数理解析研究所講究録
第236巻 1975年 111-123

On Data Structures for Manipulating Graphs and a New Efficient Program of the Dijkstra Method for the Shortest Path Problem

(Extended Abstract)

Kohei NOSHITA
(Information Science Laboratories, University of Tokyo)

In this paper data structures for manipulating graphs are briefly discussed and as their application a new efficient program of the Dijkstra method for the shortest path problem is presented with investigations on its computational efficiencies.

Recently many efficient algorithms for graph problems have been proposed. Most of them are based on the data structures which facilitate efficient searches and modifications in graphs. For example see [4] and [6].

A philosophy on data structures for manipulating graphs will state that in most cases it is efficient to represent incidence structures of graphs in computer memories as faithful as possible. In other words a graph should be represented by list processing techniques in such a way that searching and modifying some considered objects in a graph should be performed without any accesses to unrelated objects. Most basic operations such as a search of outgoing branches from a given node, a search of directly connected nodes from a given node, a deletion of a given branch and the like, should be done within constant number of steps not depending on the sizes of graphs.

Now we present an outline of the data structures for representing a graph in the programming language Pascal [9].

```
type node = record out, in: record degree: integer;
                                           natural,
                                           reverse: ↑ branchset
                                end
            end;
     branch = record from, to: ↑ nodeset;
                      out, in: record natural,
                                      reverse: ↑ branchset
                               end
              end;
var  nodeset : class 512 of node;
     branchset : class 1024 of branch;
```

An oriented graph is mathematically defined as a function from a set of branches to a set of ordered pairs of nodes. The function is represented in data of branch type, in each of which two incident nodes can be selected by the selectors "from" and "to."

However for practical reasons it is convenient to explicitly represent the inverse of that function, which is represented in data of branch type and node type. For each node the set of its outgoing branches is arbitrarily lined up in the linear order. By the selector "out.natural" the node datum refers to the datum of the first branch in that set and by the selector "out.natural" the branch datum refers to the datum of the next branch in that set. The last branch datum refers to nil. The selectors "out.reverse" are used to represent the reverse order of that set. In other words that set is represented in a two-way list.

As for the set of incoming branches a similar two-way list is constructed by the selectors "in.natural" and "in.reverse."

For each node it is recommended to calculate its degree (the number of outgoing or incoming branches), which is selected by the selector "degree."

Other related data possibly necessary for each applied problem may be added in record data of node type and/or branch type.

The constant numbers 512 and 1024 show the maximum numbers of data of node type and branch type dynamically allocated by

the standard procedure "alloc" in nodeset and branchset, respectively.

In general it is desirable to take the multiple usage of memory spaces into consideration.

There may be some applications for which it is convenient to provide the data for the connection relation between nodes.

It will be easily ascertained that in such a data structure as described above most useful basic operations for graph manipulations can be directly done without any loss in time. It should also be noticed that the required amount of memory spaces is linearly proportional to the size of the represented graph.

The idea just presented is essentially due to Iri [6], where (Fortran's) statically allocated arrays are used for expositions. For further detailed discussions see also [7].

The shortest path algorithm is one of the most important algorithms in network problems. Many algorithms on this subject have been studied not only theoretically but also experimentally [2, 5].

For the problem to find shortest paths from a specified node (source) to all other nodes, it is well-known that the Dijkstra algorithm [1] through direct implementation is the most efficient one in the sense of evaluating the upperbound of computational steps, provided that every assigned distance between two nodes is a nonnegative (real) number. This direct method fits well to such graphs as having dense branches, e.g., complete graphs. However it is far less efficient for sparse graphs, which occur quite frequently in practical problems.

Hence many practical algorithms have been devised [2], but so far no satisfactorily general methods, which efficiently cover both cases of dense graphs and sparse ones, have been obtained.

We propose a new program which is practically most efficient to the author's knowledge. The algorithm is so composed in order to make it possible to be parametrized that it coincides with the direct method as a particular instance, if a given graph is enough dense. In case of sparse graphs it runs much more efficiently than the direct method by choosing the appropriate parameters.

