EFFICIENT ALGORITHM FOR FINDING MINIMAL SPANNING TREE IN DIRECTED GRAPHS WITH INTEGER-VALUED WEIGHTS

Anna Tolkacheva
Lobachevsky State University
e-mail: AnnTolkacheva@gmail.com

Abstract

In this paper the task of finding minimal spanning tree in a weighted directed graphs is considered. Here the short survey of existed algorithms solving the given problem with various complexities is conducted. A comparatively simple algorithm that solves the given problem for graphs with integer-valued weights of arcs with the time complexity $O(m+n\log n)$ is developed as well. This result was get because of using radix sort instead of sort by comparison.

Keywords: *minimal spanning tree, directed graphs, efficient algorithm, integer-valued weights, counting sort, radix-sort, time bound.*

1. INTRODUCTION

We have a directed graph $G=(V,E)$with a marked root vertex r and the given function of cost $c(v,w)$of each edge (v,w). Let us denote a number of graph vertices as n and a number of edges as m. Every vertex is supposed to be achieved from r. The minimal spanning tree of graph G is a weighed three with the root r (a set of n-1 edges containing the ways from r to each vertex) with a minimal possible total cost of edges. J. Edmonds [8] first formulated the task of finding optimal spanning tree in a weighed directed graph and invented the polynomial algorithm for its solving. The same method was independently discovered by Y.J. Chu and T.H. Liu [6] and F. Bock [2]. The given problem had been investigating for forty years. Time estimation of algorithm solutions was being improved, algorithms for approximate solutions were being found.

2. ANALYSIS OF EXISTED SOLUTIONS

Edmonds [8] invented the polynomial algorithm for finding the minimal spanning tree in directed graph, Chu and Liu [6] and Bock [2] invented the similar algorithm independently as well. Algorithms described by Edmonds, Chu and Liu are identical; Bock's algorithm is similar to them, but it is formulated as an algorithm for matrix rather than for graphs. In papers all mentioned above algorithms of searching the minimal spanning tree are referred to as Edmonds' algorithm. There are various proofs of a given algorithm correctness: the proof of Edmonds' correctness uses the conception of linear programming, Richard Karp in his work [11] gave purely combinatorial proof of truthfulness.

By now there are several implementations of Edmonds' algorithm. Tarjan [13] realize the given algorithm with the time of $O(\min\{m\log n, n^2\})$ (here and everywhere in the paper the base-two logarithms are supposed to be used). Later Cammerini, Fratta and Maffioli [4] found and corrected the mistake in a given Tarjan's realization. Gabow, Galil and Spencer [10] developed an algorithm with the time bound in $O(n\log n+m\log\log\log_{m/n+2} n)$. Later Tarjan et.al. [9] suggested an algorithm with $O(m+n\log n)$ time bounds. At present Tarjan's algorithm of all accurate algorithms possesses the lowest time bounds for approximate estimation of complexity.

A.Sh. Nepomniaschaya [12] also represented in her work a parallel realization of Edmonds' algorithm with time bound $O(n\log n)$for the abstract mode of architecture of the type SIMD with the vertical processing of data (STAR machine) [7].

In Bagchi's work and et. al. [1] approximate solution of the problem of finding minimal spanning tree is considered. It is shown that for the case of limitation of edge number, entering to each vertex of the original graph only by k edges, having a minimal weight the tree, weight being the solution of a given optimization problem will differ not more than in $k/(k+1)$ times from the tree weight being the solution of a given optimization problem for the source graph . Here the task of data compression is also stated that often occurs in practice, where Edmond's algorithm is used for the solution of subtasks and the described approximate solution essentially accelerates the solution of the task on the whole.

In Ziegler's work [14] the version of Edmonds' algorithm for the linear time $O(m/e)$ under the reaching result in $1\text{-}e$, differing from optimal one, is given. The proof is als presented here that any realization of Edmonds' algorithm in the worst case will have $O(m+n\log n)$time bounds even in the case of limitation of a total number of entering and exiting edges for each vertex up to two.

Here I present my own implementation of Edmond's algorithm solving the given task for the graphs which edge weight is limited by a constant. Such task often occurs in practice. The version of radix sort that uses counting sort as the intermediate stable sort is used in the algorithm which allows to achieve time bounds $O(m+n\log n)$, that are similar by now to the best Tarjan's algorithm. But the given algorithm does not require such complex data structures as, for example, Fibonachi's heaps, that Tarjan uses in his algorithm, and it is, accordingly, easier for realization in practice.

