AN Ø(n·Ilog^{2}I) MAXIMUM-FLOW ALGORITHM

by

Yossi Shiloach

STAN-CS-78-7Ø2
December 1978

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

An $O(n \cdot I \log^2 I)$ Maximum-Flow Algorithm

Yossi Shiloach [*]

Computer Science Department
Stanford University
Stanford, California 94305

December 1978

Abstract. We present in this paper a new algorithm to find a maximum flow in a flow-network which has n vertices and m edges in time of $O(n \cdot I \log^2 I)$, where $I = m+n$ is the imput size (up to a constant factor). This result improves the previous upper bound of Z. Galil [G] which was $O(I^{7/3})$ in the worst case.

Keywords: Binomial queues, Dinic's algorithm, flow networks, maximal flow, path sections.

[*] This research was supported by a Chaim Weizmann Postdoctoral Fellowship and by National Science Foundation grant MCS 75-22870.

1. Introduction.

A flow network is a quadruple (G,s,t,c) where

(i) $G = (V,E)$ is a directed graph.

(ii) s and t are two distinguished vertices, the source and the terminal, respectively.

(iii) $c\colon E \to R^+$ is the capacity function (R^+ denotes the set of all positive real numbers). Henceforth n and m will denote $|V|$ and $|E|$ respectively and I will stand for $n+m$. The notation $\underline{(u,\quad}$ will represent a directed edge from u to v.

A function $f\colon E \to R^+$ is a flow if it satisfies:

(a) The capacity rule:

$$f(e) \leq c(e) \qquad \forall e \in E .$$

(b) The conservation rule:

$$\mathrm{IN}(f,v) = \mathrm{OUT}(f,v) \qquad \forall v \in V-\{s,t\}$$

where

$$\mathrm{IN}(f,v) = \sum_{\substack{(u,v) \in E \\ \text{fixed } v}} f(u,v) = \text{total flow entering } v$$

and

$$\mathrm{OuT}(f,v) = \sum_{\substack{(v,w) \in E \\ \text{fixed } v}} f(v,w) = \text{total flow emanating from } v.$$

The total flow value $|f|$ is defined by

$$|f| = \mathrm{OUT}(f,s) - \mathrm{IN}(f,s) .$$

A flow f is a maximum flow if $|f| \geq |f'|$ for any other flow f'.

The maximum flow problem was first introduced and solved by Ford and Nkerson [FF]. Since then, better and better solutions have been found. The history and state of the art of this problem are given in a very detailed and tabulated form in [G]. Thus, we allow ourselves to proceed without paying the proper credit to all those researchers who contributed so much to the study of this problem,

2. Dinic's Reduction.

E. A. Dinic showed in his paper, [D], that the maximum flow problem can be solved by solving another much simpler problem at most n times.

The simplified problem can be stated, using the following three definitions.

1. A layered network is a network whose vertices are partitioned into disjoint sets $V_0, V_1, \ldots, V_k$ where $V_0 = \{s\}$, $V_k = \{t\}$ and if $(u,v) \in E$ then $u \in V_i$ and $v \in V_{i+1}$ for some $0 \leq i \leq k-l$,

2. Given a flow f in a network, we say that an edge $e \in E$ is saturated if $f(e) = c(e)$. A saturated edge will also be called a bottleneck. The quantity $c(e) - f(e)$ is the residual capacity of e.

3. A flow f in a given network is maximal if every directed path from s to t contains a saturated edge. Obviously, a maximum flow is also maximal but the converse is not necessarily true,

Dinic's restricted problem is to find a maximal flow in a layered network which has n vertices and at most m edges.

Dinic himself solved the restricted problem in $O(n \cdot I)$ time and all the improvements which followed his solution were, in fact, improvements of the restricted problem's time bound. Our algorithm is no exception. We show that Dinic's solution for the restricted problem can be implemented in $O(I \log^2 I)$ time by using more efficient data structures. The evaluation of the complexity is also more involved, but fortunately it does not affect the algorithm. From now on, the "problem" and a "solution" will always refer to the restricted problem and its solutions.

3. The Underlying Algorithm.

In this section we'll describe our algorithm in a level which will not involve any data-structure details. We want the reader to have a clear picture of the underlying algorithm before we get into the data-structure level. Moreover, we'll make it clear in this section, what exactly are the features which our data-structure would have to support.

Since Dinic's algorithm is the framework of ours, we'll describe it briefly first.

Dinic uses a depth first search to find a flow-augmenting path (f.a. pathj and then pushes as much flow as possible through this path, deletes the bottlenecks, updates the residual capacities along the path and then starts from s a new search for another f.a. path.

The main drawback in Dinic's algorithm is that it does not store any information which is not relevant to the currently growing f.a. path. Path-sections that have already been traversed during searches for previous f.a. paths, are completely ignored. Our algorithm, though basically following the lines of Dinic's algorithm, stores such path-sections and makes use of them as soon as they are reencountered.

Instead of describing the underlying algorithm precisely, we'll give the reader the feeling of what's going on by illustrating a typical example of generating the first three f.a. paths. The various steps will be accompanied by a sequence of figures (Fig, 3.1-3.11), showing what are the path-sections that are stored after each step, Each figure corresponds to the step with the same number.

Step 1. We find the first f.a. path from s to t ,

Step 2. Two bottlenecks (u,v) and (w,x) are found and deleted. Three path sections are stored, the (s,u) section, the (v, x) section, and the (y,t) section.

Step 3. The search resumes from u . In general the search will resume from the tail of the lowest bottleneck. The section (s,u) will now be "growing" into a new f.a. path.

Step 4. We encounter an old path section, namely the (v,x) section at w.

Step 5. We cut the (v,x) section into two sections (v,w) and. (w,x) and then paste the (w,x) section with the (s,w) section.

Step 6. We continue the search from x .

Step 7. We encounter t . A new f.a. path has just been found.

Step 8. A bottleneck (a,b) is found and deleted. Two new path sections (s,a) and (b,t) are formed.

Step 9. We resume the search from a and encounter the path section (v, w) at z .

Step 10. We cut the (v,w) section into (v,z) and (z,w) sections and paste the (z,w) section with the (s,z) section, forming an (s,w) section.

Step 11. Since we are now in w , we have just encountered the old (b,t) section. Thus, we cut the (b,t) section into (b, w) and. (w,t) sections and paste the (w,t) section with the (s,w) section, forming a new f.a. path.

In order to complete the picture we have to add a few words, When we use the word "search" we mean a depth first search. As in the original algorithm of Dinic, this means that when we reach a dead end we backtrack to its predecessor and delete it together with the edges incident with it, The algorithm terminates when s becomes a dead end. At each stage of the algorithm, an edge can be in exactly one of the following states:

1. Never encountered.
2. Belongs to a (unique) path-section.
3. Deleted (because of being either saturated or incident with a dead end.

The crucial rule in our algorithm is that when we encounter a path section, we annex its upper portion to our growing f.a. path.

The rule implies the following lemma.

Lemma 3.1. At each stage of the algorithm, a given vertex can have at most one out-going edge in State 2.

Following the lines of the underlying algorithm, one can verify that the data structure should support the following operations.

1. Performing a depth first search.
2. Finding and deleting bottlenecks.
3. Updating residual capacities.
4. Inserting an edge to a path-section. (The growing f.a. paths is stored and regarded as a path-section.)
5. Cutting and pasting path sections.