We shall review the Dijkstra algorithm and some related matters. Let d_{ij} be a distance of real number from node i to node j in a given graph. Under the condition $d_{ij} \geq 0$ for any i and j, the problem is to find all shortest paths from the source to other nodes. The outline of the Dijkstra algorithm is shown in the following. Let P and T be subsets of ordered pairs <i,v>, where i is in the set of all nodes in the graph and v is a nonnegative real number. We shall use a notation v_i for the associated value of node i. For the sake of simplicity the suboperations to explicitly obtain the shortest paths during computations are omitted in the following algorithm.

```
begin
initialize: P := φ ;
            T := {<source,0>} ;
   while T ≠ φ do
      begin
      step A: search <m,v_m> such that v_m ≤ v_i for any
              <i,v_i> in T;
              T := T - {<m,v_m>} ;
              P := P ∪ {<m,v_m>} ;
      step B: for j such that j is incident from m
                        and <j,v_j> is not in P do
              begin
                 w := v_m + d_mj ;
                 if <j,v_j> in T
                 then begin
                         if w < v_j
                         then replace <j,v_j> in T
                              by <j,w>
                      end
                 else T := T ∪ {<j,w>}
              end
      end
end
```

For the sake of comparative studies two methods will be roughly explained, both of which have been extensively studied [2, 8].

(I) Direct method

This can be directly implemented without any sophisticated techniques. A one dimensional array for T with size n (the number of nodes) should be prepared in the worst case. In step A the minimum valued node is sequentially searched from the first node to the last node in that array. In step B the value of the considered node in that array can be replaced independently of other nodes, if necessary.

(II) Linear (two-way) list method

In order to speed up searches in step A the set T is always linearly sorted with respect to values in the ascending order from top to bottom in the two-way list. The minimum search only requires an access to the topmost node in the list. In step B if the considered node exists in the list, the new position of that node will be found in the upperpart of the list from the original position. On the other hand if the considered node is new, the node will go up looking for its proper position from bottom toward top.

We now present the new method called "multitree method."

(III) Multitree method

In this method the ordered τ balanced binary sorting trees (heaps) are used for partial sortings of T. In a one dimensional array M, τ trees are constructed as shown in Fig. 1 (see the data structure used in Floyd [3]). The first τ words correspond to τ roots of trees, while the last word indicated by λ to the last point of trees, where the natural ordering of points in τ trees is understood according to indices in M. Thus $\lambda = 0$ means $T = \phi$. Note that all τ trees have almost equal heights. Each point (i.e., word in M) in a tree refers to a valued node in T. For any two points p, q in a tree the value of the node to which p refers is smaller than or equal to that to which q refers, if p is located along the route from the root to q.

The program with parameter τ for step A and B in the multi-tree mehtod proceeds as follows. Assume that the necessary initialization has been completed.

step A: begin

Search a point of index p_m which refers to the minimum valued node m among indices from 1 to min $\{\tau,\lambda\}$ in M;

$T := T - \{<m,v_m>\}$;

$P := P \cup \{<m,v_m>\}$;

$M(p_m) := M(\lambda)$;

Change a reference in the node, to which the point of index λ refers, from λ to p_m ;

$\lambda := \lambda - 1$;

if $\lambda > \tau$

then begin

In order to make the tree with the root of index p_m properly sorted, rearrange the tree from root toward leaf by iteratively comparing the point and one of its two, or possibly less, direct descendent points, until the sorting condition is satisfied, where between two direct descendent points the point referring to the node with not larger value than that of the node to which the other point refers should be chosen

end

end;

step B: for j such that [j is incident from m to which the point of index p_m refers]

$\wedge$ [$<j,v> \notin P$ for any v]

do begin

$w := v_m + d_{mj}$;

if [$<j,v> \notin T$ for any v]

$\vee$ [$<j,v> \in T$ for some v $\wedge$ $w < v_j$]

then begin

$v_j := w$;

if $<j,v> \notin T$ for any v

then begin

$\lambda := \lambda + 1$;

$M(\lambda) := j$;

Set a reference in node j to refer to λ ;

$p := \lambda$

end

else p:= index of the point referring to j ;
In order to make the tree including the point of index p properly sorted, rearrange the tree from that point toward root by iteratively comparing the point and its direct ascendent point, until the sorting condition is satisfied

end
end

We are now in a position to evaluate the efficiencies of the algorithms just described through counting times of additions and comparisons between two real numbers representing distances. Since the total times of those additions and comparisons in step A and B directly reflect to the total amount of computing time, we shall only discuss on the times of those operations.

