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

Formulation of the problem, the algorithm and its proof, see the article on general Dijkstra .

## Algorithm

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

At the simplest implementation, these operations require correspondingly $O(n)$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 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 allow for the 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 almost 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 heaps are optimal from this point of view (this is an optimality actually 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 is known 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 very 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 structure to allow you to **both types of operations** (in fact, it's the minimum extraction and update element) for $O(\log n)$.Then the time of Dijkstra's algorithm will be:

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

As such data structures programmers in C + + is convenient to take a standard container $\text{set}$or $\text{priority\_queue}$. The first is based on the red-black tree, the second - on the binary heap. Therefore $\text{priority_queue}$has a lower constant hidden in the asymptotic behavior, but it has the disadvantage that it does not support the deletion element, because of what has to do "workaround" which 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 $\text{set}$. Because we need to keep the container top, ordered according to their values $d[]$, it is convenient to place the container in pairs: the first element of the pair - the distance, and the second - the number of vertices. As a result, $\text{set}$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 smallest distance from the vertex, performs $\text{set}$.Initially, he put the starting vertex $s$with its distance. The main loop of the algorithm is executed in a queue until there is at least one vertex. Is retrieved from the queue with the smallest distance to the vertex, and then executed from its relaxation. Before performing each successful relaxation we first remove from $\text{set}$the old couple, and then, after the relaxation, we add back a new pair (new distance $d[to]$).

## priority_queue

Fundamentally different from here $\text{set}$is not, except for that moment, that removed from $\text{priority_queue}$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 of the same vertices (but at different distances). Among these pairs we are interested in only one, for which the element $\text{first}$is $d[v]$, and all the rest are fictitious. Therefore it is necessary to make a slight modification: the beginning of each iteration, when we learn from the next couple of turns, 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 $\text{priority_queue}$arranges elements descending and not ascending, as usual. The easiest way to overcome this feature is not an indication of its comparison operator, but simply as an element of putting $\text{first}$distance with a minus sign. As a result, at the root of the heap will be provided with the smallest distance elements 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 $\text{set}$.

## Getting rid of pair

You can even slightly improve performance if in containers not yet store the pair, and only the numbers of vertices. In this case, it is clear, it is necessary to overload the comparison operator for the vertices: compare two nodes need to distances of $d[]$.

As a result of the relaxation value of the distance to the top of any changes, it should be understood that "by itself" data structure is not reconstructed. Therefore, although it may seem that the delete / add elements to the 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 $\mathrm{to}$, 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 $\mathrm{set}$, but not from $\mathrm{priority\_queue}$, it turns out that this method is only applicable to $\mathrm{set}$. practice it considerably improves performance, especially when the distances are used to store large data types (like $\mathrm{long\ long}$or $\mathrm{double}$).