Най-кратък път в мрежа

Денис Зангъров

F95108

Кратко резюме

Ще имплементираме „универсален“ алгоритъм, който е частично базиран на алгоритъма на Дийкстра за намиране на най-кратък път в смесен граф – бил той претеглен или не. Ще преминем през всички основни понятия, от които се нуждаем, представянето - нашите абстракции, структурите от данни, които ще използваме(и напишем) и алгоритмите за обхождане.

Основни понятия

Математическо определение:

Графът може да бъде представен като двойката *G=(V,E)*, където *V* е множеството на възлите, а *E* множеството на свързаните възли, чиито елементи наричаме ребра. Самото ребро се представя като *{c,d},* което показва че има връзка между възлите c и d.

Видове графи:

Преди да продължим с нашият начин на представяне, надграждането на абстракциите и методите, нека се спрем за момент върху няколко основни видове графи спрямо:

- посоката на придвижване:

- *ориентиран граф* – от всеки възел може да се пътува само еднопосочно – т.е. не може да се върнем обратно след напущане на възела.
- *неориентиран граф* – от всеки възел може да се пътува двустранно – след като напуснем възела, може да се върнем обратно.
- *смесен граф* – дадено ребро може да бъде еднопосочно или двупосочно.

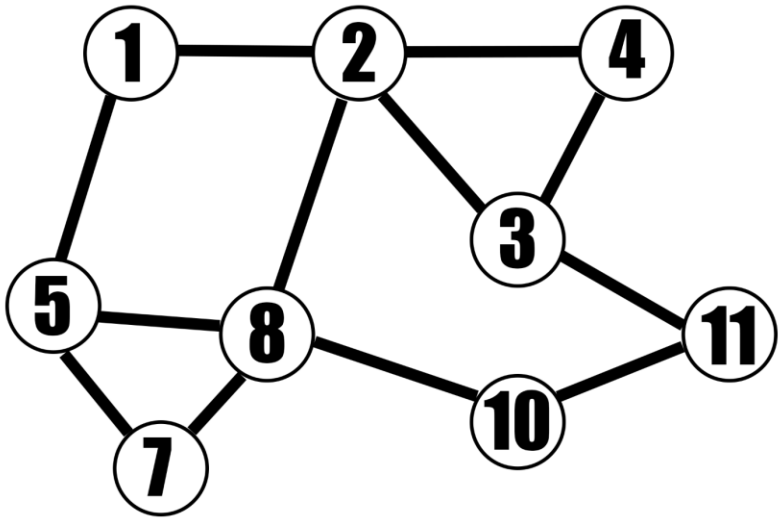
- ребрата:

- *претеглен* – ребрата между всеки два възела имат тежест на придвижването – например ако град Стара Загора е възел A, град Пловдив е възел B и град София е възел C. То от A до B имаме тегло 104 километра, но от A до C имаме тегло 230 километра.
- *непретеглен* – ребрата нямат тегло, взима се предвид броя придвижвания

Най-кратък път в непретеглен граф

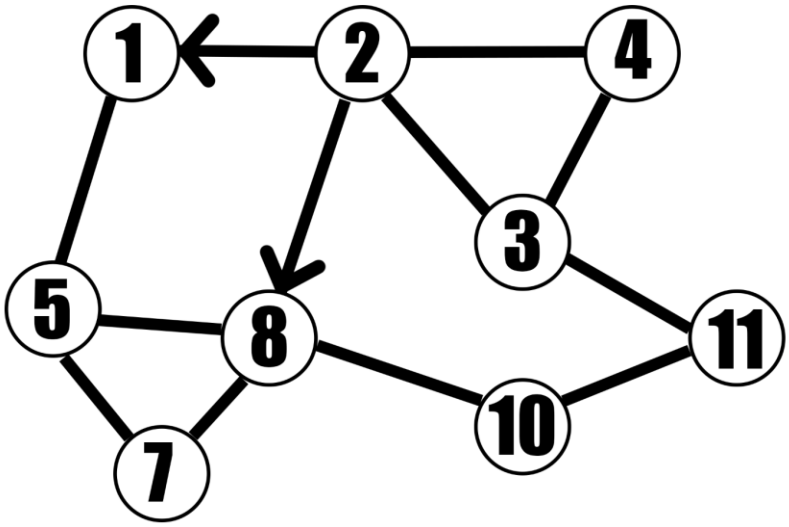
В непретегления граф, най-кратък е пътя, който преминава през най-малко възли до достигане на дестинацията. Ще го разгледаме в няколко примери.

