Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Тема: алгоритм Беллмана-Форда

Выполнил А.Д. Фиронов

студент гр. 3331506/90401

Руководитель М.С. Ананьевский

«   »________2022г.

Санкт-Петербург

2022 г.

Оглавление

Описание алгоритма 3

Реализация алгоритма 3

Анализ алгоритма 5

Применение алгоритма 6

Список литературы 7

Приложение 1. Код программы 8

Описание алгоритма

Алгоритм Беллмана-Форда – это алгоритм на графах, решающий задачу нахождения кратчайших путей от одной из вершин ориентированного графа до всех остальных. В отличие от алгоритма Дейкстры, этот алгоритм корректно работает при наличии в графе рёбер отрицательного веса.

Для каждой вершины графа создаётся переменная – метка. В начале работы алгоритма метка исходной вершины равна нулю, метки всех остальных вершин – бесконечности.

В процессе работы на каждой итерации алгоритм посещает одну из вершин. Среди вершин графа, которые не посещались ранее, выбирается вершина с минимальным значением метки (при первой итерации это всегда начальная вершина). Далее для каждой вершины, имеющей с посещаемой на данной итерации общее ребро и не посещённой ранее, вес общего ребра складывается с весом метки посещаемой вершины. Если вычисленная сумма меньше собственной метки смежной вершины, оно записывается в эту метку.

Итерации повторяются до тех пор, пока все вершины графа не будут посещены.

Реализация алгоритма

В ходе работы алгоритм реализован на языке C++.

В классе Ford реализованы функции addEdge, calculate и find.

addEdge – заводит в матрицу смежности (*graph*) данные о наличии рёбер из одной вершины в другую, а также о весе этого ребра.

find – проверяет наличие вершины в массиве *nodes*, тем самым защищая от повторной записи одной и той же вершины.

calculate – занимается непосредственно расчётом веса пути. Первым шагом проверяется наличие ребра между вершинами (из таблицы смежности), обновляет значения пути для всех вершин: путь до исходной вершины 0, до всех остальных – бесконечность (*infinity*).

Далее, происходит расчёт веса пути: текущее значение из массива *database* сравнивается с новым – весом пути до родительского нода и весом ребра, соединяющего родительский нод с дочерним.

В случае нахождения более оптимального маршрута – в массиве *database* обновляется вес пути.

Перерасчёт происходит n-1 раз. Такое ограничение верхней границы обеспечивает невозможность программы уйти в отрицательный цикл.

Рассмотрим результат работы программы на примере графа, представленного на рисунке 1.

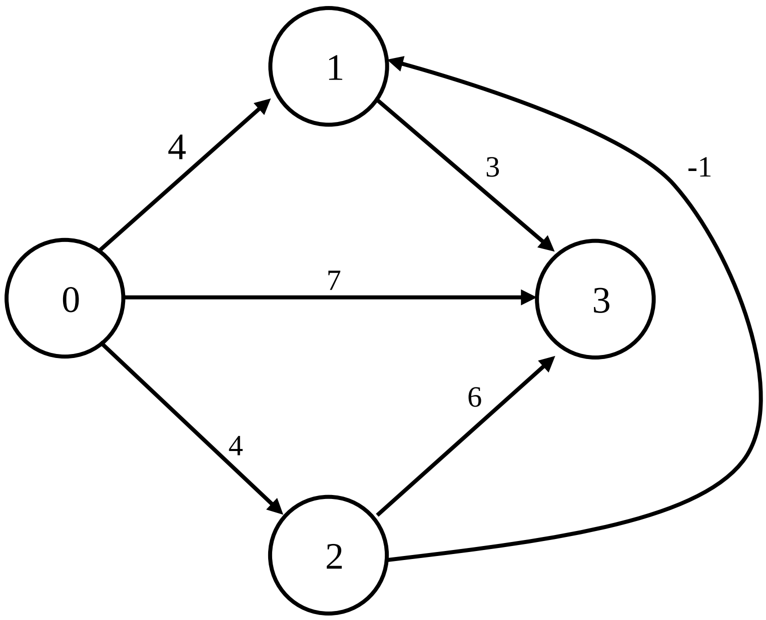
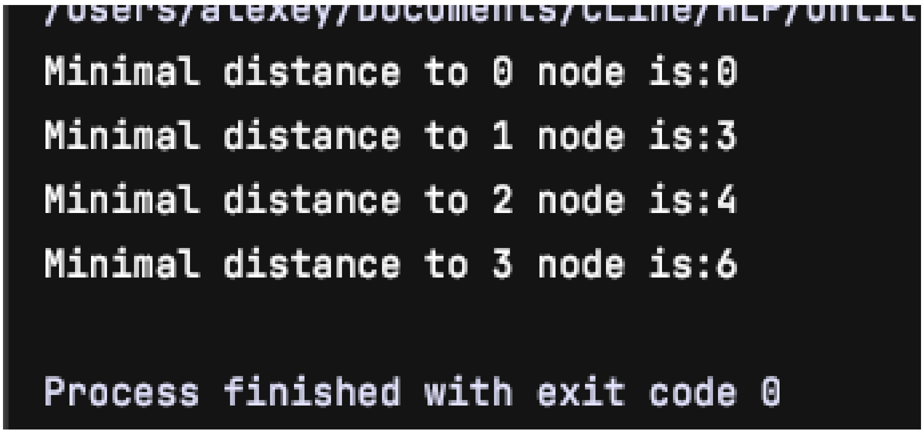


