Journal of the Operations Research
Society of Japan
Vol. 43, No. 4, December 2000

PRACTICAL EFFICIENCY OF THE LINEAR-TIME ALGORITHM FOR THE SINGLE SOURCE SHORTEST PATH PROBLEM

Yasuhito Asano Hiroshi Imai
The University of Tokyo

(Received February 4, 1999; Revised June 26, 2000)

Abstract Thorup's linear-time algorithm for the single source shortest path problem consists of two phases: a construction phase of constructing a data structure suitable for a shortest path search from a given query source s; and a search phase of finding shortest paths from the query source s to all vertices using the data structure constructed in construction phase. Since once the data structure is constructed, it can be repeatedly used for shortest path searches from given query sources, Thorup's algorithm has a nice feature not only on its linear time complexity but also on its data structure suitable for a repetitive mode of queries (not single query). In this paper, we evaluate the practical efficiency of the Thorup's linear-time algorithm through computational experiments comparing with some of well known previous algorithms. In particular, we evaluate Thorup's algorithm mainly from the viewpoint of repetitive mode of queries.

1. Introduction

The Single Source Shortest Path problem (**SSSP**, for short) is one of the most popular classic problems in network optimization and stated as follows. Let $G = (V, E)$ be a connected graph with an edge weight function $\ell : E \to N$ and a distinguished source vertex $s \in V$. (We sometimes assume $\ell(v,w) = \infty$ if $(v,w) \notin E$.) Then SSSP is, for every vertex v, to find a shortest path from s to v or its distance $d(v) = dist(s,v)$. SSSP forms an important part and basis in several large-scale search problems of network optimization, and therefore, fast algorithms for SSSP are keenly required in both theoretical and practical applications. Since the shortest paths from s to all the vertices can be found in linear time if we have all the distances $d(v) = dist(s,v)$ $(v \in V)$, we assume that SSSP is to find the distances $d(v) = dist(s,v)$ $(v \in V)$. There are several variations of SSSP, but in this paper, we assume that graphs are undirected, and weights of edges are positive integers. For simplicity, we use $n = |V|$ and $m = |E|$.

For SSSP, Dijkstra's algorithm [4] is most famous, and several improvements based on Dijkstra's algorithm have been proposed. However, no algorithm based on Dijkstra's algorithm has achieved the linear-time complexity due to the bottleneck of Dijkstra's algorithm. On the other hand, Thorup obtained a linear time algorithm in 1997 for undirected graphs with integer weights of edges ([10]), by modifying Dijkstra's algorithm in such a way to avoid its bottleneck. This is now the theoretically fastest algorithm for SSSP. We note that the algorithm is based on RAM computation model and its time complexity is independent of the word length b of computers (such an algorithm is called a *strongly-trans-dichotomous* algorithm in [6] and other papers).

There are two interesting features of Thorup's algorithm making it different from the previous algorithms based on Dijkstra's algorithm.

- Thorup's algorithm contains a minimum spanning tree (MST, for short) algorithm as a sub procedure. To achieve the linear-time complexity, Thorup used a linear-time MST algorithm based on RAM described in [6]. Note that SSSP and MST were considered as independent problems.
- Thorup's algorithm consists of two phases: a construction phase of constructing a data structure suitable for a shortest path search from a given query source s; and a search phase of finding shortest paths from the query source s to all vertices using the data structure constructed in construction phase. A construction phase in Thorup's algorithm is independent of a source, while data structures in previous algorithms (e.g., heap, queue, buckets) heavily depend on the source.

These two features are very attractive in practical applications. In particular, the latter feature enables Thorup's algorithm to be used in a database like geographical information system (GIS). There, numerous queries are raised to the fixed graph G. Therefore, it is important to preprocess G so that shortest paths from a query source can be computed efficiently. We call this kind of problem as the *shortest path search problem.* Thus, algorithms for the shortest path search problem should be characterized in terms of the following three attributes:

(1) *Preprocessing time* — the time required to construct a search structure suitable for search.

(2) *Space* — the storage used for constructing and representing the search structure.

(3) *Search time* — the time required to find shortest paths from a query source s, using the search structure.