Пример 1:



Ако започнем от възел ***1*** и искаме да достигнем възел ***3*** има много начини на преминаване, но очевидния кратък път е поредицата ***1->2->3***.

Пример 2:

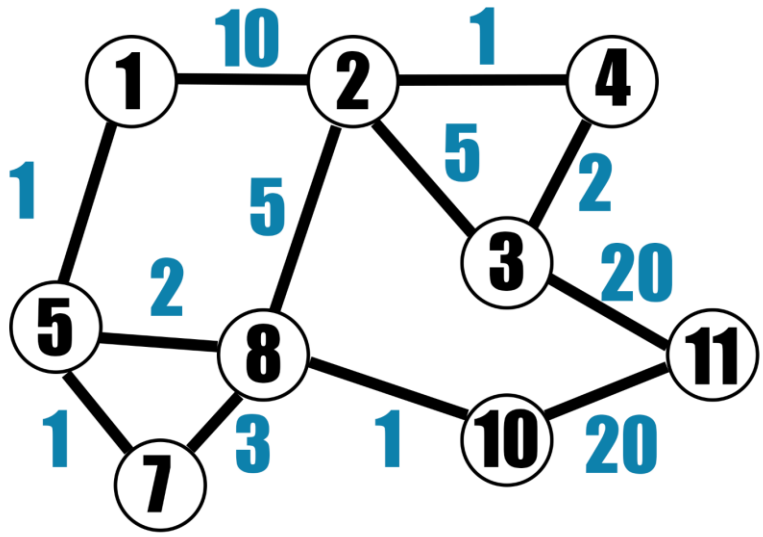


Когато променим графът дори малко, оптималният резултат може да бъде тотално различен. В случая, тъй като пътищата от ***2*** към ***8*** и ***1*** са еднопосочни, оптималното решение ще бъде ***1->5->8->10->11->3***.

Най-кратък път в претеглен граф

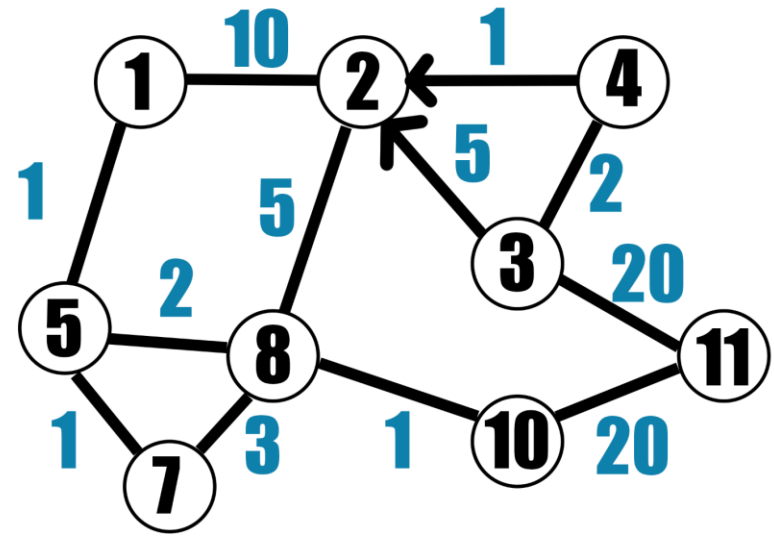
В претегления граф, най-кратък е пътя, който отнема най-малко(или най-много) точки до достигането на търсения възел.