There are two kinds of operations. The one is an operation (comparison/addition) for $v_m+d_{mj}<v_j$ in step B, and the other is a comparison in step A and B depending on each method. The former is common to all methods discussed in this paper and it requires m operations of additions and comparisons, where m is the number of branches in the graph. These operations seem to be essentially obligatory to the Dijkstra algorithm. The minimization of the other comparisons is the main goal of those studies now investigated, hence hereafter we discuss particularly on those comparisons. We shall use notations C_A and C_B for the times of comparisons in step A and step B except the obligatory operations just above mentioned, respectively. Let C_{sum} be $C_A + C_B$.

We shall examine the behaviors of the upperbound of C_{sum} through "degree" (the number of branches incident to/from a node). Let n and μ be the number of nodes and the maximum degree in the given graph, respectively. In the multitree method the asymptotic upperbounds for C_A, C_B and C_{sum} are calculated as follows, if n is sufficiently large and $\mu >> 1$ (i.e., 1 is negligibly smaller than μ).

$$\hat{C}_A = -\tau^2/2 + n\tau + 2(n-\tau)\log_2(n/\tau)$$

$$\hat{C}_B = \mu(n-\tau)\log_2(n/\tau) \ ,$$

since one downward (upward) rearrangement in step A (B) at most requires 2 $\log_2(n/\tau)$ ($\log_2(n/\tau)$).

The function for the upperbound of times of comparisons reads:

$$\hat{C}_{sum} = -\tau^2/2 + n\tau + \mu(n-\tau)\log_2(n/\tau).$$

In order to see the behaviors of this function, we define

$$f_\nu(x) = -x^2/2 + x - \nu(1-x)\log_2 x$$

with $0 < \nu \leq 1$ and $0 < x \leq 1$, where $\nu = \mu/n$ and $x = \tau/n$.
Note that $\hat{C}_{sum} = n^2 f_\nu(x)$.

Some illustrations of the behaviors of $f_\nu(x)$ are shown in Fig. 2.

If $\nu \geq \log_e 2/2$, then the minimum value $f_\nu(x)$ takes 1/2 at $x = 1$. Hence if $\mu \geq 0.34n$, n should be chosen for τ, in which case the multitree method is identical to the direct method.

Let α be $\nu/\log_e 2$. If $0 < \alpha < 1/2$, that is $0 < \nu < \log_e 2/2$, the optimal value x_m yielding the minimum $f_\nu(x_m)$ is given as the zero point of the function:

$$f_\nu'(x) = -x + 1 + \alpha\log_e x + \alpha(x-1)/x ,$$

where $\alpha < x < \alpha(1+\sqrt{1+4/\alpha})/2$.
The zero point x_m should be calculated by some numerical method every time a new graph is given.

Note that the equation $f_\nu'(x) = 0$ has exactly one solution if x is in the domain $(\alpha, \alpha(1+\sqrt{1+4/\alpha})/2)$.

If α approaches to 1/2, x_m approaches to 1. If α approaches to 0 decreasingly, x_m approximately becomes to α. In other words if the maximum degree μ is sufficiently smaller than the number of nodes n, e.g., if $\mu \simeq \log_2 n$, $\mu/\log_e 2$ may be taken for the optimal number of trees. In this case $\hat{C}_{sum}$ is nearly $\mu n \log_2 n$.

We should mention the procedure to decide μ. The computational labor for deciding μ has a different meaning from the comparisons between two real numbers, but it should be taken into consideration. In the actual program it is possible to decide μ within steps proportional to the number of branches by a bit deliberate programming, which anyhow would not become considerably large. But as mentioned during discussions on data structures each degree of node may be easily obtained during such operations as an input of a graph, then it is recommended to calculate each degree in

the graph representation itself.