Getting more into details, we find out that we also have to be able to:

6. Know that we've encountered a path section, when we encounter one.
7. Know from where to resume the search after pasting two path sections.
8. Compute the final residual capacity of each edge. (The flow through each edge is obtained by subtracting the residual capacity from the original one.)

4. Data Structure.

4.1 Sequential Binomial Queues.

Data structures which support depth first search have been discussed in many papers. Therefore we'll concentrate on the additional data structure which will enable us to store and manipulate with path sections. This data structure is essentially a binomial queue. It has been described and studied in detail in [B]. A binomial queue is a forest of binomial trees (b-trees and b-queues in short). In a b-tree we store sets of size which is a perfect power of 2 , A b-queue which is a forest of such trees enables us to store sets of any given size. These-sets must be well ordered and in our case they will be sets of numbers which represent edge capacities. A binomial tree which stores the set {71,52,63,11,7,4,22,20} is shown in Figure 4.1. Basically all the values are stored in the external nodes and each other node contains the minimal value of its two off-springs. The root contains the minimal value of the entire set.

In our b-queues, we'll store the capacities of edges of path sections. Being a path, such a group of edges has a sequential structure and therefore we would like that our b-trees will represent sequences rather than sets. Figure 4.2 shows a b-tree which represents the sequence 71 ,52 ,63 , 11 ,7 , 4 ,22 ,20. In general, the external nodes of these trees will have a left to right order which will be imposed by the sequence. These trees will be denoted as s-b-trees (sequential binomial trees) and the corresponding queues will be s-b-queues. Another modification should still be made. Instead of storing the right capacity values at each node, we will store the riaht value of the root only. Each other node will store the difference between his value and his father's value, Thus, the tree of Figure 4.2 will be modified to that of Figure 4.3. This modification will enable us to update residual capacities fast.

In the coming discussion we would like to distinguish between *vertex-layers* (*v-layers*) and *edge-layers* (*e-layers*). The vertex s composes the 0-th v-layer, while the edges which emanate from it, form the 0 -th e-layer, In general, the heads of the edges of the i-th e-layer will form the i+1 -st v-layer and the edges which emanate from them will form the i+1 -s-t e-layer. We shall also assume for the sake of simplicity that the number of e-layers is a perfect power of 2 , say 2^k . (If this number is between 2^{k-1} and 2^k we can add a path which starts from t and has the appropriate length. The edges along this path will have infinite capacities and t will be moved to its end.)

We have described in detail how one s-b-tree looks like. We'll now describe how a path section is represented by an s-b-queue. The following definition is required. Given a full binary tree T and a set S of external nodes of T , we say that $\mathcal{F}$ is the *forest determined by S* if it is the (unique) forest of maximal full sub-trees of T such that its external nodes set is S . In Figure 4.4 we show a full binary tree T which has 8 external vertices and the forests which are determined by $\{1,2,3,4\}$, $\{5,6\}$, and $\{0,1,2,3,4\}$.

Let's recall now that there are 2^k e-layers in our network, and let T be a full finary tree with 2^k external nodes which are numbered from left to right by $0,1,\ldots,2^k-1$. The following rule specifies exactly the structure of an s-b-queue which represents a given path section.

The Frame Rule. A path section that extends from the i-th e-level to the j-th e-level will be represented by an s-b-queue which is isomorphic to the forest determined by the set $\{i,...,j\}$ of external nodes in T .

The rule is called "the frame rule" since T is used as an underlying frame for all the queues which represent path sections.

In Figure 4.5 we show a path section together with his representing s-b-queue and with all the features that we have already mentioned. We assume that the total number of e-layers is 8 . Note that the left-to-right order among the trees in the queue is significant.

4.2 Path Sections Algorithms.

In this subsection we'll present the algorithms for finding and deleting bottlenecks, updating residual capacities, inserting an edge to a growing f.a. path and cutting and pasting path sections. (These are algorithms # 2 -5 in the list at the end of Section 3.) Each of these algorithms will take at most logarithmic time.

The other operations that should be supported by our data structure (# 6,7,8 in the list) will be described in the next subsection where an auxiliary data structure will be introduced and a more detailed discussion about storage schemes, pointers and space will be made. This discussion will also make the description of algorithms # 2 -5 more complete.

Finding and Deleting Bottlenecks.

These operations are performed after a whole f.a. path has been found. Such a path is always represented by an s-b-queue which is a single s-b-tree (since the number of layers is 2^k). The residual capacity of the bottlenecks is exactly the value which is stored in the root of this s-b-tree. "Pushing" that amount of flow through the path is tantamount to reducing the root's value to zero. Then we delete the root and obtain two s-b-trees of equal size. At least one of them will have a root with value zero. The algorithm continues on such a tree in the same way. When we get to the level of the external nodes, deleting a node also means that we delete the corresponding edge from the graph. This algorithm might take more than logarithmic time if more than one edge is deleted. However, it is easy to verify that it does not take more than logarithmic time per a deleted edge. In Figure 4.6 we show an f.a. path of length 8 and all the stages of deleting its bottlenecks.

Updating Residual Capacities.

Due to our way of storing differences between residual capacities, rather than capacities themselves, we don't have to do anything, it is done "automatically".

Inserting an Edge to a Growing f.a. Path.

Let P be a growing f.a. path of length ℓ and let Q(P) be its corresponding queue. The binary representation of the number ℓ contains all the information about the structure of the underlying forest of Q . More precisely, the i-th digit from the right is 1 iff there is an s-b-tree of height i-l in the forest. For example, a path of length 6 is represented by a forest that contains one tree of height 1 and one tree of height 2 . (Note that this property holds only for path sections that start at s .)

When we insert an edge to P , we always insert it from the "right" (assuming that s is on the left and t is on the right) and by doing that we increase the length to $\ell+1$. As far as the forest structure of the new queue is concerned, we just-make a binary addition of ℓ and 1, (see also [B], pp. 21-27). Obviously, since the nodes of the s-b-trees contain some numerical values, we have to do a little bit more. When we add a bit of 1 to a bit of 0 we don't have to do anything. However, adding two bits of 1 means the following:

1. Take the corresponding two trees (which are of equal size) and connect their roots to a new single root, forming one tree which is one level higher.
2. Put the minimal value of the two old roots in the new one, zero in the old root that contributed the minimal value and the difference between the two values in the other old root. It is easy to see that insertion is logarithmic.

In Figure 4.7 we show how an edge e is inserted to a path P of length 3 .

Cutting and Pasting Path Sections.

Given a path section P we want to cut it in a given vertex, and make two path sections, say P_1 and P_2 , out of it. If the cut point turns out to be exactly between two trees of Q(P) then all the trees to the left of the cut point form $Q(P_1)$ and the others form $Q(P_2)$.

If the cut-point is inside a tree T of Q(P) then all the trees to the left of T belong to $Q(P_1)$ and those on its right hand side belong to $Q(P_2)$. We now cut T in the following way:

1. We delete the root of T and add its value to both sons which now become the roots of the resulting trees T_1 and T_2. Let's assume that T_1 is to the left of T_2.
2. If the cut point is exactly between T_1 and T_2, then T_1 is added to $Q(P_1)$ and T_2 is added to $Q(P_2)$ and we are done. If the cut point is inside T_1 (T_2) we add T_2 (T_1) to $Q(P_2)$ $(Q(P_1))$ and apply the same procedure to cut T_1 (T_2).

