added: 10 Jun 2008 20:55
Edited: 18 Aug 2011 22:50

A minimum spanning tree. Prim's algorithm

Contents [hide]

- A minimum spanning tree. Prim's algorithm
  - Prim's algorithm
    - Description of the algorithm
    - Proof
  - Implementation
    - Trivial Pursuit: algorithms for $O(nm)$ and $O(n^2 + m \log n)$
    - Case of dense graphs: an algorithm for $O(n^2)$
    - Case of sparse graphs: an algorithm for $O(m \log n)$
    - The analogy with the Dijkstra algorithm
  - Properties of the minimum spanning tree

Given a weighted undirected graph G with n vertices and m edges. Required to find a subtree of this graph, which has linked to all its vertices, and thus has the smallest possible weight (ie, the sum of the weights of the edges). Subtree - a set of edges connecting vertices, with every vertex can reach any other by exactly one simple path.

This subtree is called a minimal spanning tree, or just a **minimum spanning tree** . It is easy to understand that any core necessarily contain $n - 1$ an edge.

In **a natural setting** , this problem is as follows: there are n cities, and for each pair is known for their expensive cost of connection (or know that they can not connect). Required to connect all cities so that you can get from any city to another, and with the cost of construction of roads would be minimal.

Prim's algorithm

This algorithm is named after the American mathematician Robert Prima (Robert Prim), which opened this algorithm in 1957, however, even in 1930, this algorithm was discovered by Czech mathematician Wojtek Jarnik (Vojtěch Jarník). In addition, Edgar Dijkstra (Edsger Dijkstra) in 1959, also invented the algorithm, regardless of them.

Description of the algorithm

Sam **algorithm** has a very simple form. Seeking the minimum spanning tree is built gradually, adding to his ribs at a time. Originally skeleton relies consisting of a single node (it can be chosen arbitrarily). Then select the minimum weight edge emanating from the vertex, and is added to the minimum spanning tree. After that, the skeleton already contains two vertices, and is now looking for an edge and adds minimal weight, with one end of one of the two selected vertices, and the other - on the contrary, in all other than these two. And so on, i.e. whenever sought the minimum weight edge, one end of which is - has already taken in the frame of the vertex and the other end - not yet taken, and this edge is added to the skeleton (if there are several edges, we can take any). This process is repeated until the frame will not yet contain all peaks (or, equivalently, $n - 1$ an edge).

As a result, the skeleton will be built, which is minimal. If the graph was originally not connected, then the skeleton is not found (the number of selected edges will be less $n - 1$).

Proof

Suppose that the graph G was connected, ie answer exists. We denote T the skeleton found by Prim's algorithm, and through S - the minimum spanning tree. Obviously, that T really is the backbone (ie, a subtree of the graph G). We show that the weight S and T the same.

Consider the first time when T it is added ribs are not included in the optimum backbone S. We denote this edge through $e$the ends of it - through $a$and b, and the set included at that time in the skeleton vertices - through V(according to the algorithm $a \in V$, $b \notin V$or vice versa). Advantageously the skeleton Svertices $a$and $b$connected in some way P; find in this way every edge g, one end of which is in V, and the other - no. Since Prim algorithm chose the rib $e$instead of the ribs g, it means that the weight of the edge $g$is greater than or equal to the weight of the edge e.

Now we remove from the Sedge g, and add an edge e. By just what to say, the weight of the core as a result could not increase (decrease it too could not because Sit was the best). Besides, S has not ceased to be a skeleton (in that connection is not broken, it is not hard to make sure we closed the way $P$to the ring, and then removed from one edge of the cycle).

Thus we have shown that it is possible to choose the optimum frame Sso that it will include an edge e. Repeating this procedure as many times, we see that we can choose the optimum frame Sso that it matches with T. Consequently, the weight of the constructed algorithm Prima T minimal, as required.

Implementation

Time of the algorithm depends essentially on the manner in which we produce the next search of suitable minimum rib edges. There may be different approaches lead to different asymptotic behavior and different implementations.