It has already been pointed out that the multitree method with n trees coincides with the direct method, hence the multitree method may be considered as an efficient generalization of the direct method.

In the direct method if a complete graph is given, C_{sum} is exactly $n(n-1)/2$. It can be shown that in the linear list method $n^3/6$ comparisons are necessarily required, if the complete graph with trickily assigned values to branches is given. This shows the theoretical defficiency of the linear list method, which has been written in [9].

We shall investigate our methods from the viewpoint of actual programming and experiments. The direct method is simple and straightforward in programming. The multitree method, as well as the linear list method, is slightly complicated. They have equally comparable sizes of programs. They also equally require the amount of memory spaces. As for the multitree method it is necessary to use four words per node besides those for the incidence relation, i.e., a word for the distance from the source, for a pointer referring to the incoming branch on the shortest path, for a pointer referring to array M and a word in M itself allocated in the separate place. For a branch a word for the assigned distance is necessary, besides those for the incidence relation.

A small computer FACOM 270-20 was used for the experiments. All programs considered were written in Fortran. Three test graph generators were prepared. They generate complete graphs, almost regular (having almost equal degrees) graphs and two dimensional lattice graphs. Values assigned to branches are generated by a (pseudo) random number generator. In Fig. 3, No. 256, 512 or 917 indicates the name of the random number generator, though No. 917 generates a number sequence with a short period hardly to say as random.

The experimental results show the strong dependence of times of comparisons upon given graph structures. Other results not displayed in Fig. 3 show very similar behaviors according as their

graph structures.

We shall explain some results shown in Fig. 3. All graphs considered are nonoriented. For example see the second curve from the topmost. The given graph has 100 nodes and it is almost regular with degree 9. The incidence relation and values are randomly decided by the generator No. 256. In this case the linear list method requires 3099 comparisons, while the direct method 3837 comparisons. If we choose the optimal number of trees the multitree method requires only 1014 comparisons. The obligatory additions/comparisons amount to 447.

It is remarkable that those probably average behaviors of times of comparisons show the quite faithful resemblance with the theoretical upperbound function. However notice that the optimal number of trees does not sufficiently coincide with the theoretically obtained number for the upperbounds.

Finally we supplement two comments.

Slightly better balanced binary trees may be considered in the same data structure used in the proposed multitree method, but during index calculations of predecessors or of successors it would be necessary to detect in which blocks the considered index exists, hence it is not used here.

A multi-linear list method may also be devised as a generalization of the linear list method, but the overheads of list processings in time and space would become larger. And it will be theoretically inefficient because of its excess of sortings as is the same case in the simple linear list method.

ACKNOWLEDGMENTS

The author wishes to express his sincere gratitudes to Professors M. Iri and A. Nozaki for their valuable discussions and also to Mr. M. Sassa for his useful curve-plotting subroutine on the lineprinter.

REFERENCES

[1] Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik, 1 (1959), pp.269-271.

[2] Dreyfus, S. E., "An Appraisal of Some Shortest-Path Algorithms," Operations Research, Vol.7, No.3 (1968), pp.395-412.

[3] Floyd, R. W., "Algorithm 245: TREESORT3," CACM, Vol.7, No.12 (1964), p.701.

[4] Hopcroft, J. and R. Tarjan, "Efficient Algorithms for Graph Manipulation," STAN-CS-71-207, Stanford University (1971), 19 pp.

[5] Iri, M., "Recent Developments in Theories and Techniques for Network Problems (a survey)," Keiei Kagaku, Vol.16, No.2 (1972), pp.75-87, in Japanese.

[6] Iri, M., "On Techniques for Processing Informations with Network Structures," Proceedings of Joint Conference of Four Electrical Societies (1972), pp.850-853, in Japanese.

[7] Noshita, K., "On Data Structures for Representing Graphs," in Report of Network Working Group under the Operations Research Society of Japan (1973).