Cutting is obviously logarithmic.

Pasting is a little bit more involved to describe but also belongs to the set of "do it yourself" algorithms.

Thus, instead of describing it formally, we start with an example. Let's assume that the total number of layers is 32 and we have to paste a path section P_1 which extends from the third to the 18-th layer with another path section P_2, extending from the 19-th layer to the 25-th. The underlying structure of $Q(P_1)$ and $Q(P_2)$ is given in Figure 4.8. Only the roots of the trees of $Q(P_1)$ and $Q(P_2)$ play a role in the pasting.

We start with the leftmost root of $Q(P_2)$, R_6 in our case, The frame tree (the full binary tree with 32 external nodes) tells us whether its competant is to its right or to its left. In any case it is the closest root and in our example it is R_5. The values of R_6 and R_5 are compared and a new root R_9 is formed. Its sons are R_5 and R_3. The minimal of the values of R_5 and R_6 is stored in R_9. The node that contributed the minimum gets zero as its new value and the other one gets the difference between the two values as its new value -- exactly as we did in the insertion algorithm. Following the frame tree R_9 should now be compared with R_4 and a new root R_{10} will be formed. The value transformations are the same as before. Now R_{10} is compared with the closest root to its right, namely R_7 and a new root R_{11} is formed. Now R_{11} should be compared with the closest root to its right, namely R_8. However R_8 is not in the same level as R_{11} and therefore the algorithm terminates yielding the queue shown in Figure 4.9. The reader can easily extend this example to a general algorithm, which obviously has a logarithmic time bound.

4.3 Storage Schemes and Space Bounds.

In this subsection we shall specify exactly how s-b-queues are stored in a way which supports all the path section algorithms that have been described above and also operations # 6, 7 and 8. We'll conclude with a very short discussion on the space linearity.

Definition. We say that a vertex v belongs to a path section P if v is the tail of an edge in P (i.e., the last vertex of P does not belong to P).

Following the underlying algorithm, it is easy to verify that a vertex does not belong to more than one path section at a given moment. (See Lemma 3.1.) This fact enables us to store the information associated with a given path section P and its s-b-queue Q(P) , not only in its edges but also in the vertices which belong to P . As we shall soon see, there is a very "natural" way to do it. In Figure 4.10 we show a path section P and its associated s-b-queue. The dashed lines demonstrate a natural mapping of the tree nodes into the edges and vertices of P , The external nodes are mapped into edges and the rest -- into vertices,

The edges of the s-b-trees represent two-way pointers that enable us to move up and down in the trees and perform the cutting and the bottlenecks deleting algorithms. The insertion and pasting algorithms require another set of pointers which are called peak pointers. Two such pointers are associated with every root node of an s-b-tree and enable us to locate the neighbor root nodes (of other s-b-trees in the same s-b-queue) from left and right, in constant time. Another couple of peak pointers is required for each path section P . One is stored at the first (leftmost) vertex of P and points to the first peak and the other is stored at the last vertex of P and points to the last peak. In Figure 4.11 we give a one-dimensional picture of the s-b-queue of Figure 4.8, indicating the way in which the queue nodes are stored in their corresponding edges and vertices, The "curly" pointers are the peak pointers. All the pointers are two way pointers and therefore they are drawn as undirected edges.

Note that we can get from any vertex of P to the last one in -logarithmic time by climbing to the root of the tree in which we are and then use the peak pointers to get to the rightmost vertex. This solves the problem of locating the vertex from which we have to resume the search. Marking all the vertices that belong to any path section will solve the problem of recognizing that we have encountered a path section.

If we want to be completely rigorous we still have to show exactly how peak pointers are used and updated in each of the path-section algorithms, However, this is quite a straightforward technique and we leave it to the reader.

Finally, the most important thing, how do we compute the final residual capacities of each edge. Obviously, all the deleted edges have zero residual capacity and those which have not been encountered have zero flow. Those which have been encountered and not deleted are stored in an s-b-queue upon termination of the algorithm. If we sum up the values of the nodes from the one representing a given edge upwards to the root of the s-b-tree to which it belongs, we obtain its residual capacity. This operation can be done in linear time if we start from the root and go to all the edges in the tree simultaneously.

Space Linearity.

Conventional data structures for representing a flow network and supporting a depth first search in linear space, can be found all over.

Our additional data structure requires six more fields for each vertex and four for each edge, and therefore uses linear space too.

The six vertex fields are: One for the value of the node associated with it, two pointers to the vertices which represent its sons and one to its father. Finally we need two peak pointer fields.

An edge needs one field for the value of its node, one to point to its father, and two peak pointer fields.

5. Complexity.

In this section we'll evaluate the complexity of the restricted problem, showing that it is bounded by $O(I \log^2 I)$. This yields an $O(n I \log^2 I)$ time bound for the whole algorithm.

The depth first search and the final evaluation of the flow value at each edge take linear time.

Deletions of bottlenecks and insertions of new edges to growing f.a. paths, take logarithmic time per edge (deleted or inserted) and thus, they sum up to a total of $O(I \log I)$ time. Both cutting and pasting take logarithmic time. Since they always occur together (in fact, if we encounter the first vertex of a path section, cutting is not required but we'll assume that it is performed) we shall consider them as one unit time operation and call it CP . Thus, in order to establish our time bound, we just have to show that the number of CP's is $O(I \log I)$ and that's what we are going to do.

Let $E = \{e_1, \ldots, e_m\}$ and let $\pi_1, \pi_2, \ldots, \pi_r$ be all the f.a. paths in the order in which they were generated. Since each such path is associated with at least one bottleneck that disconnects it, we can deduce that $r < m$.

We are going to show that the number of CP's is bounded by $4m(1 + \log r)$ by demonstrating a way to assign all the CP's to edges in such a way that no edge will be associated with more than $4(1 + \log r)$ CP's.

Given two f.a. paths π_i and π_j and a vertex v , we say that π_j splits π_i at v if π_j is the first path after π_i that enters v not through the same edge as π_1 . Note that one path can split several others but can be split by at most one other path. Moreover, every CP that occurs at v is caused by some path π_j splitting another path π_i for some $i < j$.

The Charging Rule. Given a CP that occured at v when π_j split π_i, let π_k be the path that split π_j at v (if such a one exists). This CP will be charged to the right account of' π_i if either $j-i \leq k-j$ or π_k does not exist, and will be charged to the left account of π_k if $k-j < j-i$.

Thus, for every f.a. path we'll maintain two accounts in which we'll store the CP's assigned to it. In the right account we'll store CP's that were caused by later paths and in the left account we'll store those that were caused by previous paths.

Let's try now to trace the right account of a given f.a. path π_{i_0} . The CP's that are charged to this account can be ordered according to the vertices at which they occur -- from s to t . Let $e_{b_1},\dots,e_{b_r}$ be the bottlenecks of π_{i_0} . Removing these bottlenecks we split π_{i_0} into r+1 sections $\pi_{i_0}^0,\dots,\pi_{i_0}^r$. We are going to show that no more than 1 + log r CP's will be charged to our account in any of these sections. If we show that we are almost done, the edge e_{b_1} will be charged for the CP's that occured along $\pi_{i_0}^0$ and $\pi_{i_0}^1$ and e_{b_j} will be charged for the CP's that occured along $\pi_{i_0}^j$ for $2 \leq j \leq r$. The right account of π_{i_0} will thus be cleared. Since these edges were saturated by π_{i_0} they cannot clear the right account of any other path. Later on we'll see that each of them might be used once more to clear the left account of a given path, namely the first path that passed through it.