3. THE DESCRIPTION OF THE ALGORITHM

3.1. The Description of Edmonds' Algorithm

First let's give a short description of a source Edmond's algorithm. More detailed description of the algorithm may be found in the original Edmonds' paper [8], or in Zeigler's paper [14] where he recapitulated the algorithm using his own notation.

The algorithm consists of two parts. In the first stage a *set* of edges containing a minimal spanning tree and extra edges is found. The stage begins with the fact that no edge is selected and a *set* of selected edges (originally empty) that determines the forest is always supported. While performing the stage, for every vertex v, if there are no edge entering to v in the *set*, we find a minimal weight arc (v^*,v) and add it to the *set*. If a cycle is generated in this way we contract it into one new vertex. All vertices of the cycle are not considered any more in the first stage but a new vertex is generated instead of them. All graph edges, that led to the vertex of a contracted cycle, lead to it now. In addition, the edge cost entering to a new vertex is counted: from each graph edge cost, earlier entering to the cycle vertex, the edge cost entering to it in the cycle (belonging to the *set*) is subtracted. The loops and multiple edges are removed except the edge of minimal cost. At present there are no edges in the *set* leading into it. At the end of the stage all vertices in a graph will be contracted in one vertex.

In the second stage of the algorithm we expand the cycles formed during the first stage, in the inverse order to their contraction (successively passing through the rest of the edges from the *set* in the inverse order to their adding to the *set*) throwing up one edge from each cycle. It is one that leads in the cycle to the vertex of the source graph to which a considered current edge of the set leads. Finally the tree remains. The second stage of the algorithm takes $O(m)$ time.

3.2. The Description of Our Implementation

Here the algorithm, having a time bound $O(m+n\log n)$, that is equal to time bounds of the algorithm, suggested by Tarjan [9], will be given. But a given algorithm is easier for realization and uses the less number of complex structures. Here the weights of graph edges are integer-value and supposed to be limited by m. But the algorithm is easily modified even for larger weight: if k is a constant, time bounds will be $O(km+n\log n)$for the weighs limited by m^k.

For the correct work of the algorithm it is supposed that the graph was originally given as a set of vertices with the list of entering edges and their weighs for each vertex are integer-value. The weights of the edges are limited by a constant.

For a given realization we will slightly modify the Edmonds' algorithm given above without changing its meaning. We achieve the required time bounds due to the initial sorting all graph edges by means of the version of radix sort that uses counting sort as the intermediate stable sort [5]. In the first stage of the algorithm we`ll be constantly building the subgraph of this graph G, in which only one edge (which cost is minimal) enters into each vertex, and contract the vertices, that belong to the cycles in the received subgraph, to form a new vertex, thus forming a new graph G, until the cycles will be formed. In the second stage without changing Edmonds` algorithm we`ll expand contracted cycles in the inverse order to their contraction, forming the minimal spanning tree from the edges entering into the contracted cycle.

The first stage of the algorithm consist of some phases. At the beginning of each i-th phase we have the graph $G^i=(V^i,E^i)$ and the list of vertices $V^{*i}\in V^i$. In the first phase $G^1=G(V,E)$, $V^1=V$ and $V^{*1}=\{r\}$. We find for every vertex $v\in\{V^i/V^{*i}\}$ edge$(v^*,v)\in E^i$ with minimal possible weight ($v^*\in V^i$). It is obvious that the received edges (let's denote them as E^{*i}) form the trees with the roots from V^{*i} and some k independent cycles C_j, $j\in\{1...k\}$. At the end of the phase we'll get the graph G^{i+1} formed by contracting each received cycle C_j into a new vertex v_j. All the vertices which belong to the trees make new V^{*i+1}. $V^{i+1}=V^{*i+1}+\{v_j, j\in\{1...k\}\}$. The edges coming out of v_j inherit the cost of corresponding to them edges, coming out of the vertices of cycle C_j. The cost of every edge entering to v_j is equal to the cost of corresponding to it edge, entering into the vertex $w\in C_j$ minus the cost of the edge (w^*,w), where w^*, $w\in C_j$ and $(w^*,w)\in E^{*i}$.

The stage is completed when no other cycle is formed at the current phase ($i=t$). As the number of vertices of a new graph is decreasing at each phase, the fist stage is finite. As each cycle consists at least of two vertices and at each phase the cycles are contracted into one vertex, at each phase the number of vertices, for which we search the minimal edges , are removed at least two times. At the end of the stage all cycles will be contracted into one vertex, hence $t\leq\log n$.

