added: 10 Jun 2008 19:32
Edited: 9 Jul 2009 23:51

Finding the shortest paths from a given vertex to all other vertices Dijkstra algorithm for sparse graphs

Contents [hide]

Formulation of the problem, the algorithm and its proof can be found. In an article about the general algorithm of Dijkstra .

Algorithm

Recall that the complexity of Dijkstra's algorithm consists of two basic operations: Time Spent vertex with the lowest distance $d[v]$, and time of relaxation, ie time change of the $d[to]$.

At the simplest implementation, these operations require, respectively $O(n)$, and $O(1)$time. Given that the first operation is performed just $O(n)$once, and the second - $O(m)$, we obtain the asymptotic behavior of the simplest implementation of the Dijkstra's algorithm: $O(n^2 + m)$.

It is clear that this asymptotic behavior is optimal for dense graphs, ie, when $m \approx n^2$. The more sparse graph (ie, the smaller $m$compared to the maximum number of edges n^2), the less optimal becomes this estimate, and the fault of the first term. Thus, it is necessary to improve the operating times of the first type is not greatly deteriorating the run-time operations of the second type.

To do this, use a variety of auxiliary data structures. The most attractive are the **Fibonacci heap** , which allows operation of the first kind $O(\log n)$, and the second - for $O(1)$. Therefore, when using Fibonacci heaps time Dijkstra's algorithm will be $O(n \log n + m)$, which is practically the theoretical minimum for the algorithm to find the shortest path. Incidentally, this bound is optimal for algorithms based on Dijkstra's algorithm, ie Fibonacci heap is optimal from this point of view (this is about the optimality in fact based on the impossibility of the existence of such an "ideal" data structure - if it existed, it would be possible to sort in linear time, which, as is well known, in the general case impossible, however, it is interesting that there is an algorithm Thorup (Thorup), who is looking for the shortest path to the optimal linear asymptotic behavior, but it is based on a completely different idea than Dijkstra's algorithm, so no contradiction here). However, Fibonacci heaps are quite complicated to implement (and, it should be noted, have considerable constant hidden in the asymptotic behavior).

As a compromise, you can use the data structures that let you perform **both types of operations** (in fact, is the removal of the minimum and update element) for $O(\log n)$. Then the running time Dijkstra algorithm is:

$$O(n \log n + m \log n) = O(m \log n)$$

As such a data structure for programmers in C ++ convenient to take a standard container setor $\mathrm{priority\_queue}$. The first is based on a red-black tree, the second - on a binary heap. Therefore $\mathrm{priority_queue}$has a lower constant hidden in the asymptotic behavior, but it has a drawback: it does not support the delete operation element, because of what is necessary to do "workaround"

that actually leads to the replacement of the asymptotic behavior $\log n$ for $\log m$ (in terms of the asymptotic behavior is in fact really does not change anything, but the hidden constant increases).

Implementation

set

Let's start with the container set. Since the container we need to keep the top, ordered according to their values $d[]$, it is convenient to place the container in pairs: the first element of the pair - distance, and the second - the number of vertices. As a result, set the pair will be stored automatically sorted by the distance that we need.

```
const int INF = 1000000000;

int main() {
        int n;
        ... чтение n ...
        vector < vector < pair<int,int> > > g (n);
        ... чтение графа ...
        int s = ...; // стартовая вершина

        vector<int> d (n, INF),  p (n);
        d[s] = 0;
        set < pair<int,int> > q;
        q.insert (make_pair (d[s], s));
        while (!q.empty()) {
                int v = q.begin()->second;
                q.erase (q.begin());

                for (size_t j=0; j<g[v].size(); ++j) {
                        int to = g[v][j].first,
                                len = g[v][j].second;
                        if (d[v] + len < d[to]) {
                                q.erase (make_pair (d[to], to));
                                d[to] = d[v] + len;
                                p[to] = v;
                                q.insert (make_pair (d[to], to));
                        }
                }
        }
}
```

