины.

**const int INF = 1000000000;**

**vector <int> dist(n, INF);**

**dist[start] = 0;**

**vector <int> prev(n, -1);**

**vector <bool> used(n);**

**int min\_dist = 0;**

**int min\_vertex = start;**

**while (min\_dist < INF)**

**{**

**int i = min\_vertex;**

**used[i] = true;**

**for (auto edge: w[i])**

**{**

**int j = edge.first;**

**int wt = edge.second;**

**if (dist[i] + wt < dist[j])**

**{**

**dist[j] = dist[i] + wt;**

**prev[j] = i;**

**}**

**}**

**min\_dist = INF;**

**for (int j = 0; j < n; ++j)**

**if (!used[j] && dist[j] < min\_dist)**

**{**

**min\_dist = dist[j];**

**min\_vertex = j;**

**}**

**}**

Восстановление ответа производится аналогично поиску в ширину или в глубину.

**vector <int> path;**

**while (j != -1)**

**{**

**path.push\_back(j);**

**j = prev[j];**

**}**

**reverse(path.begin(), path.end());**

Рассмотрим реализацию алгоритм Дейкстры с восстановлением ответа на графе, хранимым в виде списка смежности на языке Python. Набор вершин, смежных с вершиной i будет храниться в множестве w[i]. Также необходимо хранить веса ребер, будем считать, что для хранения весов ребер используется словарь weight, где ключом является кортеж из двух вершин. То есть вес ребра из i в j хранится в элементе weight[i, j] словаря весов.

**dist = [INF] \* n**

**dist[start] = 0**

**prev = [None] \* n**

**used = [False] \* n**

**min\_dist = 0**

**min\_vertex = start**

**while min\_dist < INF:**

**i = min\_vertex**

**used[i] = True**

**for j in w[i]:**

**if dist[i] + weight[i, j] < dist[j]:**

**dist[j] = dist[i] + weight[i, j]**

**prev[j] = i**

**min\_dist = INF**

**for i in range(n):**

**if not used[i] and dist[i] < min\_dist:**

**min\_dist = dist[i]**

**min\_vertex = i**

Для нахождения кратчайшего пути из вершины start до вершины j будем переходить от каждой вершины к ее предшественнику:

**path = []**

**while j is not None:**

**path.append(j)**

**j = prev[j]**

**path = path[::-1]**

Алгоритм Дейкстры применим только в том случае, когда веса всех ребер неотрицательные. Это гарантирует то, что после окраски расстояние до вершины не может быть улучшено. Если в графе могут быть ребра отрицательного веса, то следует использовать другие алгоритмы.

**Реализация алгоритма Дейкстры с использованием кучи или контейнера set в STL**

Алгоритм Дейкстры в ранее приведенной реализации имеет сложность O(n2). В этой реализации производится выбор элемента с наименьшим расстоянием до него, что производится путем просмотра всех вершин. Если хранить все неокрашенные вершины в куче или в контейнере set STL (которое реализовано при помощи сбалансированного дерева поиска), то поиск очередной вершины для окрашивания можно производить более оптимально.

Но обновление расстояния до другой  в этом случае будет выполняться за O(log⁡n), так как это требует перестройки кучи или дерева поиска. Если в графе m ребер, то максимальное число релаксаций ребер также будет не больше m и суммарная сложность всех релаксаций будет O(mlog⁡n). Таким образом, алгоритм Дейкстры с использованием кучи будет иметь сложность O(nlog⁡n+mlog⁡n)=O((n+m)log⁡n). Если граф — разреженный, то такой алгоритм работает существенно быстрее, чем обычный алгоритм Дейкстры, но на плотных графах (если m∼n2) он, наоборот, менее эффективен, чем простая реализация Дейкстры.

Приведем реализацию алгоритма Дейкстры с использованием структуры set на языке STL. Нам необходимо извлекать из set элемент с наименьшим расстоянием до него, а также узнавать при этом номер извлеченной вершины. Для этого в структуре set мы будем хранить пары значений (dist[i], i), то есть пары, у который первый компонент - расстояние до вершины, второй компонент - номер вершины. Поскольку пары сортируются лексикографически по полю first, а при равенстве - по полю second, то объекты в нашем set будут упорядочены прежде всего по возрастанию значения dist и в начале set будет храниться вершина с минимальным значением set.

**vector<int> dist(n, INF);**

**dist[start] = 0;**

**set<pair<int, int> > unused;**

**unused.insert(make\_pair(0, start));**

**while (!unused.empty())**

**{**

**int i = unused.begin()->second;**

**unused.erase(unused.begin());**

**for (auto edge : w[i])**

**{**

**int j = edge.first;**

**int wt = edge.second;**

**if (dist[i] + wt < dist[j])**

**{**

**unused.erase(make\_pair(dist[j], j));**

**dist[j] = dist[i] + wt;**

**unused.insert(make\_pair(dist[j], j));**

**}**

**}**

**}**

При этом в set мы будем хранить только вершины, которые достижимы из начальной, то есть для них значение dist меньше 0. Поэтому в самом начале в set кладется только одна начальная вершина. Внутри главного цикла из set удаляется начальный элемент, а дальше при обработке вершины рассматриваются все исходящие из нее ребра и для каждого ребра соответствующая ему пара удаляется из set (если вершина не была ранее достижима, т.е. до нее расстояние было равно INF, то операция удаления ничего не сделает, но и не приведет к ошибке), а потом в set добавляется новая пара, соответствующая измененному значению расстояния до данной вершины.