Trivial Pursuit: algorithms for $O(nm)$and$O(n^2 + m \log n)$

If we look for every time an edge easy viewing of all possible options, then asymptotically be required viewing $O(m)$edges, to find among all the valid edge with the least weight. The total amount to the asymptotic behavior of the algorithm in this case $O(nm)$, in the worst case there $O(n^3)$- too slow algorithm.

This algorithm can be improved if the view every time, not all the edges, and only one edge of each of the pre-selected vertices. For this example, you can sort the edges of each vertex in the order of increasing weights, and store a pointer to the first valid edge (recall permissible only those edges that are in the set is not selected vertices). Then, if you count these pointers whenever you add an edge to the backbone, the total asymptotic behavior of the algorithm is $O(n^2 + m)$, but first need to sort all edges for $O(m \log n)$that in the worst case (for dense graphs) gives the asymptotic behavior $O(n^2 \log n)$.

Below we consider two slightly different algorithm: for dense and sparse graphs, eventually getting noticeably better asymptotic behavior.

Case of dense graphs: an algorithm for $O(n^2)$

Approaching the matter find the lowest rib on the other side: for each is not selected will be kept minimal edge going into an already selected vertex.

Then, to the current step to make a choice of the minimum edges, you just see these minimum edges each vertex is not selected yet - be the asymptotic behavior $O(n)$.

But now when you add in the next frame edges and vertices of these pointers must be recalculated. Note that these pointers can only decrease, ie each vertex has not been reviewed yet we must either leave it without changing the pointer, or give it the weight of the edge in the newly added vertex. Therefore, this phase can also be done over $O(n)$.

So we got one of Prim's algorithm with the asymptotic behavior $O(n^2)$.

In particular, such an implementation is particularly useful for solving the so-called **problem of the Euclidean minimum spanning tree** when given $n$points on the plane, the distance between which is measured by the standard Euclidean metric, and you want to find the skeleton of a minimum weight that connects them all (and add a new vertex anywhere elsewhere is prohibited). This problem is solved by the algorithm described here in $O(n^2)$time and $O(n)$memory, which will not work to achieve the Kruskal algorithm .

The implementation of Prim's algorithm for the graph, given the adjacency matrix $g[][]$:

```
// входные данные
int n;
vector < vector<int> > g;
const int INF = 1000000000; // значение "бесконечность"

// алгоритм
vector<bool> used (n);
vector<int> min_e (n, INF), sel_e (n, -1);
min_e[0] = 0;
for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j=0; j<n; ++j)
                if (!used[j] && (v == -1 || min_e[j] < min_e[v]))
                        v = j;
        if (min_e[v] == INF) {
                cout << "No MST!";
                exit(0);
        }

        used[v] = true;
        if (sel_e[v] != -1)
                cout << v << " " << sel_e[v] << endl;

        for (int to=0; to<n; ++to)
                if (g[v][to] < min_e[to]) {
                        min_e[to] = g[v][to];
                        sel_e[to] = v;
                }
}
```

The input is the number of vertices $n$and the matrix $g[][]$size $n \times n$, which marked the weight of edges, and stand number INFif the corresponding edge is absent. The algorithm maintains three arrays: the flag $\mathrm{used}[i] = \mathrm{true}$means that the top $i$is included in the frame, the size of $\mathrm{min_e}[i]$the smallest permissible weight keeps edges from the vertex i, and the element $\mathrm{sel\_e}[i]$ contains the smallest end of the ribs (this is necessary for the output edges of the reply). The algorithm makes the $n$steps, each of which selects the vertex $v$with the smallest label $\mathrm{min\_e}$, marks it used, and then looks at all the edges of this vertex, recounting their labels.

Case of sparse graphs: an algorithm for $O(m \log n)$

In the above algorithm can be seen the standard operation of finding the minimum of the set and change the values in this set. These two operations are classic, and perform many data structures, for example, implemented in C ++ red-black tree set.