Пример 1:



Ако отново желаем да достигнем възел ***3***, този път оптималният път ще бъде в последователността ***1->5->8->2->4->3***.

Пример 2:



Тъй като вече ***2*** няма проходимост до ***4*** и ***3***, сега най-краткият път ще бъде ***1->5->8->10->11->3***.

Представяне

При създаването на класове за графи, лесно може да попаднем в ситуация, в която да създадем множество класове за различните видове – ориентиран, неориентиран, смесен, претеглен, непретеглен и т.н. Може би дори обектна абстракция за представяне на възлите? Абсолютно ненужно.
Ще имплементираме само един единствен обект , който ще бъде само един вид – смесен. Също и достатъчно динамичен, за да поддържа претеглени и непретеглени графи – дори преобразуването на непретеглен в претеглен. Разбира се – темплейтен:

Основна структура на класа:

```
template <typename T = int>
class Graph {
private:
    std::unordered_map<T, std::unordered_map<T, double>> edges;
    bool weighted=false;
//...
}
```

Целият граф ще бъде представен чрез ***Hash Map***. Всеки ключ в map-а ще има отделен ***Hash Map***, което ще представляват възлите, с които е свързан и тежестта, за да ги достигне.
Какво печелим, когато използваме тази структура? O(1) време за „издърпване“ на възел и негов съсед/тежест.
Променливата ***weighted*** ще индикира дали сме преминали в състояние да третираме графа като претеглен или не.

Конструктор:

```
Graph(std::vector<std::pair<std::vector<T>, bool>> listOfEdges) {
        this->add_edges(listOfEdges);
    }
```

Конструкторът ще очаква вход с формат във вида:

```
        { {10, 11}, true },
        { {5,   7}, false },
        // или
        { {1, 3, 1}, true },
        { {1, 2, 1}, true },
```

С наредената двойка показваме кой възел с кого е свързан, а ако е наредена тройка, то третото число показва тежестта, която има реброто, чрез true и false показваме респективно дали реброто е двупосочно.
Функцията add_edges() е с вътрешно детерминираме като какъв вид да третираме реброто и дали е нужно да преминем към претеглен вид.

Детерминиране на вида на реброто:

```
void add_edges(std::vector<std::pair<std::vector<T>, bool>>
listOfEdges) {
    for (unsigned long long i = 0; i < listOfEdges.size(); i++) {
        this->add_pair(listOfEdges[i].first, listOfEdges[i].second);
    }
}
void add_pair(std::vector<int>& x, bool bidirected) {
    if (x.size() > 2) {
        if (bidirected) {
            this->add_undirected(x[0], x[1], x[2]);
        }
        else this->add_directed(x[0], x[1], x[2]);
    }
    else {
        if (bidirected) {
            this->add_undirected(x[0], x[1]);
        }
        else this->add_directed(x[0], x[1]);
    }
}
void add_directed(T x, T y, double value = 0){
    if (x != y) {
        make_weighted_if_neccessary(x, y, value);
        edges[x][y] = value;
    }
}
void add_undirected(T x, T y, double value = 0) {
    if (x != y) {
        make_weighted_if_neccessary(x, y, value);
        edges[x][y] = value;
        edges[y][x] = value;
    }
}
```

Чрез проверка относно подадената булева стойност, можем да определим дали да третираме реброто като еднопосочно, а чрез броя стойности като претеглено.