Thus, in practical applications, it is quite interesting to implement Thorup's algorithm which is theoretically fastest now and evaluate its practical efficiency through computer experiments by comparing it with those of other representative algorithms. In particular, we would like to know how efficient Thorup's algorithm is from the point of the shortest path search of view. There are several previous works on evaluating the practical efficiencies of representative shortest path algorithms [9, 8, 3]. In [9], Imai and Iri concluded that an algorithm based on binary heap is a little slower than algorithms based on buckets through computer experience, and in particular, in random graphs and grid graphs the binary heap is sufficiently fast. The algorithms based on the buckets and other data structures are analyzed in [8] and [3]. However, no works were done from the point of the shortest path search of view because all the tested algorithms have used data structures depending the source. Furthermore, it might be quite difficult to implement Thorup's algorithm since the data structures used in the algorithm are too complicated and need a huge word length.

In this paper, we implement Thorup's algorithm and evaluate its practical efficiency through computer experiments. Since the original Thorup's algorithm assumes a huge word length of a computer, we first clarify the problem in implementing it to run on today's computer and propose ideas to modify it in such a way that it runs on today's computer without loosing its efficiency. We also implement representative algorithms based on Dijkstra's algorithms, which are shown to be efficient by the previous works [9]. Among them are algorithms based on the data structures using binary heaps proposed in [11] and Fibonacci heaps proposed in [5]. We evaluate these algorithms by testing on grid graphs and on random graphs and measuring their execution times from two aspects: the whole execution time and search time. As for the whole execution time, Thorup's algorithm is very much slower than the algorithms with the heaps. In particular, the part of the linear-time MST algorithm in Thorup's algorithm dominates and occupies the most of the execution time. As for the shortest path search problem, the algorithm is expected to be fast, since each query source needs only searching in the data structure already constructed. In our experiments, the searching phase of Thorup's algorithm is faster than Dijkstra's algorithm with the Fibonacci heap, but slower than the algorithm with the binary heap.

The rest of this paper is organized as follows. In section 2, we describe Dijkstra's algorithm and its bottleneck. In section 3, we summarize Thorup's algorithm and its implementation (including the linear-time MST algorithm in [6]) by proposing some modifications required to implement the data structures on today's computers without breaking the linear-time complexity. After that, we show a summary of results of computational experiments. In section 4, we analyze execution times and several quantities of searching phase theoretically and experimentally to clarify properties and practical efficiency of the algorithm. Some concluding remarks are in section 5.

2. Dijkstra's Algorithm and its Bottleneck

Most theoretical developments in SSSP algorithms for general directed or undirected graphs have been based on the following Dijkstra's algorithm [4].

Dijkstra's algorithm

We maintain a super distance $D(v)$ with $D(v) \geq d(v)$ for each vertex v and a set $S \subseteq V$ such that $D(v) = d(v)$ for all $v \in S$, and $D(v) = \min_{u \in S}\{d(u) + \ell(u, v)\}$ for all $v \notin S$.

```
Step 1. Initially, set $D(s) := d(s) := 0$, $S := \{s\}$ and $D(v) := \ell(s, v)$ for all $v \neq s$.
Step 2. Repeat the following (a) and (b) until $S = V$.
    (a) Find a vertex $v$ with $D(v) = \min_{u \in V \setminus S} D(u)$ and add $v$ to $S$. (We call $v$ is visited.)
    (b) For each $(v, w) \in E$ with tail $v$, if $D(w) > D(v) + \ell(v, w)$, then decrease $D(w)$
        to $D(v) + \ell(v, w)$.
```

Note that $D(v) = d(v)$ for all $v \in V$ after Step 2. Vertex v is called *visited* if v is added in S. Thus, a vertex v is already visited if and only if $v \in S$. Vertices not in S are called *unvisited*. A naive implementation of Dijkstra's algorithm takes $O(m + n^2)$ time, since we visit a vertex minimizing $D(v)$ in Step 2(a) $n - 1$ times and decrease some $D(w)$ in Step 2(b) at most m times. Note that in Dijkstra's algorithm we visit vertices in Step 2(a) in order of the increasing distances from s, and therefore, the visiting corresponds to a sorting problem in order of the increasing $d(v)$. Today, we have no linear-time algorithm for the sorting problem, and thus, we have no linear-time implementation of Dijkstra's algorithm. We call this the *bottleneck of Dijkstra's algorithm.* Note that in this paper "linear-time algorithm" means *linear-time and strongly-trans-dichotomous algorithm*, which is an algorithm based on RAM such that its time complexity is independent of the word length b of RAM, according to [6]. Thus, for example, the radix-sort is based on RAM, but not strongly-trans-dichotomous since its time complexity is $O(bn)$. Owing to the bottleneck of Dijkstra's bottleneck, unless modifying the order of visiting vertices, we cannot achieve the linear-time complexity. Thorup avoided the bottleneck by proposing a concept of *components* and using some complicated data structures based on RAM, which will be described in detail in the next section.