Рисунок 1 – Рассматриваемый граф

Результат работы программы и входные данные представлены на рисунке 2.



```
Minimal distance to 0 node is:0
Minimal distance to 1 node is:3
Minimal distance to 2 node is:4
Minimal distance to 3 node is:6

Process finished with exit code 0
```

Рисунок 2 – Результат работы программы

На выходе программа выдаёт правильные данные: кратчайший путь до соответствующей вершины.

Анализ алгоритма

Основной трудностью при реализации алгоритма является возможность образования отрицательного цикла: замкнутого пути с отрицательным суммарным весом. В связи с этим приходится вводить дополнительное ограничение на число итераций: их число должно быть $n-1$, где n – число вершин графа. Также неприятностью является

Временная сложность алгоритма – $O(n \cdot m)$.

В таблице 1 представлена зависимость времени выполнения алгоритма от количества вершин (для простоты, считаем все вершины соединёнными между собой), а на рисунке 3 – соответствующий график этой зависимости.

Таблица 1 – Зависимость времени выполнения t от числа вершин n

n	2	5	10	18	24	30
t, мс	235	1 280	4 068	15 147	26 691	42 084

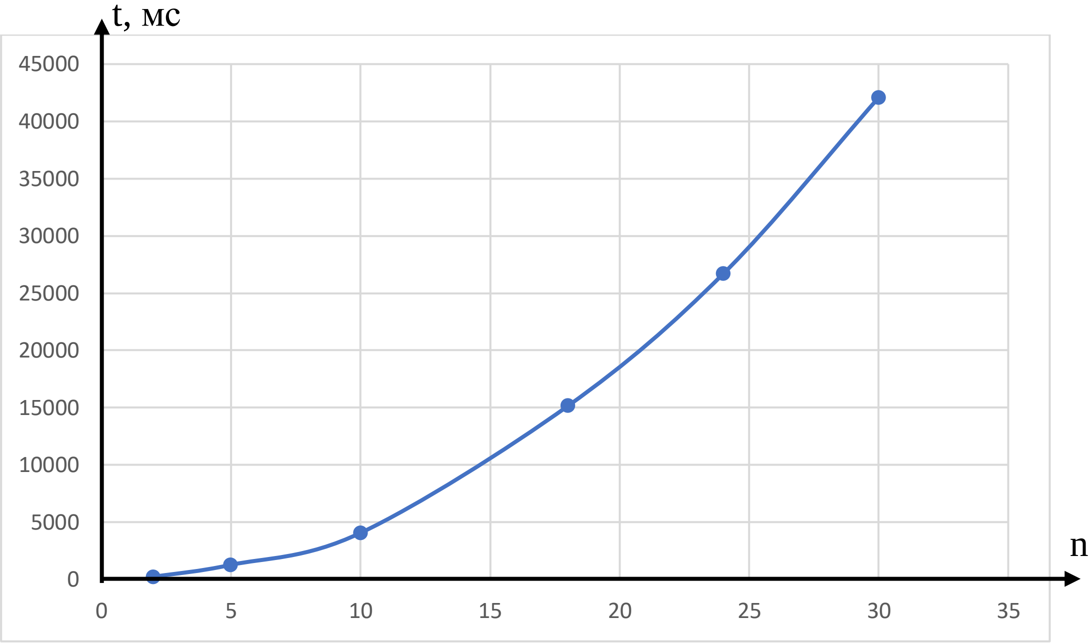


Рисунок 3 – График зависимости времени выполнения от количества вершин

Применение алгоритма

Алгоритм может применяться в задачах построение кратчайшего маршрута, а также для поиска наличия отрицательных циклов в графе. Для реализации последнего – необходимо увеличить количество выполняемых итераций с $n - 1$ до n. В случае изменения выходных параметров на последнем шаге, можно утверждать о наличии отрицательного цикла.