Given any section $\pi_{i_0}^j$ we have to show now that no more than 1 + log r CP's were charged to our account in this section. Let's assume that all the CP's that were charged to π_{i_0}'s right account at this section occured at $v_{i_1},\dots,v_{i_k}$ (numbered in the s → t direction) and were caused by $\pi_{i_1},\ldots,\pi_{i_k}$ respectively. Since π_{i_0} did not saturate any edge between v_{i_1} and v_{i_2} we know that either π_{i_1} or some π_ℓ for $i_0 < \ell < i_1$ passed through v_{i_2}. In any case, since the right account of π_{i_0} was charged for the CP that was 'caused by π_{i_2} at v_{i_2}, we know that $i_2 - i_0 \le \frac{1}{2}(i_1 - i_0)$. This argument can be repeated k times to yield

$$1 \le i_k - i_0 \le \frac{1}{2^{k-1}}(i_1 - i_0) \le \frac{1}{2^{k-1}} \cdot r$$

which implies that $k \le 1 + \log r$.

An almost symmetric argument holds for left accounts. In this case, however, we don't use bottlenecks to split π_{i_0} but those edges for which π_{i_0} was the first to pass through. The details are left to the reader. In general an edge can be charged for $4(1 + \log r)$ CP's at most and that bounds the number of CP's by $4m(1 + \log r)$.

The following example shows that this bound is tight up to a constant factor. In Figure 5.1 we show a network in which 8 f.a. paths are generated and 22 CP's are performed. This structure can be easily generalized to a network in which $r = 2^k$ f.a. paths are generated and $\sum_{i=1}^{k} i \cdot 2^{i-1} = O(r \log r)$ CP's are executed. $O(r \log r) = O(m \log r)$ since $m = 3r-1$ in these networks, Note that the order in which the f.a.-paths are generated, is very important and the number of CP's might decrease if we generate them in another order.

6. Summary.

The maximum-flow problem has a long history of solutions which keep improving all the time.

This one is somewhat different from the last three improvements of Karzanov, Cherkasky and Galil in two points.

1. Its underlying algorithm is much simpler and its complexity is shifted to the data structure.
2. It seems that this algorithm can be generalized to finding maximal (not maximum) flow in any directed acyclic flow network within the same time bound of $O(I \log^2 I)$. The complexity proof obviously works for general acyclic graphs. In a first glance, the data structure seems to rely heavily on the layered structure of the graph. However, directed acyclic graphs also have a natural layered structure. It seems that the s-b-queues and the path section algorithms can be generalized to these graphs with minor modifications. This observation suggests that we might reduce the number of phases in Dinic's algorithm by taking larger graphs in each phase.

References

[B] Brown, Mark R., "The analysis of a practical and nearly optimal priority queue," Stanford University Computer Science Department Technical Report STAN-CS-77-600 (1977).

[C] Cherkasky, B. V., "Algorithm of construction of maximal flow in networks with complexity of $O(V^3\sqrt{E})$ operations," Mathematical Methods of Solution of Economical Problems 7(1977), 117-125 (in Russian).

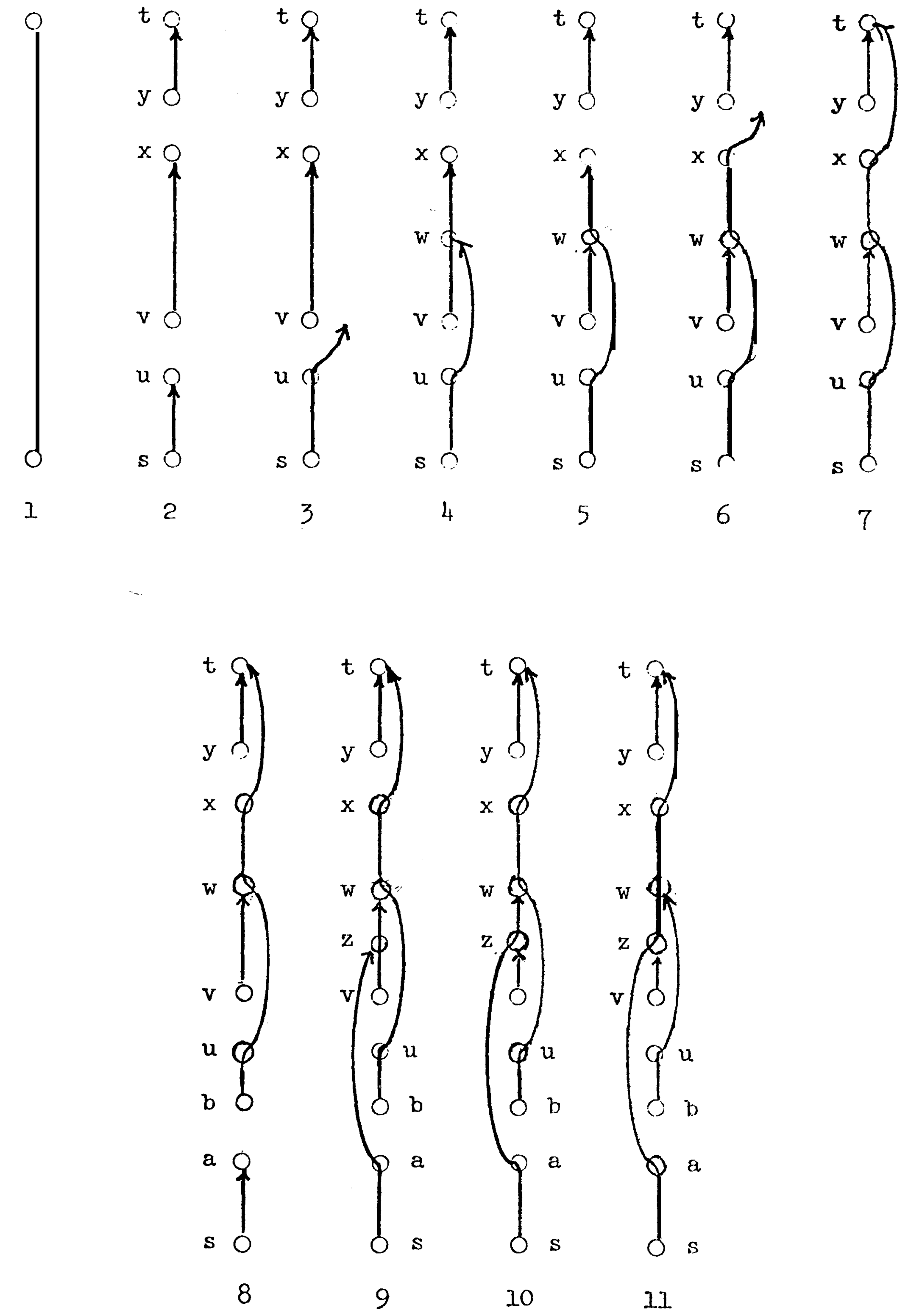
[D] Dinic, E. A., "Algorithm for solution of a problem of maximal flow in a network with power estimation," Soviet Math. Dokl. 11(1970), 1277 -1280. --

[E] Even, S., "The max-flow algorithm of Dinic and Karzanov: An exposition," M.I.T., LCS, TM-80, (December 1976).