At the first stage $G^*=\in\bigcup_{i=1}^{t} G^{*i}$ is just the set of edges containing the minimal spanning tree similar to the source Edmonds' algorithm.

The second stage consists of the successive forming the minimal spanning tree from the received edges E^* by exploring the cycles in an inverse order to their contraction. Starting with $i=k$ we include each edge from E^{*i} to the spanning tree, while removing the edges entering into the same vertex from all E^{*j} for $j<i$. Further we proceed to the next set of edges

E^{*i} and repeat the same procedure including the edges remained in E^{*i} to the spanning tree until i becomes equal to 1.

Let's examine the first stage in details. For each vertex of the source graph we present the set of edges entering into it in the form of a one-way list and call it *passive list*. For correct operation of a given algorithm the edges in every list must be sorted according to the increase of weights (the beginning of the list is a minimal edge entering into the vertex and each edge refers to the following edge in weight). For this purpose let's perform sorting all graph edges by radix sort that uses counting sort as the intermediate stable sort [5]. Then by a single pass the sorted edges from the least to the largest let's distribute them through *passive lists* of vertices (obviously, this distribution will take the time $O(m)$). By using counting sort as the intermediate stable sort radix sort is carried out in time $\Theta((b/r)\ (1+2^r))$ for l b-bits numbers and natural number r ($r<b$). If edge weights are considered to be dependent of the number of edges in the graph and are restricted by m^k (k is a constant), then $b=\log(m^k)=k\log m$. Let's take $r=\log m$ then the time of list sorting from m edges will be rewritten in the form of $\Theta((b/r)\ (1+2^r))=\Theta((k\log m/\log m)\ (m+2^{\log m}))=\Theta(\mathrm{k}(m+\mathrm{m}))\leq O(m)$. Hence the total time of construction of sorted entering list for graph vertices is $O(m)$.

For presenting the edges in the current contracted graph G^i we'll store their modified weights by means of additional data structure *tree with path compression* [15]. The given structure was created for storing and fast operations with the collection of disjoint sets in which its cost corresponds to each element (integer value in our case). Originally each set is singleton. In our algorithm the graph vertices (V) will be elements, and the sets are the cycle vertices contracted into a new vertex of the next contracted graph G^i (the name of the set will be the name of the formed vertex). The cost of the source vertices is the sum of weight changes of each edge entering to it. Let's describe the operations that the given structure supports in brief:

- *Find*(a). The operation retrieves the name of the set containing element a.
- *Find the cost*(a). The operation retrieves the current cost of element a.
- *Change the Cost*(w,A). The operation adds w to the weight of all elements in the set A. It takes $O(1)$time.
- *Unite*($A1$,$A2$,$A3$). The operation will unite sets $A1$ and $A2$ into set $A3$ removing source sets $A1$ and $A2$. It takes $O(1)$ time.

Thus, if (v,w) is a source edge and c is the function of weight, the corresponding current edge at a new stage is (*Find*(v), *Find*(w)) and its weight will be equal to *Find the Cost*(w) plus $c(v,w)$.

During the cycle contraction consisting of the vertices v_1, v_2,..., v_k we control the structure of compressed tree as follows. For $i=\{1...k\}$ let (x_i,y_i) be a source edge corresponding to a current edge ($v_{i\text{-}1 \bmod (k+1)}$,$vi$). To renew the cost of the edge we perform *Change the Cost*($-c(x_i,y_j)$-*Find the cost*(y_j), y_j) for each y_j, j={1,k}. Then operation *Unite* is performed k times and sets with names v_1, v_2,..., v_k are merged into single set.

While carrying out the algorithm the operations *Change the Cost* and *Find the Cost* are performed $O(n)$ times. General performance of k operations of the type *Find* and *Find the Cost* all together take the time $O((k+n)/\alpha(k+n,n))$. (For $k \geq n\log n$, $\alpha(k,n)=O(1)$ [15]). The amount of such operations while performing the algorithm is $O(n\log n+m)$ hence their performance will take the same $O(n\log n+m)$. The total time of performing all operatons with the structure *tree with path compression* will be $O(n\log n+m)$ in our algorithm. We also represent each current edge by corresponding current edge of graph G as we may obtain the current edge corresponding to any source edge for $O(1)$. For every i-th we'll store i-th list of the *using edges* E^{*i} in it (at the beginning of each phase it is empty) and the list of pointers to those vertices for which minimal edges (V^i/V^{*i}) are vertices formed by contracting the cycles of the previous phase, and for the first phase all the phases of the source graph except the root)must be found. For every source vertex we will store *entering list* of edges chosen from any phase of the first stage (edges themselves are not stored in the list but pointers to them in *using edges* , thus it is possible to perform the second stage faster).