Алгоритъм за непретеглен най-кратък път

```
std::vector<T> nondijkstra_path(T root, T target) {
    std::unordered_map<T, std::unordered_map<T,bool>> visited;
    std::vector<T> nodes;
    std::vector<T> final_path;
    int sum = 0;
    bool set = false;
...}
```

Ще създадем масив ***nodes***(не стек или опашка), в който ще пъхаме непосетените възли, и който ще третираме като текущ път, когато завършваме едно обхождане до целта. Променливата ***sum*** ще бъде използвана за калкулиране на текущото спускане, ***set*** – дали всъщност съществува наистина път до целта, а ***visited*** – отново Hash Map третиран като булева маска за посещение с вид:
{
{node: {node:visited}, {node:visited}, …},
{node: {node:visited} …},
…
}

Ще започваме от корена като посещаваме в дълбочина всички съседи, докато не стигнем до търсеният възел. Ако за първи път стигаме до него, намереният път ще бъде запазен, ако ли не, изминатата дължина от началото ще бъде сравнена и ако е по-малка от текущата запазена, ще бъде променена.

Алгоритъмът:

```
nodes.push_back(root);
while (!nodes.empty())
{
    auto x = edges[nodes.back()].begin();
    while (x != edges[nodes.back()].end())
    {
        auto& visited_node = visited[nodes.back()][x->first];
        if (!visited_node) {
            sum++;
            visited_node = true;
            if (x->first == target)
            {
                set = true;
                nodes.push_back(target);
                if (final_path.size() == 0 || sum < final_path.size())
                {
                    std::vector<T> path;
                    for (unsigned long long i = 0;
                         i < nodes.size();
                         i++)
                    {
                        path.push_back(nodes[i]);
                    }
                    final_path = path;
                }
                break;
            }
            nodes.push_back(x->first);
            x = this->edges[nodes.back()].begin();
        }
        else
        {
            x++;
        }
}
sum--;
nodes.pop_back();
}
```

Алгоритъм за претеглен най-кратък път

```
std::vector<T> dijkstra_path(T root, T target) {
    std::unordered_map<T, bool> visited;
    std::unordered_map<T, std::pair<T, std::pair<double, bool>>> path;
    std::queue<T> nodes;

    for (auto& x : this->edges) {
        auto& key = x.first;
        path[key].first = 0; // the parent of the node if possible to
be reached
        path[key].second.second = false;
        visited[key] = false;
    }
```

Идеята тук е малко по-различна. Ще използваме отново Hash Map за посещението, но този път няма да е двуизмерна, тъй като алгоритъмът е една идея по-простичък.
Ще имаме един допълнителен Hash Map, в който ще съхраняваме най-краткия път с вид:
{
{node: { parent, {parent_cost,set} } },
{node: { parent, {parent_cost,set} } },
}
Идеята е в първата част на първата двойка да сложим родителя, от когото най-бързо се стига до текущия възел, а във първата част на втората двойка акумулираната до момента сума, втората част на втората двойка ще служи като индикатор дали сме стигали някога до този възел.

Обхождането в алгоритъмът е BFS за разлика от предходния(DFS).

Запазването на най-ниската тежест от родител за достигането до даден възел става по следният начин:
Ако например сме стигнали от възел ***1*** във възел ***2*** с акумулирана тежест ***10***, но от възел ***5*** можем да стигнем до възел ***2*** с акумулирана тежест *8*, то тогава, родителят за достигане до ***2*** ще бъде ***5***, а акумулираната тежест – *9*.

Алгоритъмът:

```
    while (!nodes.empty()) {
        // save root in a variable
        auto& root = nodes.front();

        if (!visited[root]) {
            for (auto& x : edges[root]) {

                // save each key of the roots in a variable
                auto& key = x.first;
                auto& cost = x.second;
                nodes.push(key);



                double sum = path[root].second.first + cost;

                if (!path[key].second.second) {
                    path[key].second.second = true;
                    path[key].first = root;
                    path[key].second.first = sum;
                }
                else if (path[key].second.first > sum) {
                    path[key].first = root;
                    path[key].second.first = sum;
                }
            }
        }
        visited[root] = true;
        nodes.pop();
    }
```