Список литературы

1. Алгоритм Дейкстры [Электронный ресурс] http://comp-science.narod.ru/KPG/Deikstr.htm
2. Алгоритм Белмана-Форда [Электронный ресурс] http://comp-science.narod.ru/KPG/BelmanFord.htm
3. Нахождение отрицательного цикла в графе [Электронный ресурс] http://e-maxx.ru/algo/negative_cycle
4. Алгоритм Белмана-Форда [Электронный ресурс] https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%B5%D0%BB%D0%BB%D0%BC%D0%B0%D0%BD%D0%B0_%E2%80%94_%D0%A4%D0%BE%D1%80%D0%B4%D0%B0
5. Алгоритм форда белмана [Электронный ресурс] https://www.cyberforum.ru/cpp-beginners/thread2435646.html

Приложение 1. Код программы

```
#include <iostream>
#include <vector>
#include <limits>
using namespace std;

class Ford{
private:
    int noOfNodes;
    int length;
    vector<vector<int>> graph;
    vector<vector<int>> database;
public:
    void addEdge(int i, int j, int cost);
    void calculate(int startNode);
    bool find(vector<int> nodes, int number);
    Ford(int length, int n){
        this->noOfNodes = n;
        this->length = length;
        this->graph.resize(length);
        this->database.resize(length);
        for (int i=0; i < length; i++){
            this->database[i].resize(2);
            for(int j = 0; j < 2; j++){
                this->database[i][j]=0;
            }
            this->graph[i].resize(length);
            for(int j=0; j < length; j++){
                this->graph[i].push_back(0);
            }
        }
    }
};

bool Ford::find(vector<int> nodes, int number){
    for(int i=0; i<nodes.size(); i++){
        if (nodes[i] == number){
            return false;
        }
    }
    return true;
}


void Ford::addEdge(int i, int j, int cost){
    this->graph[i][j] = cost;
}


void Ford::calculate(int startNode){
    int infinity = numeric_limits<int>::max();
    vector<int> nodes;
    for(int i=0; i<this->length; i++){
        for (int j = 0; j<this->length; j++){
            if (this->graph[i][j] != 0){
                if(this->find(nodes, i)){
                    if(i == startNode) {
                        this->database[i][0] = 0;
                        this->database[i][1] = 0;
                    }else{
                        this->database[i][0] = infinity;
                        this->database[i][1] = 0;
```

```
                    }
                    nodes.push_back(i);
                }
                if (this->find(nodes, j)){
                    if(j == startNode) {
                        this->database[j][0] = 0;
                        this->database[j][1] = 0;
                    }else{
                        this->database[j][0] = infinity;
                        this->database[j][1] = 0;
                    }
                    nodes.push_back(j);
                }
            }
        }
    }


    for (int k=0; k<this->noOfNodes-1; k++){
        for (int i = 0; i < nodes.size(); i++) {
            int prev = this->database[nodes[i]][0];
            for (int j = 0; j < this->length; j++) {
                if (this->graph[nodes[i]][j] != 0) {
                    if (prev < infinity) {
                        int cost = prev + this->graph[nodes[i]][j];
                        if (cost < database[j][0]) {
                            database[j][0] = cost;
                            database[j][1] = i;

                        }
                    }
                }
            }
        }
    }

    for(int node : nodes){
        cout << "Minimal distance to "<<node<<" node is:"<<this-
>database[node][0]<<'\n';
    }



}
int main()
{
    Ford ford = Ford(4,4);
    ford.addEdge(0,1,4);
    ford.addEdge(0,2,4);
    ford.addEdge(0,3,7);
    ford.addEdge(2,1,-1);
    ford.addEdge(1,3,3);
    ford.addEdge(2,3,6);

    ford.calculate(0);
    return 0;
}
```