For the fast definition of cycles we'll mark the vertices by the number of the current phase while performing the first stage. At the beginning of algorithm work in the first phase all vertices except the root r are not marked, but the root has the label 0. At the next phase the vertices, formed anew by contracting, won't be marked initially. The vertex will store two indicators as well: first to cyclically connected *list of subvertices* and second to the *next vertex* for it's own cyclically connected list. Contracting one will point to it's original vertices, and the source vertex initially refers to itself.

Notice that each vertex refers to the *passive list of edges*, entering to it where the first edge of the source vertices is always minimal but there will be only one edge(minimal) in the newly formed ones as we'll see further.

Let's consider the i-th phase of the first stage:

1. While there are vertices in the vertex list of the phase V^i we perform the following:

    a. We take the first vertex v from the list. If it is marked we remove it from the list and perform *step* 1*a* otherwise continue.

    b. We mark the vertex by the number of the current stage.

c. We take the first edge from the *passive list* of the vertex v (let it be edge(v',v''), where $v''=v$ or v'' is contracted into v), remove it from the *passive list* of the vertex v'' and put to the *using edges* list and add pointer to this edge to the *entering list*. Then we perform operation $v^*=Find(v')$ *next vertex* pointer of v point should point to v^*, and cyclically connected *list of subvertices* of v and v^* are merged in $O(1)$. If v^* is not marked then we take $v=v^*$ and go to *step* 1*b*. If it is marked by a label that is less than the number of the current stage (it means the cycle was not formed) we go back to the *step* 1. If it is marked by a label of equal current stage (the cycle was formed) we go to *step* 1*c*.

d. According to the pointers to the *next vertex* we go round all vertices of the formed cycle, contracting them into a newly created vertex w, by performing the operation $Unite(v_i,v_{i+1},w)$ for each vertex v_i and following vertex v_{i+1} according to its reference. Original vertices which represent the cycle vertices (v_i) are all already connected with each other in *step* 1*c*. We add the reference to any of them into a new contracted vertex. We go round the cycle list of source vertices once representing now w in order to find the minimal edge from all entering into its vertices edges. But such edge doesn't have to belong to a cycle by its initial vertex, therefore for every vertex v, of this cycle we look at the first edge of its *passive list* — (v',v) performing the operation $Find(v')$. If $Find(v')=w$, we remove the first edge from the *passive list* and look at the next edge. If it is not the case we compare it with the current minimal edge for w (if there is no such edge we make (v',v) minimal). Notice that while performing the first stage (in all its phases in totality), removal of edges may not be more than m.

2. If in the *vertex list* of the phase $i+1$ there is at least one vertex we pass to the next phase ($i=i+1$).

Thus while performing the stage we take each contracted vertex of this stage at least 3 time. If all stages are summed up according to a number of vertices then: $|V|=|V^1|+|V^2|+\ldots+|V^t|\leq n+n/2+n/4+\ldots+n/2t\leq$ $\leq n+n/2+\mathrm{n}/4+\ldots+n/(2\log\ n)=n(((1/2)\log\ n-1)/(1/2-1))=2n(1\log\ n)\leq 2n$. We also address to the source vertices in every stage to operate with their *passive lists* not more than n times, and not more than $n\log n$ times at all phases respectively. Then there are not more $n\log n+m$ operations with the vertices of passive lists. The resulting operations at the first stage are $O(2n+n\log n+m)=O(n\log n+m)$. Hence, the total labor capacity in the first stage is $O(n\log n+m)$.

At the second stage we successively include all edges E^{*i} (from $i=k$ till 1), formed in the first stage, in spanning tree. Doing so, for each edge (v',v'') included into the spanning tree we clean the *entering list* of vertex v' references in order to remove each edge, to which the pointer from this *list* leads, from the sets E^{*j}, $j<i$. Thus, for each vertex in the spanning tree (except the root), there will be equally one entering to it edge. And the received spanning tree will be minimal (corresponding to Edmonds' algorithm completely).

