Графы
Кратчайшие пути

education

Графы: кратчайшие пути

- **Кратчайший путь между двумя вершинами в графе** — это путь с минимальной суммой весов рёбер
- Выбор конкретного алгоритма зависит от специфики задачи и графа

education

Графы: кратчайшие пути

1. Алгоритм Дейкстры
2. Алгоритм Беллмана — Форда
3. Алгоритм Флойда — Уоршелла
4. Алгоритм Джонсона
   Алгоритм A*

Графы: кратчайшие пути

Взвешенный граф

- Содержит дополнительную информацию — вес каждого ребра. Этот вес может представлять расстояние между двумя точками, стоимость перехода от одной точки к другой, время, затраченное на перемещение, и так далее.
- Вес ребра в графе — это численное значение, которое отображает «стоимость» перехода от одной вершины к другой.

Ориентированный граф (дограф)

- В ориентированном графе рёбра имеют направление. Это означает, что движение возможно только в указанном направлении.
- Например, если есть ребро из вершины A к вершине B, это не гарантирует, что существует путь из B в A.

Алгоритмы поиска кратчайшего пути

Алгоритм Дейкстры

- Находит кратчайший путь от одной заданной вершины до всех остальных вершин в графе.

- Он работает только с графами, в которых все веса рёбер неотрицательны. Алгоритм работает путём последовательного «расслабления» рёбер графа. «Расслабление» ребра в данном контексте означает процесс улучшения текущей оценки кратчайшего пути.

Алгоритмы поиска кратчайшего пути

Алгоритм Беллмана — Форда

- Находит кратчайшие пути от одной вершины до всех остальных, но он может работать и с отрицательными весами рёбер.
- В графах, содержащих циклы отрицательного веса, не существует определения «кратчайшего пути». Алгоритм Беллмана — Форда также расслабляет рёбра, но делает это V − 1 раз, где V — количество вершин в графе.

Это гарантирует, что мы получим кратчайший путь даже в случае наличия отрицательных весов при условии,
что в графе нет циклов отрицательного веса.

Алгоритмы поиска кратчайшего пути

Алгоритм Флойда — Уоршелла

- Находит кратчайшие пути между всеми парами вершин и может работать с графами, в которых веса рёбер могут быть и положительными, и отрицательными.

- Алгоритм состоит из трёх вложенных циклов, которые последовательно рассматривают все вершины графа как возможные промежуточные точки на пути от одной вершины к другой. Этот алгоритм позволяет нам получить кратчайшие пути между всеми парами вершин. Однако так же, как и в случае алгоритма Беллмана — Форда, в графе не должно быть циклов отрицательного веса.

Другие алгоритмы

1

Алгоритм Джонсона: подходит для поиска кратчайших путей в графах с отрицательными весами.

2

Алгоритм A*: используется в области искусственного интеллекта для поиска пути в пространстве с множеством состояний.

BFS и поиск кратчайших путей

BFS (поиск в ширину)

- BFS идеально подходит для поиска кратчайшего пути в невзвешенном графе, где «кратчайший» означает путь с наименьшим количеством рёбер.
- BFS работает, обрабатывая вершины в графе по уровням, начиная от исходной вершины.
- BFS не учитывает веса рёбер, так что этот подход не будет работать в взвешенных графах.

DFS и поиск кратчайших путей

DFS (поиск в глубину)

- Не используется для поиска кратчайших путей:
  он будет глубоко проникать в граф, следуя каждому пути до конца,
  прежде чем перейти к следующему пути.
- DFS может «пропустить» более короткий путь, который был бы доступен,
  если бы он выбрал другой путь для исследования.