Unlike conventional Dijkstra algorithm becomes unnecessary array $u[]$. His role as the function of finding the vertex with the smallest distance, performs set. Initially, he put the starting vertex s to its distance. The main loop of the algorithm is repeated until the queue has at least one vertex. Removed from the queue with the lowest vertex distance, and then run it from the relaxation. Before performing each successful relaxation we first remove from set the old couple, and then, after the relaxation, we add back a new pair (new distance $d[to]$).

priority_queue

Fundamentally different from here set there is, except for the point that removed from the priority_queue arbitrary elements is not possible (although theoretically heap support such an operation, in the standard library is not implemented). Therefore it is necessary to make "workaround": the relaxation simply will not remove the old pair from the queue. As a result, the queue can be simultaneously several pairs for one and the same node (but with different distances). Among these pairs we are interested in only one, for which the element first is equal

to $d[v]$, and all others are bogus. Therefore it is necessary to make a slight modification: at the beginning of each iteration, when we remove from the queue the next pair, will check fictitious or not (it is enough to compare $\text{first}$and $d[v]$). It should be noted that this is an important modification: if you do not make it, it will lead to a significant deterioration of the asymptotics (up $O(nm)$).

Yet it must be remembered that priority_queuearranges elements descending rather than ascending, as usual. The easiest way to overcome this feature is not an indication of its comparison operator, but simply putting as elements of firstdistance with a minus sign. As a result, at the root of the heap will be provided with the elements of the smallest distance that we need.

```
const int INF = 1000000000;

int main() {
        int n;
        ... чтение n ...
        vector < vector < pair<int,int> > > g (n);
        ... чтение графа ...
        int s = ...; // стартовая вершина

        vector<int> d (n, INF),  p (n);
        d[s] = 0;
        priority_queue < pair<int,int> > q;
        q.push (make_pair (0, s));
        while (!q.empty()) {
                int v = q.top().second,  cur_d = -q.top().first;
                q.pop();
                if (cur_d > d[v])  continue;

                for (size_t j=0; j<g[v].size(); ++j) {
                        int to = g[v][j].first,
                                len = g[v][j].second;
                        if (d[v] + len < d[to]) {
                                d[to] = d[v] + len;
                                p[to] = v;
                                q.push (make_pair (-d[to], to));
                        }
                }
        }
}
```

As a rule, in practice version $\text{priority_queue}$is slightly faster version set.

Getting rid of the pair

You can even slightly improve performance if in containers still not keep the pair, and only the number of vertices. At the same time, of course, it is necessary to overload the comparison operator for the vertices: two vertices are necessary to compare the distances to them $d[]$.

As a result of the relaxation of the value of the distance to some vertex changes, it should be understood that "in itself" data structure is not reconstructed. Therefore, although it may seem that the delete / add elements in a container in the relaxation process is not necessary, it will lead to the destruction of data structures. Still before the relaxation should be removed from the top of the data structure to, and then insert it back relaxation - then no relations between the elements of the data structure are not violated.

And since you can delete items from set, but not out priority_queue, it turns out that this technique is only applicable to set. practice it significantly improves performance, especially when the distances are used to store large data types (such as long longor double).

16 Комментариев e-maxx Войти

Лучшее вначале Поделиться Избранный

Присоединиться к обсуждению...

Владислав • 7 месяцев назад

Вместо priority-queue можно использовать обычный массив и функции make_heap и pop_heap. Heapify, к сожалению придется реализовать самостоятельно.

• Ответить • Поделиться ›

rockets • 11 месяцев назад

Здравствуйте!
А где посмотреть, как считать граф при помощи scanf()?
(... чтение графа ...)
Спасибо.

• Ответить • Поделиться ›

Константин Ольмезов ➚ rockets • 10 месяцев назад

Про scanf() может рассказать гугл.