[FF] Ford, L. R. and D. R. Fulkerson, "Maximal flow through a network," Canadian J. of Math. 8(1956),399-404.

[G] Galil, Z., "A new algorithm for the maximal flow problem," Proceedings 19th IEEE Symposium on Foundations of Computer Science, Ann Arbor, Mich., October 1978, 231-245.

[K] Karzanov, A. V., "Determining the maximal flow in a network by the method of preflows," Soviet Math. Dokl. 15(1974), 434-437.



Figures 3 1   3.11.

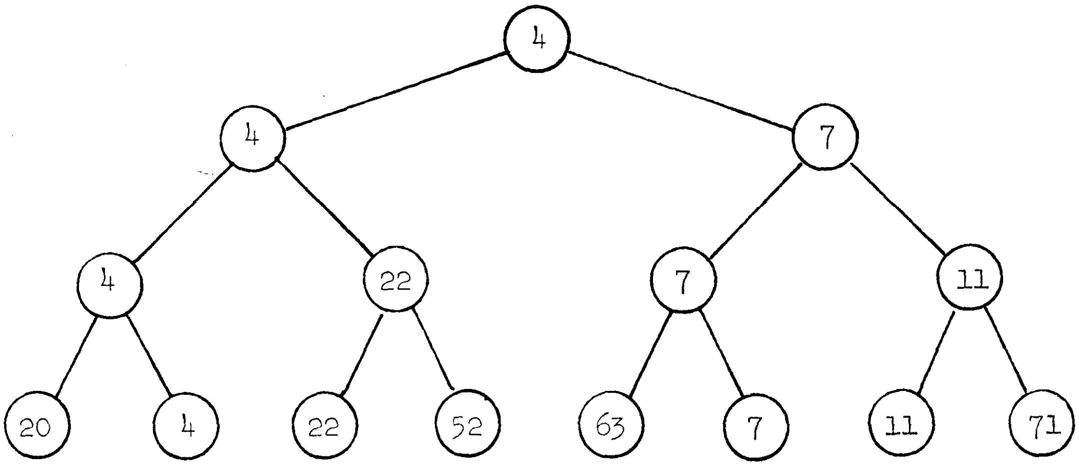


Figure 4.1.

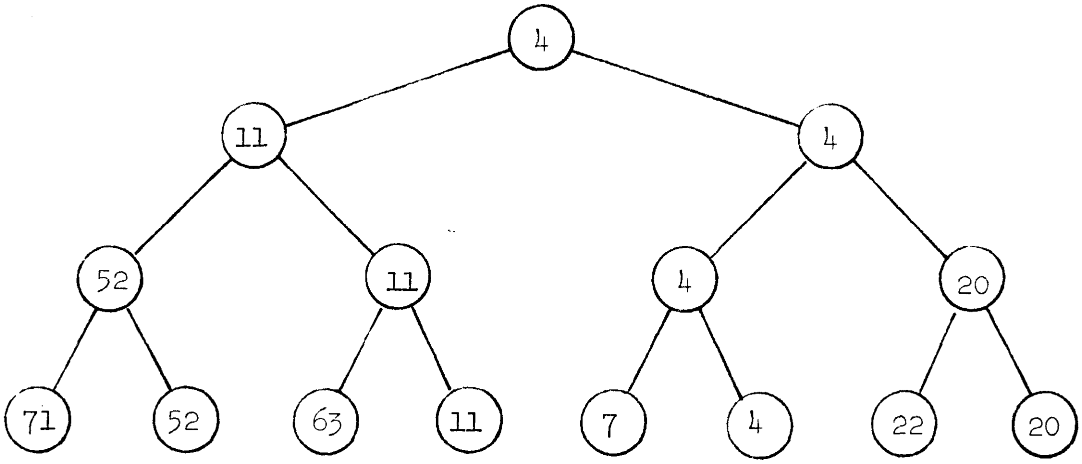


Figure 4.2.

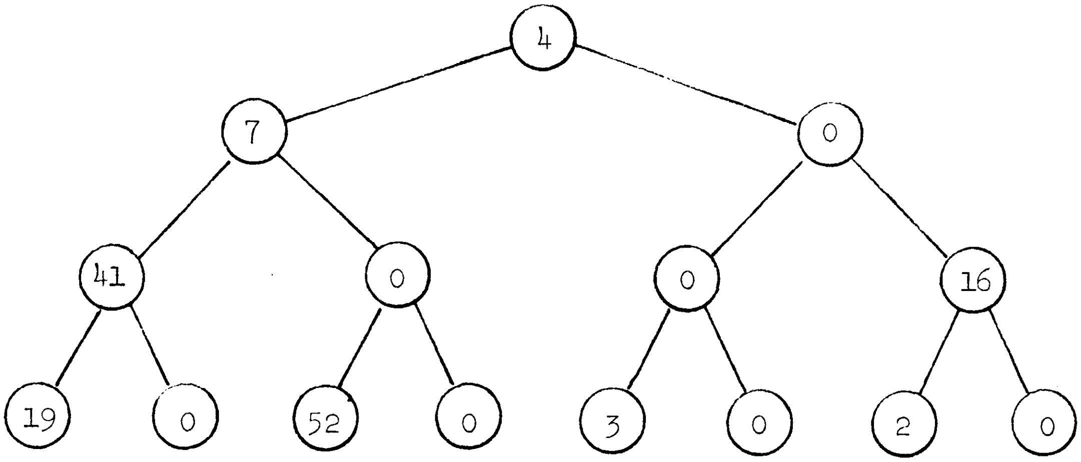
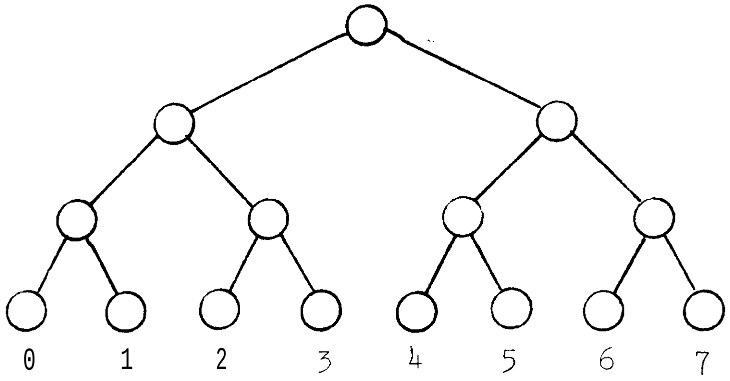


Figure 4.3.



