Added: 10 Jun 2008 19:32
EDIT: 9 Jul 2009 23:51

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

Contents [hide]

Statement of the problem, the algorithm and its proof can be found. In the article on the general Dijkstra's algorithm .

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 while changing the value $d[to]$.

In a simple implementation of 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 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 less $m$than 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 are not strongly compromising the operating times 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 of Dijkstra's algorithm will be $O(n \log n + m)$, which is almost the theoretical minimum for the algorithm to find the shortest path. By the way, this estimate is optimal for algorithms based on Dijkstra's algorithm, ie, Fibonacci heap are optimal from this point of view (this is about the optimality actually based on the impossibility of the existence of such an "ideal" data structure - if it existed, it would be possible to perform sorting in linear time, which, as you know, in the general case impossible; however, it is interesting that there is an algorithm Torup (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 heap rather difficult to implement (and, it should be noted, have a considerable constant hidden in the asymptotic).

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

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

In the data structure such as C ++ programmers conveniently take a standard container $\rm set$or $\rm priority\_queue$. The first is based on a red-black tree, the second - on the binary heap. Therefore $\rm priority_queue$has a smaller 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 substitution in the asymptotics $\log n$for $\log m$(in terms of the asymptotic behavior of this really 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, sorted by their values $d[]$, it is convenient to put in a container couples: the first member of the pair - the distance, and the second - the number of vertices. As a result, $\mathrm{set}$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's algorithm, the array becomes unnecessary $u[]$. His role as the function of finding the vertex with the smallest distance performs set. Initially, he put the starting vertex $s$with its distance. The main loop of the algorithm is executed in the queue until there is at least one vertex. Removed from the queue vertex with the smallest distance, and then run it from the relaxation. Before performing each successful relaxation we first removed from setan old pair, and then, after relaxation, add back a new pair (with a new distance $d[to]$).

priority_queue

Fundamentally different from here setthere is, except for the point that removed from the $\mathrm{priority_queue}$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 just will not remove the old couple from the queue. As a result, the queue can be simultaneously several pairs of the same vertices (but with different distances). Among these pairs we are interested in only one for which the element $\mathrm{first}$is $d[v]$, and all the rest are fictitious. Therefore it is necessary to make a slight modification: the beginning of each iteration, as we learn from the queue the next pair, will check fictitious or not (it is enough to compare $\mathrm{first}$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 $O(nm)$).

More must be remembered that $\rm priority_queue$organizes the elements in descending order, 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 $\rm first$distance with a minus sign. As a result, at the root of the heap will be provided with the smallest elements of 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;
	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 $\rm priority_queue$is slightly faster version $\rm set$.

Getting rid of the pair

You can still slightly improve performance if the container is still not keep couples, and only the numbers of vertices. At the same time, of course, you have to reboot the comparison operator for the vertices: compare two vertices must be over distances up to them $d[]$.

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

And since you can remove items from $\rm set$, but not of $\rm priority\_queue$, it turns out that this technique is only applicable to $\rm set$. In practice, it significantly increases performance, especially when the distances are used for storing large data types (such as $\rm long\ long$or $\rm double$).