Within the meaning of the algorithm remains exactly the same, but now we can find the smallest edge in time $O(\log n)$. On the other hand, while on the Central $n$pointers now will be $O(n \log n)$ even worse than in the above algorithm.

If we consider that there will be $O(m)$calculations of the original indexes and $O(n)$searches the minimum edge, then the asymptotic behavior of the total amount $O(m \log n)$- for sparse graphs is better than both of the above algorithm, but in dense graphs, this algorithm will be slower previous.

The implementation of Prim's algorithm for the graph given adjacency lists $g[]$:

```
// входные данные
int n;
vector < vector < pair<int,int> > > g;
const int INF = 1000000000; // значение "бесконечность"

// алгоритм
vector<int> min_e (n, INF), sel_e (n, -1);
min_e[0] = 0;
set < pair<int,int> > q;
q.insert (make_pair (0, 0));
for (int i=0; i<n; ++i) {
        if (q.empty()) {
                cout << "No MST!";
                exit(0);
        }
        int v = q.begin()->second;
        q.erase (q.begin());

        if (sel_e[v] != -1)
                cout << v << " " << sel_e[v] << endl;

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

The input is the number of vertices $n$and the $n$adjacency list: $g[i]$- a list of all edges emanating from the vertex i, in the form of pairs (the second end of the edge, edge weight). The algorithm maintains two arrays: the value of $\mathrm{min_e}[i]$the smallest permissible weight keeps edges from the vertex i, and the element $\mathrm{sel_e}[i]$contains the smallest end of the ribs (this is necessary for the output edges of the reply). Additionally, a queue $q$of all the nodes in increasing order of their labels $\mathrm{min\_e}$. The algorithm makes the $n$steps, each of which selects the vertex $v$with the smallest label $\mathrm{min\_e}$(just removing it from the queue), and then looks at all the edges of this vertex, recounting their labels (when calculated from the queue, we remove the old value, and then we place a new back) .

The analogy with the Dijkstra algorithm

Two algorithms just described can be traced quite a clear analogy with Dijkstra's algorithm : it has the same structure ($n-1$phase, each of which is first selected the optimal edge is added to the response, and then recalculated values for all not yet selected vertices). Moreover, Dijkstra's algorithm also has two options for implementation: for $O(n^2)$, and $O(m \log n)$(of course we do not consider here the possibility of using complex data structures in order to achieve even lower asymptotics).

If you look at the Prim algorithm and Dijkstra's more formally, it turns out that they are all identical to each other except for **the weighting function** peaks: if Dijkstra each vertex is supported by the length of the shortest path (ie the sum of the weights of some edges), then Prim's algorithm to each vertex is attributed to only the minimum weight edge leading into the set of vertices already taken.

At the implementation level, this means that after the addition of the next vertex $v$in the set of selected vertices, when we begin to see all the edges $(v, to)$of this vertex, the algorithm Prima pointer to is updated weight of the edge (v, to), and Dijkstra - mark distance $d[to]$ is updated sum marks $d[v]$ and edge weight (v, to). Otherwise, these two algorithms can be considered identical (though they are very different and solve the problem).

Properties of the minimum spanning tree

- **Maximum** frame can also search algorithm Prima (for example, replacing all the weights of edges on the opposite: the algorithm does not require non-negativity of the weights of edges).
- The minimum spanning tree **is unique** , if the weights of all edges are distinct. Otherwise, there may be some minimum spanning tree (which one will be selected Prim's algorithm depends on the order in view of edges / vertices with the same weights / pointers)