[8] Tunekawa, J., "On Implementations of the Dijkstra Algorithm for the Shortest-Path Problem," ibid.

[9] Wirth, N., "The Programming Language Pascal," Acta Informatica, Vol.1, No.1 (1971), pp.35-63.

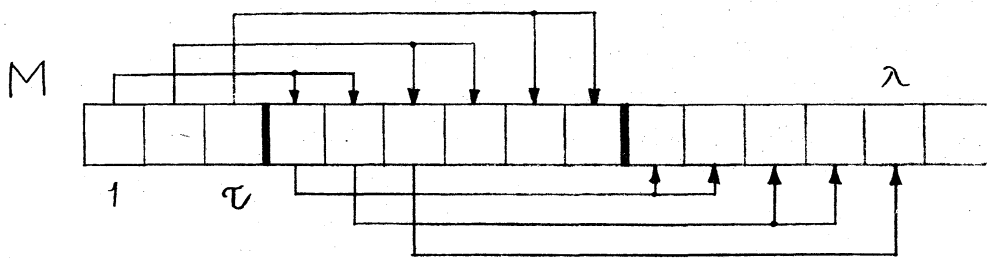

Fig. 1 τ balanced binary sorting trees

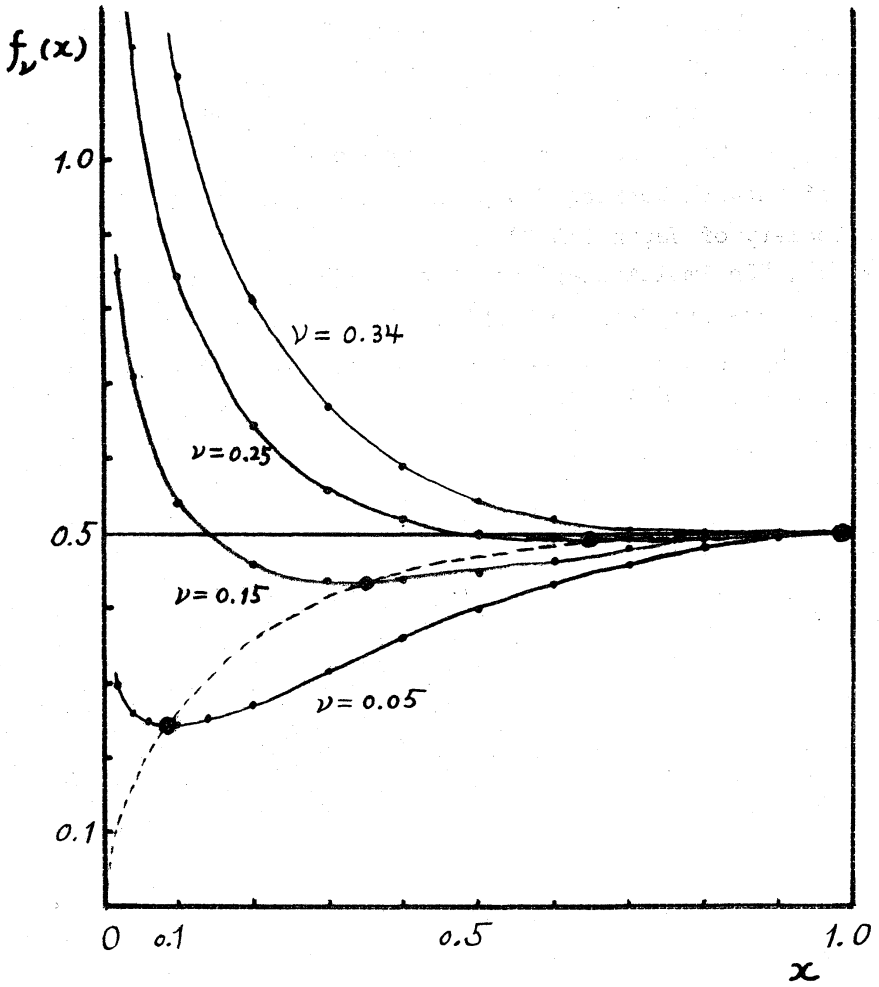

Fig. 2 Behaviors of Upperbound $f_\nu(x)$

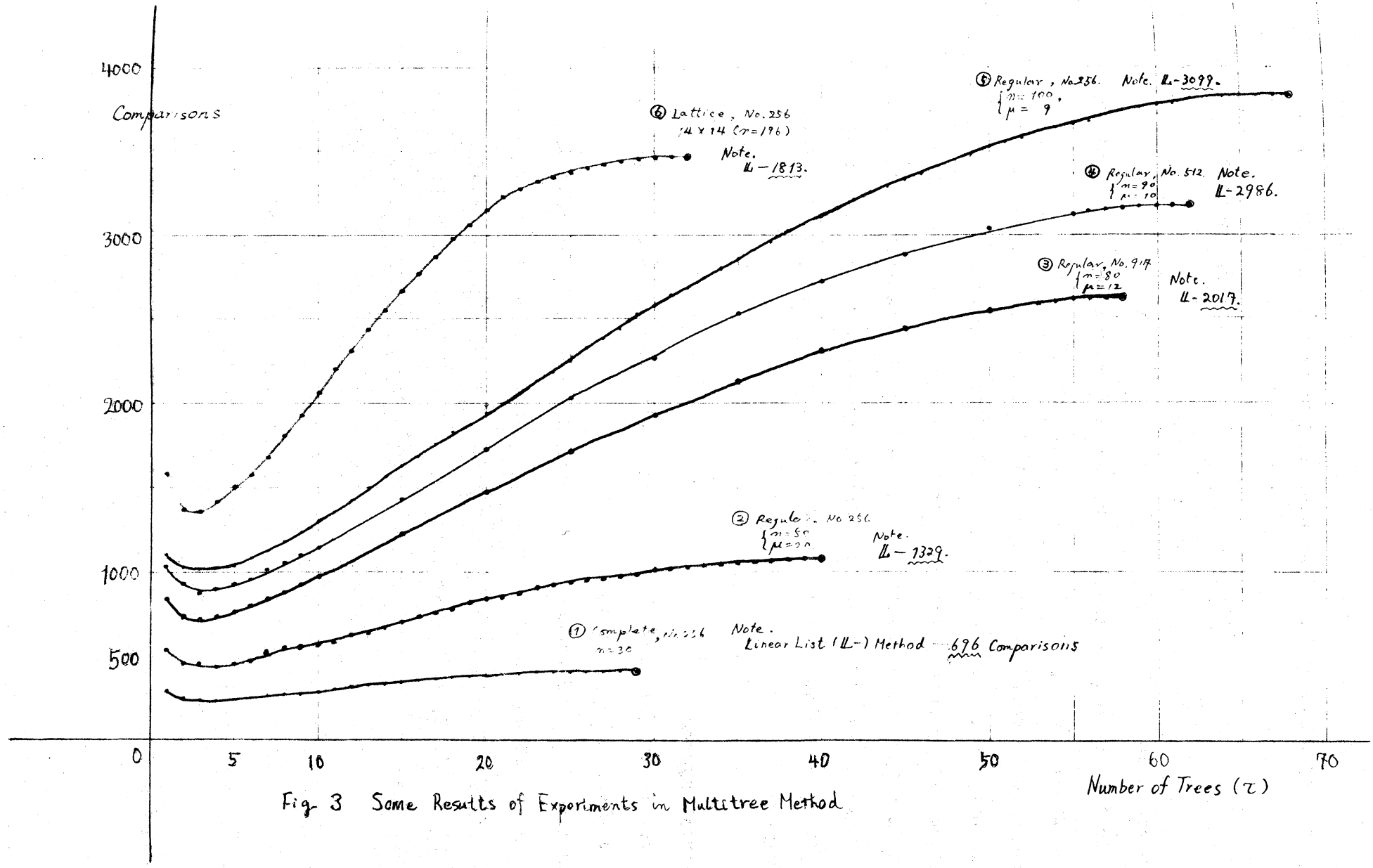

Fig. 3 Some Results of Experiments in Multitree Method