The forest determined by:

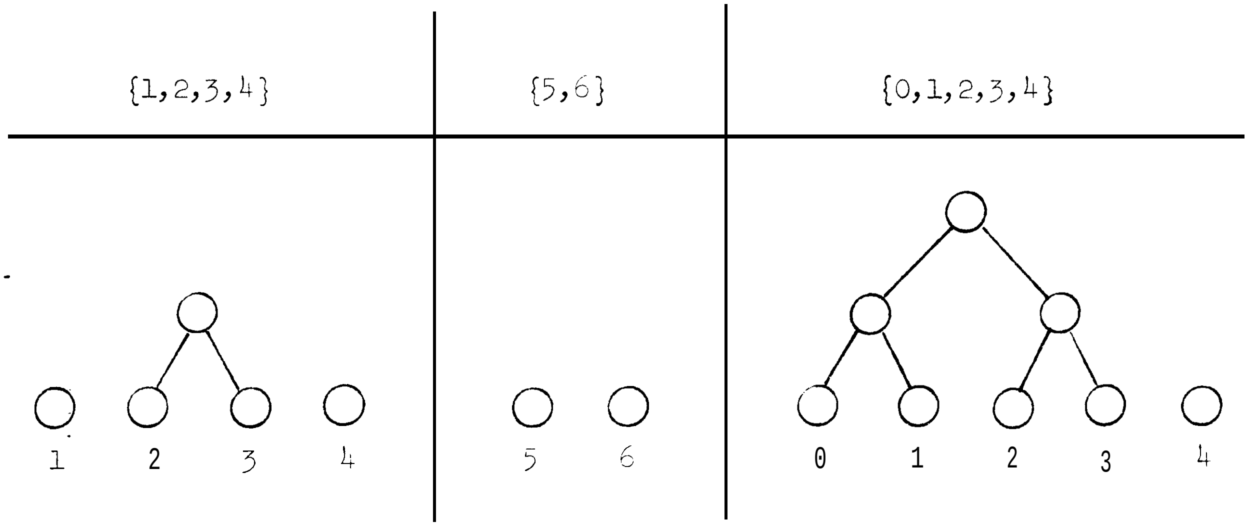
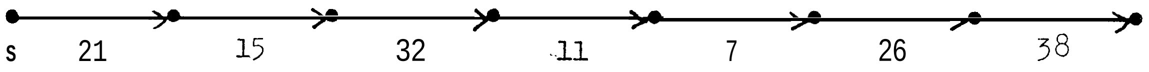


Figure 4.4.

The path:



The corresponding queue:

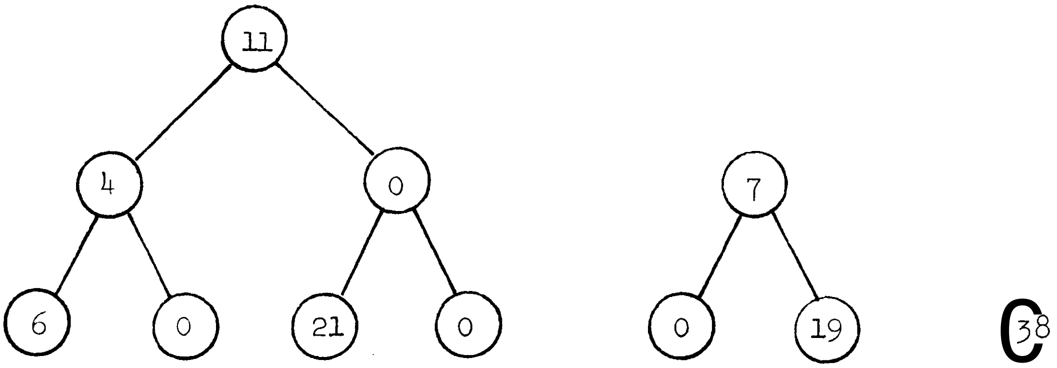
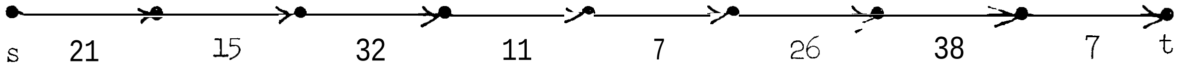
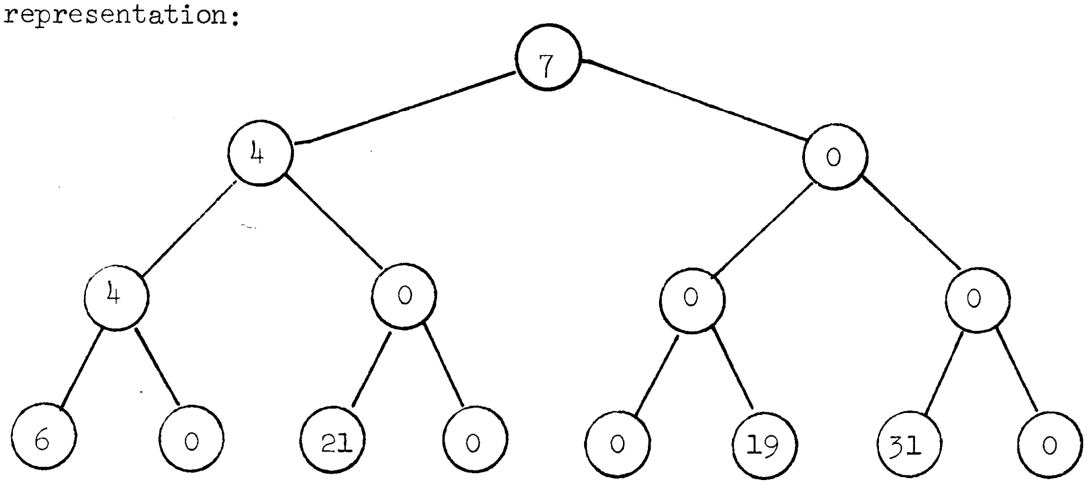


Figure 4.5.

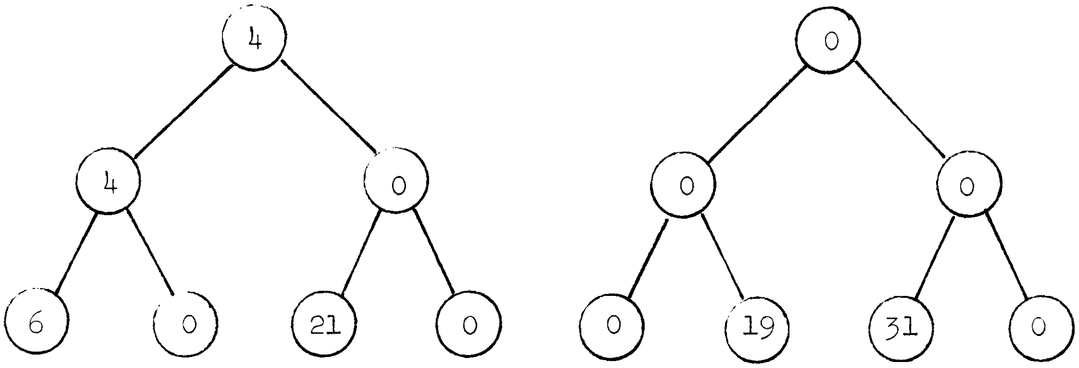
The path:



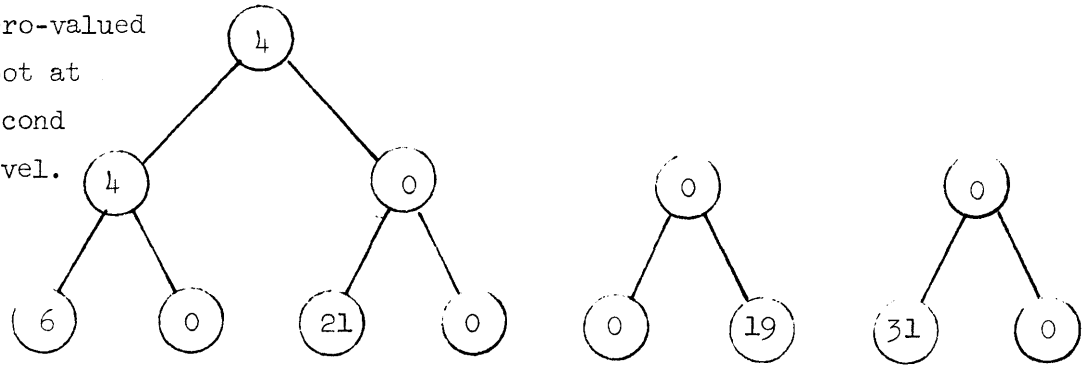
Its representation:



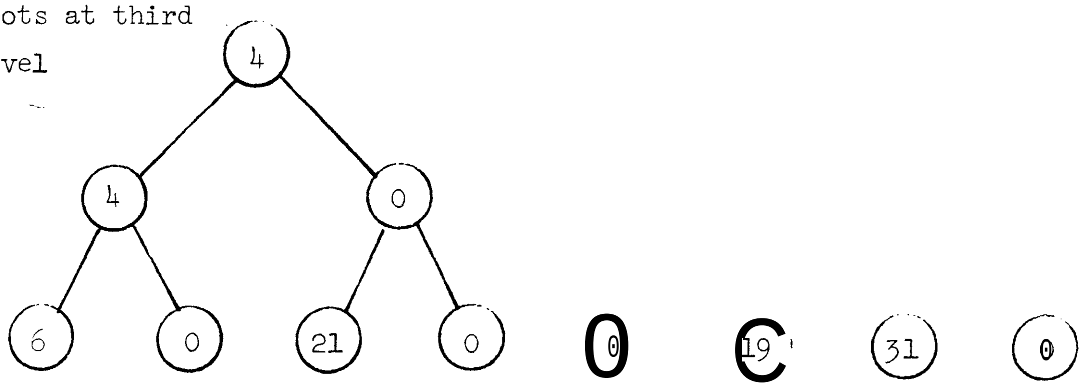
-Step 1. Deleting the root.



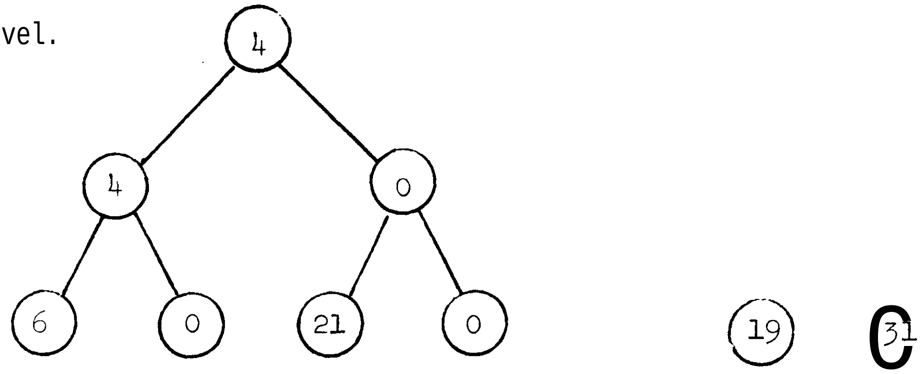
Step 2. Deleting a zero-valued root at second level.



Step 3. Deleting zero valued roots at third level

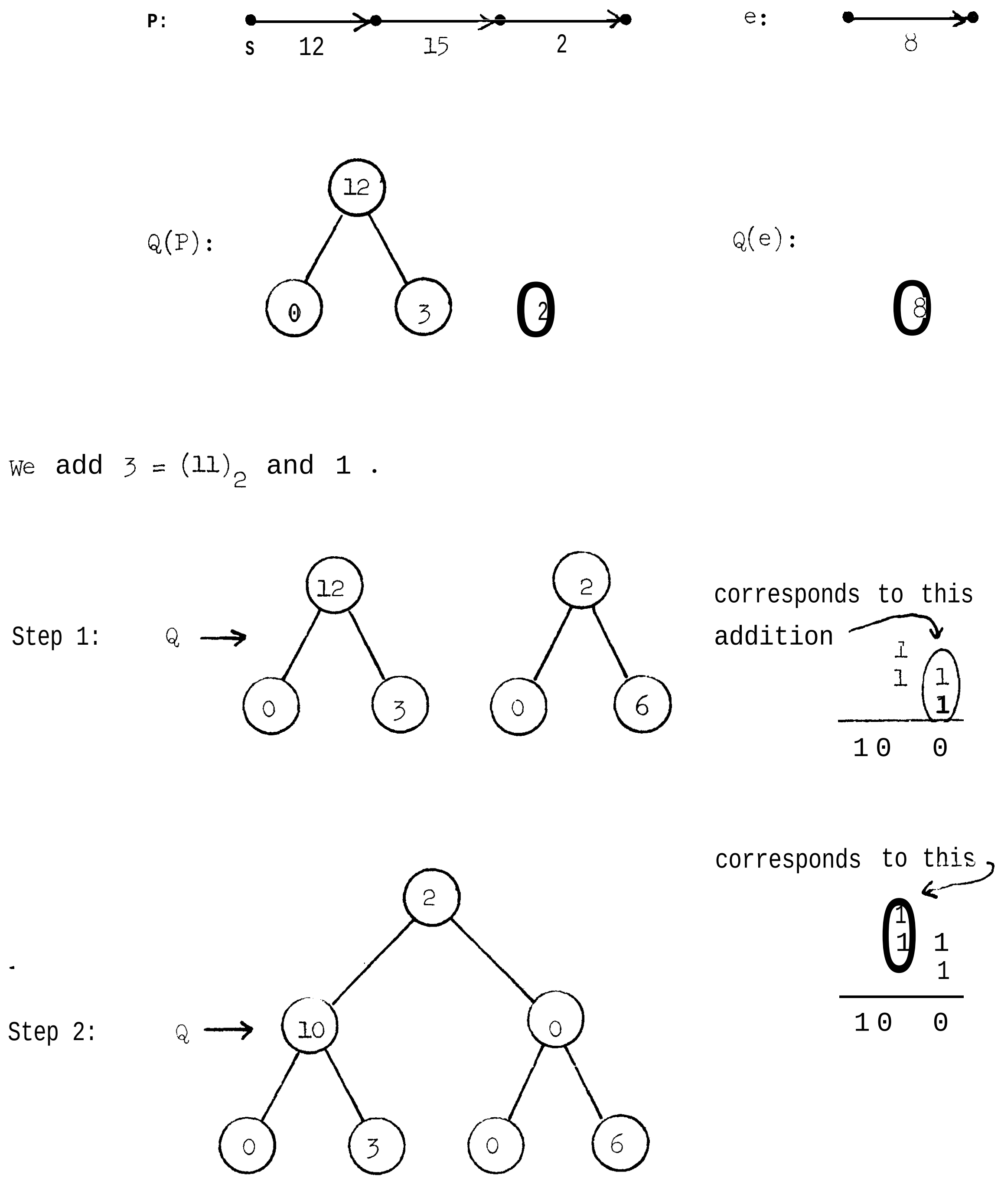


Step 4. Deleting zero valued roots at fourth level.



Step 5. The 5-th and 8-th edges are deleted from the graph.

Figure 4.6.



We add $3 = (11)_2$ and 1 .

Figure 4.7.