3. Implementation of Thorup's Linear-time Algorithm

In this section, we summarize Thorup's algorithm with several illustrations and describe the problems arising in implementing the algorithm to run on today's computers without loosing its efficiency and our modifications to cope with them.

3.1. Summary of Thorup's Algorithm

Let a given undirected graph be $G = (V, E)$ with positive integer edge weights $\ell(e)$ $(e \in E)$. We summarize Thorup's algorithm according to the following outline of the algorithm.

(a) Construct an MST $(\mathcal{M})$ in $O(m)$ time as in [6].
(b) Construct a component tree $(\mathcal{T})$ in $O(n)$ time using the MST.
(c) Compute widths of buckets $(\mathcal{B})$.
(d) Construct an interval tree $(\mathcal{U})$ in $O(n)$ time.
(e) Visit all components in $\mathcal{T}$ by using $\mathcal{B}$ and $\mathcal{U}$, in $O(m+n)$ time.

Explanations of undefined terms will be given below. (a)-(d) together correspond to the construction phase and (e) corresponds to the search phase. In the following, we first explain structures of $\mathcal{M}, \mathcal{T}, \mathcal{B}, \mathcal{U}$ and roles of them in the algorithm and then, we describe (e), the main routine of the algorithm.

3.1.1. Minimum spanning tree $\mathcal{M}$

Thorup used the linear-time MST algorithm based on RAM proposed in [6]. The algorithm contains some complicated data structures which we cannot implement with naive methods on today's computers since these data structures need a huge n more than $2^{12^{20}}$. These kinds of problems and our modifications to them are given in Section 3.2.1.

3.1.2. Component tree $\mathcal{T}$

We denote by G_i the subgraph of G whose edge set is defined to be $\{e \mid \ell(e) < 2^i, e \in E\}$. Thus, G_0 consists of singleton vertices and $G_b = G$, where b is a word length with $\ell(e) < 2^b$ for all $e \in E$. A maximal connected subgraph of G_i is called a *component* of G_i. Then the *component hierarchy* expressing a hierarchy structure of the components is defined as follows: On level i in the component hierarchy, we have the components of G_i. The component on level i containing a vertex v is denoted $[v]_i$. The children of $[v]_i$ are the components $\{[w]_{i-1} \mid [w]_i = [v]_i \text{ (i.e., } w \in [v]_i) \}$. Figure 1 illustrates an example of components of a graph.

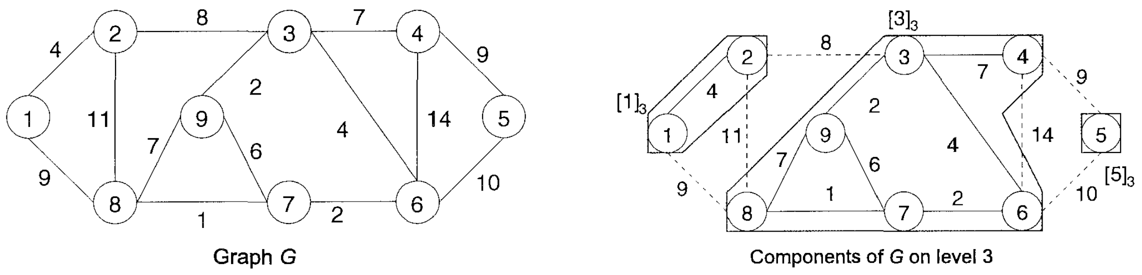


Figure 1: A graph G and its components $[1]_3$, $[3]_3$, $[5]_3$ on level 3.