Реализация

```
#include <iostream>
#include <functional>
#include <map>
#include <queue>
#include <list>
#include <climits>
#include <vector>
#include <string>
```

```
class Node {
public:
    int id;
    std::string info;

    Node(int id, std::string info) : id(id), info(std::move(info)) {}
};

class Edge {
public:
    int node;
    int weight;

    Edge(int node, int weight) : node(node), weight(weight) {}
};

class Graph {
private:
    std::map<int, std::list<Edge>> adjList; // adjacency list
    std::map<int, std::shared_ptr<Node>> nodes; // map to store the Node objects
```

Реализация

```
#include <iostream>
#include <functional>
#include <map>
#include <queue>
#include <list>
#include <climits>
#include <vector>
#include <string>
```

```
class Node {
public:
    int id;
    std::string info;

    Node(int id, std::string info) : id(id), info(std::move(info)) {}
};

class Edge {
public:
    int node;
    int weight;

    Edge(int node, int weight) : node(node), weight(weight) {}
};

class Graph {
private:
    std::map<int, std::list<Edge>> adjList; // adjacency list
    std::map<int, std::shared_ptr<Node>> nodes; // map to store the Node
objects
```

education

Реализация

```
#include <iostream>
#include <functional>
#include <map>
#include <queue>
#include <list>
#include <climits>
#include <vector>
#include <string>
```

```
class Node {
public:
    int id;
    std::string info;

    Node(int id, std::string info) : id(id), info(std::move(info)) {}
};
```

```
class Edge {
public:
    int node;
    int weight;

    Edge(int node, int weight) : node(node), weight(weight) {}
};

class Graph {
private:
    std::map<int, std::list<Edge>> adjList; // adjacency list
    std::map<int, std::shared_ptr<Node>> nodes; // map to store the Node objects
```

Реализация

```
#include <iostream>
#include <functional>
#include <map>
#include <queue>
#include <list>
#include <climits>
#include <vector>
#include <string>
```

```
class Node {
public:
    int id;
    std::string info;

    Node(int id, std::string info) : id(id), info(std::move(info)) {}
};
```

```
class Edge {
public:
    int node;
    int weight;

    Edge(int node, int weight) : node(node), weight(weight) {}
};
```

```
class Graph {
private:
    std::map<int, std::list<Edge>> adjList; // adjacency list
    std::map<int, std::shared_ptr<Node>> nodes; // map to store the Node objects
```

Реализация

```
void AddNode(int id, std::string info) {
    std::shared_ptr<Node> newNode = std::make_shared<Node>(id, std::move(info));
    nodes[id] = newNode;
}

void AddEdge(int id1, int id2, int weight) {
    adjList[id1].push_back(Edge(id2, weight));
    adjList[id2].push_back(Edge(id1, weight)); // for undirected graph
}
```

```
void Dijkstra(int startId) {
    std::map<int, int> dist;
    for (const auto& node : nodes) {
        dist[node.first] = INT_MAX;
    }
    dist[startId] = 0;

    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, Compare> pq;
    pq.push({ startId, 0 });

    while (!pq.empty()) {
        int node = pq.top().first;
        pq.pop();

        for (auto& edge : adjList[node]) {
            if (dist[node] + edge.weight < dist[edge.node]) {
                dist[edge.node] = dist[node] + edge.weight;
                pq.push({ edge.node, dist[edge.node] });
            }
        }
    }
```

Дейкстра

```
void Dijkstra(int startId) {
    std::map<int, int> dist;
    for (const auto& node : nodes) {
        dist[node.first] = INT_MAX;
    }
    dist[startId] = 0;

    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, Compare> pq;
    pq.push({ startId, 0 });

    while (!pq.empty()) {
        int node = pq.top().first;
        pq.pop();

        for (auto& edge : adjList[node]) {
            if (dist[node] + edge.weight < dist[edge.node]) {
                dist[edge.node] = dist[node] + edge.weight;
                pq.push({ edge.node, dist[edge.node] });
            }
        }
    }

    // Print shortest distances
    for (auto node : nodes) {
        std::cout << "Distance to node " << node.first << " = " << dist[node.first] << std::endl;
    }
}
```

Инициализация расстояний

```
 std::map<int, int> dist;
for (const auto& node : nodes) {
    dist[node.first] = INT_MAX;
}
dist[startId] = 0;
```

Инициализация очереди с приоритетами

```
std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, Compare> pq;
pq.push({ startId, 0 });
```

Основной цикл алгоритма:

```
  while (!pq.empty()) {
    int node = pq.top().first;
    pq.pop();

    for (auto& edge : adjList[node]) {
        if (dist[node] + edge.weight < dist[edge.node]) {
            dist[edge.node] = dist[node] + edge.weight;
            pq.push({ edge.node, dist[edge.node] });
        }
    }
}
```