The time of performance of stage 2 is equal $O(2n)$. Thus, the time of the operation of the total algorithm is $O(n\log n+m)$. The required memory for the algorithm is $O(m)$.

4. ACTUALITY OF THE TASK AND ITS PRACTICAL USAGE

The task of searching the minimal spanning tree of the given weighed directed graph is commonly formulated and the algorithms solving it can be applied in a rather broad range of tasks for different spheres of scientific research. As all the tasks formulated on graphs, on the one hand this task is of practical importance on the other hand it may be an applied task for any field of science. Such task occurs as a subtask in biology, networks theory, chemistry. In a particular practical task the optimal spanning tree may be calculated many times for the solution of super tasks therefore it is important for calculating it algorithms to have high fast action. In some cases accuracy is even neglected and the solution with a given approach is approximated for increasing fast action.

Initially the origin of this task was connected with applications where the subset of minimal cost providing communication of the given source with all the rest of net vertices is required to be found. At present this kind of the task occurs often in bioinformatics. In Jasmin Bogojeska's and et. al. paper [3] the task of searching the minimal tree is used as a subtask for finding the structure of mixture of mutation trees. In their work [1] Amitabha Bagchi et. al. show the application of the given task for compressing the data, namely for the tasks of delta-compression that constantly occur in internet applications. With the increase of volumes of processed data faster and faster algorithms are required, therefore approximate solutions of this task, less calculative complex, are becoming more and more actual.

5. REMARKS

In Zeigler's work [14] the proof is given that the bound $O(n\log n+m)$ is strict, that is, the given task may not be solved for the time less than $\theta(n\log n+m)$. But in this proof the key idea is estimation of sorting by comparison, and by using other sorting in application to the solution of this task the given bound isn't probably lower. Hence, the question about the existence of faster accurate algorithm solving the given task, using other methods of sorting (for example, radix-sort) is open.

REFERENCES

1. **Bagchi, A., Bhargava, A., Suel, T.**: Approximate Maximum Weight Branchings. J. Information Processing Letters v.99 n.2, 54--58 (July 2006).
2. **Bock, F.**: An Algorithm to Construct a Minimum Directed Spanning Tree in a Directed Network: Developments in Operatins Research, Gord and Breach, 29-44 (1971).
3. **Bogojeska, J., Lengauer, T., Rahnenfhrer, J.**: Stability analysis of mixtures of mutagenetic trees. J. BMC Bioinformatics v. 9, 165-181 (March 2008).
4. **Camerini, P. M., Fratta, L., Maffioli, F.**: A Note on Finding Optimum Branchings: Networks 9, 309-312 (1979).
5. **Cormen., T., H.,, Leiserson, Ch., E., Rivest, R., L., Stein, Cl.**: Introduction to Algorithms, Third Edition - 2009. pp. 191-200. The MIT Press, Cambridge, Massachusetts London, England (2009).
6. **Chu, Y.J., Liu, T.H.**: On the Shortest Arborescence of a Directed Graph, Sci. Sinica 14, 1396-1400 (1965)
7. **Fet, Y. I.**: Vertical Processing Systems: A Survey, IEEE Micro v.15 n.1, 65 75 (February 1995)
8. **Edmonds, J.**: Optimum branchings, J.Res.Nct. Bur. Standards 71B, 233-240 (1967)
9. **GaBow, H. N., Galil, Z., Spencer, T., Tarjan, R. E.**: Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs: Combinatorica 6(2), 109-122 (1986)
10. **GaBow, H. N., Galil, Z., Spencer, T.**: Efficient implementatin of graph Algorithms using Contraction, J.Assoc. Comput. Mach., submitted
11. **Karp, R. M.**: A Simple Derivation of Edmonds' Algorithm for Optimum Branchings: Networks, v.1, 265-272 (1971)
12. **Nepomniaschaya, A. Sh.**: Representation of Edmonds' Algorithm for Finding Optimum Graph Branching on Associative Parallel Processors: Programming and Computing Software v. 27, 200-206 (July-August 2001)
13. **Tarjan, R. E.**: Finding Optimum Branchings: Networks 7, 25-35 (1977)
14. **Ziegler, V.**: Approximating Optimum Branchings in Linear Time: Information Processing Letters archive v. 109. Issue 3, 175-178 (January 2009)
15. **Tarjan, R. E.**: Efficiency a Good but not Linear Set Union Algorithm. J. Assoc. Comput. Math. 22, 215-225 (1975)