The *component tree* $\mathcal{T}$ is obtained from the component hierarchy by skipping all nodes $[v]_i$ with $[v]_i = [v]_{i-1}$. Thus the leaves of $\mathcal{T}$ are the singleton components $[v]_0 = v$ $(v \in V)$ and the internal nodes are the components $[v]_i$ with $i > 0$ and $[v]_{i-1} \subset [v]_i$. The root in $\mathcal{T}$ is the node $[v]_r = G$ with r minimized. The parent of a node $[v]_i$ is its nearest ancestor distinct from $[v]_i$ in the component hierarchy. Thus, each internal node in $\mathcal{T}$ has at least two children, and therefore, the total number of components in $\mathcal{T}$ is at most $2n$. We use the component tree $\mathcal{T}$ to obtain the parent-child relations of the components in the algorithm. Figure 3.1.2 illustrates the component hierarchy and the component tree $\mathcal{T}$.

As observed by Thorup, we can construct the component tree $\mathcal{T}$ by using only edges of the MST as follows.

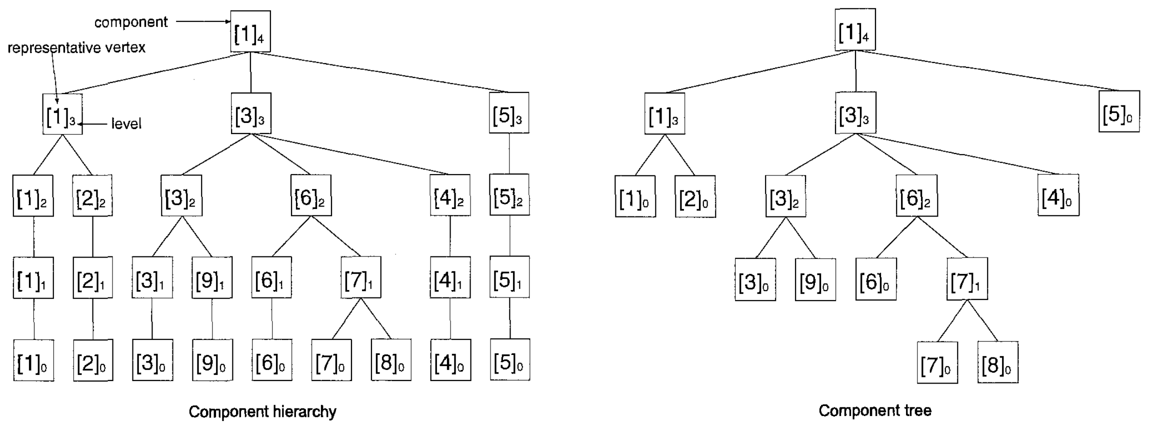


Figure 2: The component hierarchy and the component tree $\mathcal{T}$ for G in Figure 1.

Construct $\mathcal{T}$

1. Let e_{Mj} $(1 \le j \le n-1)$ be the edges of the MST $\mathcal{M}$, and let b_{Mj} be the most significant bit of $\ell(e_{Mj})$ (i.e. $b_{Mj} = \lfloor \log_2 \ell(e_{Mj}) \rfloor$, and $1 \le b_{Mj} \le b$). By using *packed merging sort* [2], make the edge set $E_i = \{e_{Mj} | b_{Mj} = i,\ 1 \le j \le n-1\}$ for $1 \le i \le b$.
2. Start from the n singleton components (of G_0).
3. For $i = 1$ to b do the following.
   (a) Make all components of G_i from the current component and E_i by using *union and find* [7], that is, if an edge $e \in E_i$ connects two components (then a new component of G_i is constructed), make a parent-child relationship in the component tree $\mathcal{T}$.

Problems and our modifications to implement *packed merging sort* and *union and find* will be given in Section 3.2.2.

3.1.3. Buckets $\mathcal{B}$

The buckets $\mathcal{B}$ consist of multi-level buckets to maintain the components according to their super distance, and are used as a technique allowing to visit a next node directly without searching on $\mathcal{T}$.