Вывод результатов

```
for (auto node : nodes) {
        std::cout << "Distance to node " << node.first << " = " << dist[node.first] << std::endl;
}
```

Беллман — форд

```
void BellmanFord(const int startId) {
    std::map<int, int> dist;
    for (auto node : nodes) {
        dist[node.first] = INT_MAX;
    }
    dist[startId] = 0;
    for (unsigned int i = 1; i <= nodes.size() - 1; i++) {
        for (const auto& node : adjList) {
            for (auto edge : node.second) {
                if (dist[node.first] != INT_MAX && dist[node.first] + edge.weight < dist[edge.node]) {
                    dist[edge.node] = dist[node.first] + edge.weight;
                }
            }
        }
    }

    // Check for negative-weight cycles
    for (const auto& node : adjList) {
        for (auto& edge : node.second) {
            if (dist[node.first] != INT_MAX && dist[node.first] + edge.weight < dist[edge.node]) {
                std::cout << "Graph contains negative-weight cycle" << std::endl;
                return;
            }
        }
    }

    // Print shortest distances
    for (const auto& node : nodes) {
        std::cout << "Distance to node " << node.first << " = " << dist[node.first] << std::endl;
    }
}
```

```
void FloydWarshall() {
                    const int V = nodes.size();
    std::vector<std::vector<int>> dist(V, std::vector<int>(V, INT_MAX));

    for (const auto& node : adjList) {
        dist[node.first][node.first] = 0;
        for (auto edge : node.second) {
            dist[node.first][edge.node] = edge.weight;
        }
    }

    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print shortest distances
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INT_MAX) {
                std::cout << "INF ";
            }
            else {
                std::cout << dist[i][j] << " ";
            }
        }
        std::cout << std::endl;
    }
}
```

Общий вывод

```
Graph g;

    // Создаем узлы графа
    g.AddNode(0, "Node 0");
    g.AddNode(1, "Node 1");
    g.AddNode(2, "Node 2");
    g.AddNode(3, "Node 3");
    g.AddNode(4, "Node 4");

    // Создаем ребра графа
    g.AddEdge(0, 1, 1);
    g.AddEdge(0, 2, 3);
    g.AddEdge(1, 2, 2);
    g.AddEdge(2, 3, 1);
    g.AddEdge(3, 4, 1);
    g.AddEdge(1, 4, 5);

    std::cout << "Dijkstra's algorithm:\n";
    g.Dijkstra(0);

    std::cout << "\nBellman-Ford algorithm:\n";
    g.BellmanFord(0);

    std::cout << "\nFloyd-Warshall algorithm:\n";
    g.FloydWarshall();
```

Общий вывод

```
// Добавляем вершины в граф
g.AddNode(0,"First");
g.AddNode(1,"Second");
g.AddNode(2,"Third");
g.AddNode(3,"Fourth");
// Добавляем рёбра между вершинами
g.AddEdge(0, 1, 2);
g.AddEdge(1, 2, 3);
g.AddEdge(0, 3, 5);
g.AddEdge(2, 3, -1);
```

Общий вывод

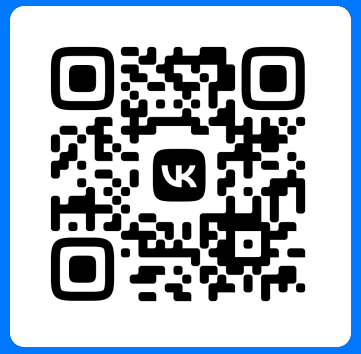
```
Bellman-Ford algorithm:
Graph contains negative-weight cycle

Floyd-Warshall algorithm:
0 2 3 2
2 0 1 0
3 1 -2 -3
2 0 -3 -4
```

Общий вывод

Строка из вывода алгоритма Флойда — Варшалла: 2 0 1 0.

- Расстояние от второй вершины до первой (или от вершины 1 до вершины 2) равно 2.
- Расстояние от второй вершины до второй (то есть до самой себя) равно 0.
- Расстояние от второй вершины до третьей равно 1.
- Расстояние от второй вершины до четвёртой равно 0.

Это означает, что кратчайший путь от второй вершины до первой, третьей и четвёртой вершин составляет 2, 1 и 0 единиц соответственно. Расстояние до самой себя всегда равно 0.



Будем ВКонтакте!