Процедура чтения графа зависит от того, каким видом вы представляете граф в памяти (матрица смежности, список смежности, список рёбер) и какой формат входных данных представлен в задаче.

Вот, например, код для чтения графа как списка рёбер и их весов, где граф представляется в виде списков смежности (каждый элемент - пара - вершина, вес ребра):

```
cin >> N;

vector< vector< pair<int,int> > > g(N);

cin >> a >> b >> c;

while (a>=0 && b>=0)

{

g[--a].push_back(make_pair<int,int>(--b,c));

g[b].push_back(make_pair<int,int>(a,c));

cin >> a >> b >> c;

}
```

2 • Ответить • Поделиться ›

Константин Ольмезов ➚ Константин Ольмезов • 10 месяцев назад

Добавились зачем-то закрывающие теги после }. Не обращайте внимания.

1 ^ | v • Ответить • Поделиться ›

Vanya Davidenko • 2 года назад

Зачем же в priority_queue добавлять с отрицательным весом, когда можно просто сменить компаратор (второй аргумент шаблона, по умолчанию: ::std::less) на *стандартный* ::std::greater (находится в заголовочном файле functional)?

^ | v • Ответить • Поделиться ›

e_maxx Модератор ➚ Vanya Davidenko • 2 года назад

Да, можно и так, просто с отрицательными весами чуть меньше писанины :)

^ | v • Ответить • Поделиться ›

Vanya Davidenko ➚ e_maxx • 2 года назад

priority_queue < pair<int,int>, greater<int> > q;

вместо двух непонятных минусов, про которые нужно прочитать комментарии.
Впрочем да, это холивар. Но мне больше нравится чистый код, нежели хак с минусом. Тем более, что последний выглядит как недостаток C++, а не лень программиста =)

^ | v • Ответить • Поделиться ›

Avitella ➚ Vanya Davidenko • 2 года назад

В ваше объявление очереди с приоритетами закралась ошибка. Пытался написать так, как вы, было куча ошибок. Пришлось почитать C++ Reference. Вторым параметром шаблона является контейнер. И только третьим компоратор. Если мы не объявляем компоратор, то не нужно объявлять и контейнер, потому что для этого есть параметры по умолчанию. Но если мы все-таки хотим сменить компоратор, то придется объявить и контейнер.

priority_queue< pair<int, int=""">, vector< pair<int, int=""">>, greater< pair<int, int=""">> > q;

2 ^ | v • Ответить • Поделиться ›

Avitella ➚ Avitella • 2 года назад

Что-то распарсилось плохо :(Максим, подредактируйте, пожалуйста, мое сообщение, чтобы оно отображалось нормально, если есть возможность. Или добавьте в статью. Думаю, многим будет полезно.

^ | v • Ответить • Поделиться ›

e_maxx Модератор ➚ Avitella • 2 года назад

К сожалению, такое форматирование - это баги Disqus'a (или же у него есть какая-нибудь настройка обращения с угловыми скобками, которую я не нашёл). Попробую разобраться.

^ | v • Ответить • Поделиться ›

Guest ➚ Vanya Davidenko • 10 месяцев назад

По-моему, - заставляет программу умножать число на -1, что занимает хоть и маленькое, но время. Так что лучше - действительно, поменять ФО сравнения.

^ | v • Ответить • Поделиться ›

lgand • 2 года назад

По-моему, в реализации с контейнером set вместо int to = g[v][j].first, len = g[v][j].second; правильней будет так: int len = g[v][j].first, to = g[v][j].second;

• Ответить • Поделиться ›

e_maxx Модератор ➚ lgand • 2 года назад

Всё зависит от того, что класть в g[v] :) Я предполагал, что g[v] стандартно содержит список пар (вершина_конец_ребра, длина_ребра). В таком случае верен вариант из статьи.

1 • Ответить • Поделиться ›

lgand ➚ e_maxx • 2 года назад

Но ведь в одном месте функции стоит int to = g[v][i].first, len = g[v][i].second;