- The minimum spanning tree is also the core, **the minimum for the product of** all edges (assuming that all weights are positive). In fact, if we replace the weights of all edges on their logarithms, it is easy to notice that in the algorithm does not change, and will be found the same ribs.
- The minimum spanning tree is the skeleton of a minimum weight **of the heaviest edges** . Most clearly this statement is clear if we consider the work of Kruskal's algorithm .
- **Criterion of minimal** core: core is minimal if and only if for every edge that does not belong skeleton, the cycle formed by this edge when added to the core, does not contain any harder edges of the rib. In fact, if for some ribs turned out that it is easier for some ribs formed loop, it is possible to obtain a lighter frame (adding this edge into the core, and removing the heaviest edge of the loop). If this condition is not fulfilled for any of the edges, then these edges do not improve the weight of the core when they are added.

---

13 Комментариев e-maxx Войти

Лучшее вначале Поделиться Избранный ★

Присоединиться к обсуждению...

her • 2 года назад

Косячные реализации машувать

6 • Ответить • Поделиться ›

Sklyack • 2 года назад

В версии с сетом в строке

if (cost < min_e[to])
кажется ещё нужно проверять, что to ещё не добавлена в MST, иначе to может быть вновь добавлена в q. Возможно подразумевается, что для уже добавенных в MST вершин условие cost<min_e[to] не="" выполниться,="" но="" кажется="" это="" неверное="" утверждение.="">

2 • Ответить • Поделиться ›

Sklyack ➦ Sklyack • 2 года назад

А, не нужно, если вершина извлечена из q, то она не в MST.

• Ответить • Поделиться ›

Sklyack ➦ Sklyack • год назад

Нет, все-таки нужно проверять. Вот тест, на котором приведенный код падает:
4 3
1 2 2
2 3 1
1 4 4
Выводит ребро (2,3) дважды, при этом 4 вершина в MST не попадает. Или вот с отрицательными ребрами:
3 2
1 2 -2
2 3 -1

• Ответить • Поделиться ›

Sklyack ➦ Sklyack • год назад

Нет, все-таки нужно проверять. Вот тест, на котором приведенный код падает:
4 3
1 2 2
2 3 1
1 4 4
Выводит ребро (2,3) дважды, при этом 4 вершина в MST не попадает. Или вот с отрицательными ребрами:
3 2
1 2 -2

• Ответить • Поделиться ›

Sklyack ➦ Sklyack • год назад

3 2
1 2 -2
2 3 -1

• Ответить • Поделиться ›

Владимир Фамилиев • 2 месяца назад

Нафиг вообще нужен vector used, на место min_e, если элемент вошел в остов просто достаточно iNF ставить. Чтобы ребро два раза не выводило, ниже главной диагонали матрицы смежности нужно поставить INF. На первом ходе легче бы было заполнить матрицу весами из первой вершины. А так спасибо за код) Это я про n^2

• Ответить • Поделиться ›

Slovinsky • год назад

В реализации за MlogN нужно добавить массив used, и в проверке на уменьшение ребра поверять на то что она не в MST;
Примерно код должен измениться так:
vector<bool> used(n + 1, false);
...

int v = q.begin()->second;
q.erase (q.begin());

used[v] = true;
...
if (cost < min_e[to] && !used[to]) ...

• Ответить • Поделиться ›

123 • 2 года назад

Если граф был изначально не связен, то остов найден не будет (количество выбранных рёбер останется меньше). "не связен"

• Ответить • Поделиться ›

Gangnam • 2 года назад

При такой реализации алгоритма MlogN решение валится на графе (матрице):

-1 2 -1 -1
2 -1 1 -1
-1 1 -1 5
-1 -1 5 -1

где matr[i][j] == -1 если между ними нет ребра, т.к. на третьей итерации i мы снова добавим уже добавленное в MST ребро между из 2 в 3 с весом 1.

• Ответить • Поделиться ›

Dauren Muratov • 2 года назад

Для заметок. При такой реализации в конце алгоритма, массив с ребрами которые нам нужны будет утерен. Например, у нас есть такой граф N=3, и 2 ребра
1 <-> 2 len=2
2 <-> 3 len=1
После алгоритма массив будет иметь 2 ребра размером единицы. Что не очень правильно.

• Ответить • Поделиться ›

Bfgnnov • 2 года назад

В алгоритме для графов на списках смежности есть косяг!!!11