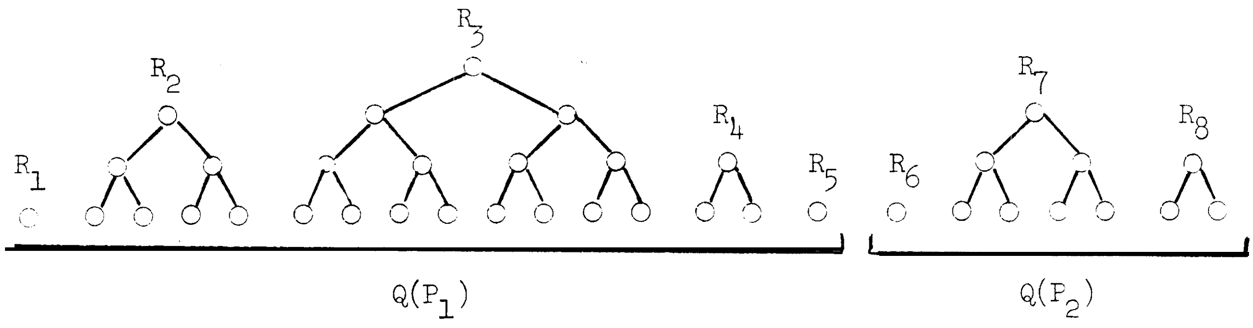


Figure 4.8

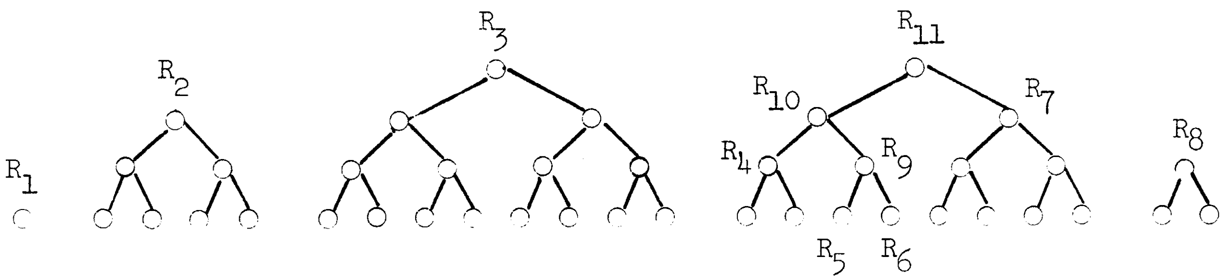


Figure 4.9

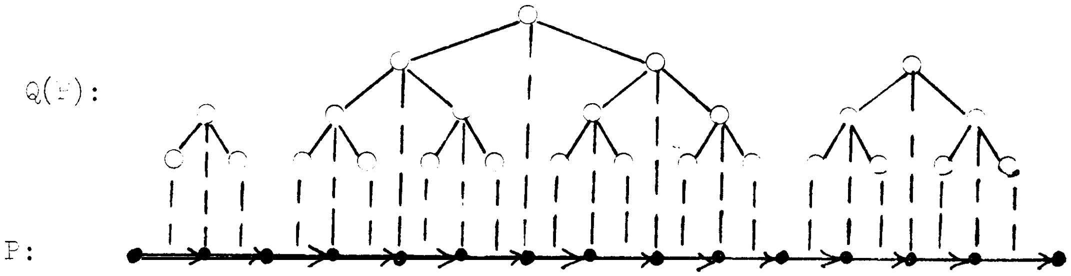


Figure 4.10.

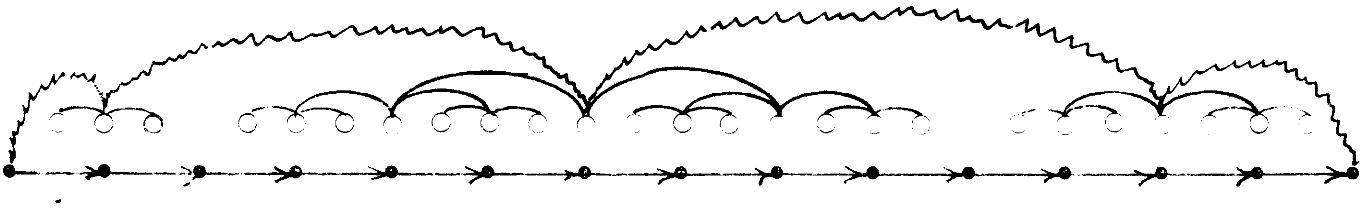


Figure 4.11.

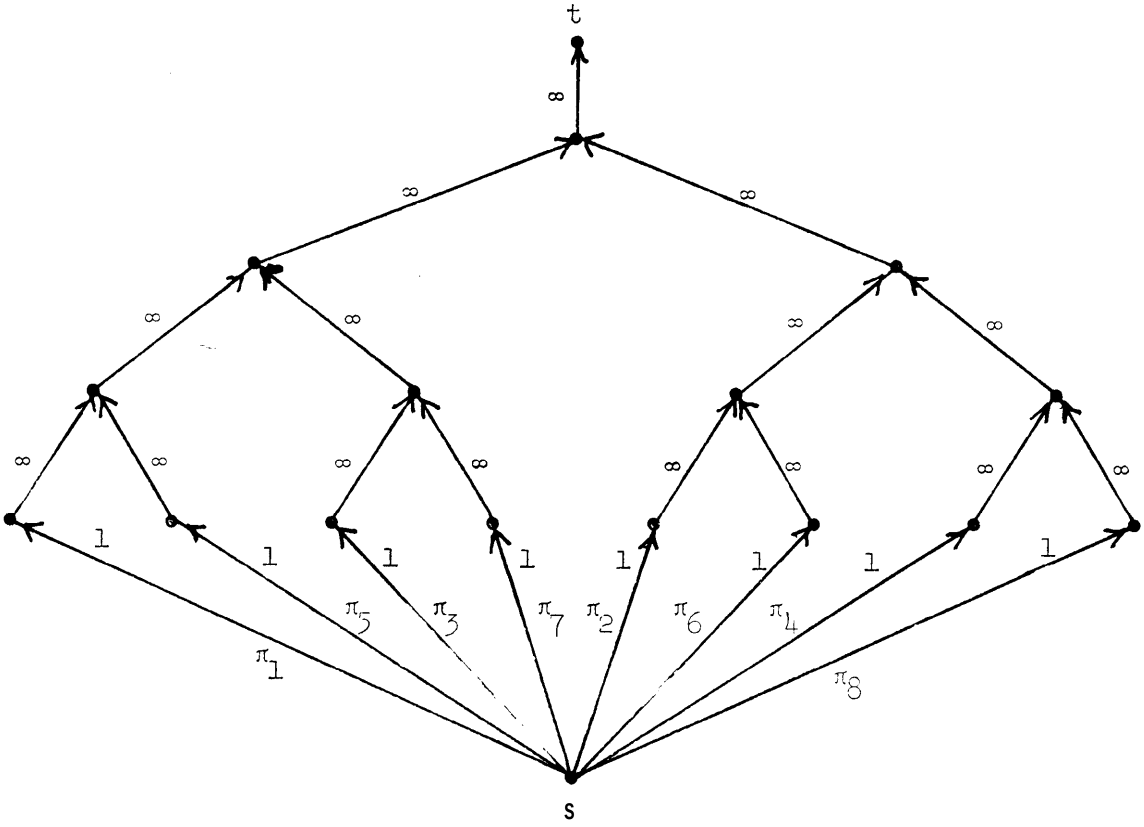


Figure 5.1.