In Thorup's algorithm, we maintain the super distance $D(v)$ for all vertices as Dijkstra's algorithm, and set $D(s) = 0$ and $D(v) = \infty$ for all $v \ne s$ $(v \in V)$ at first. Note that the value of $D(v)$ will change (decrease) in the algorithm when we visit the vertex v as described in Section 3.1.4 (*change* operation), and when the algorithm terminates $D(v)$ will be equal to $d(v)$ for all $v \in V$, similarly to Dijkstra's algorithm. Let $D(v) \downarrow i = \lfloor \frac{D(v)}{2^i} \rfloor$ and $D(X) = \{D(u) \mid u \in X\}$ for $X \subseteq V$. Thus, $\min D(X) = \min_{u \in X}\{D(u)\}$. $D(v) \downarrow i$ is called a *scaled super distance* of v. We also use $D(X) \downarrow k = \{D(u) \downarrow k \mid u \in X\}$ and $\min D(X) \downarrow k = \min_{u \in X}\{D(u) \downarrow k\}$. We define *k-scaled minimum super distance* of the component $[v]_i$ (or $[v]_i^-$) as $\min D([v]_i) \downarrow k$ (or $\min D([v]_i^-) \downarrow k$, respectively).

In Thorup's algorithm, we place in the buckets of $[v]_i$, its every child component $[u]_h$, using a finite value $\min D([u]_h) \downarrow (i-1)$ as an index. For example, see Figure 3.1.3 and consider the following case: $\min D([1]_3) = 0$, $\min D([3]_3) = 19$, $\min D([5]_0) = 22$, $\min D([1]_0) = 0$, and $\min D([2]_0) = 4$, for the components in $\mathcal{T}$ in Figure 1. Then $\min D([1]_3) \downarrow (4-1) = 0$, and $\min D([3]_3) \downarrow (4-1) = \min D([5]_0) \downarrow (4-1) = 2$, and therefore, $[1]_3$ is in the 0-th bucket of $[1]_4$, $[3]_3$ and $[5]_0$ are in a 2-nd bucket of $[1]_4$. Similarly, $\min D([1]_0) \downarrow (3-1) = 0$, (and $\min D([2]_0) \downarrow (3-1) = 1$), and therefore, $[1]_0$ is in the 0-th (and $[2]_0$ is in the 1-st) bucket of $[1]_3$, respectively.

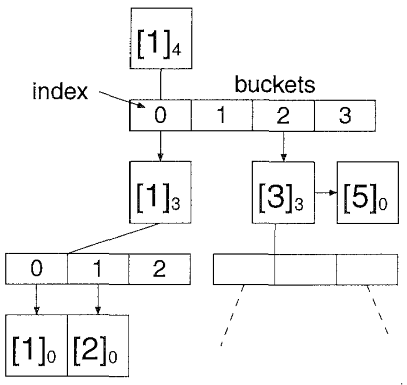


Figure 3: Illustration of the buckets $\mathcal{B}$. Parts below $[3]_3$ is omitted for simplicity.

Note that in the algorithm we place only components which have finite $\min D$ in the buckets of its parent, and therefore, at first of the algorithm, only the root component of $\mathcal{T}$ and components which contains the source s are in $\mathcal{B}$ since $D(s) = 0$ and $D(v) = \infty$ for $v \neq s$ $(v \in V)$.

To construct $\mathcal{B}$, we have only to compute the width of the buckets for each component before we place the components. The width of the buckets of $[v]_i$ is bounded by $(\max d([v]_i) - \min d([v]_i))/2^{i-1}$, and $(\max d([v]_i) - \min d([v]_i))$ is bounded by the diameter of $[v]_i$ (the *diameter* of a connected graph is the length of its longest path). We can observe that the total number of the buckets is bounded by $8n$, as observed by Thorup [10]. The fact is important for the space complexity of the algorithm.

3.1.4. Interval tree $\mathcal{U}$

$\mathcal{U}$ is called (the set of) *unvisited children* in [10], but in this paper we call it an *interval tree*, since a structure of $\mathcal{U}$ is an interval tree from the implementation point of view. The interval tree $\mathcal{U}$ maintains values $D(v)$ of the vertices and $\min D$ of unvisited components which contain visited parents to place the components correctly in $\mathcal{B}$. This is the reason that Thorup named $\mathcal{U}$ *unvisited children.*

First, we explain the structure of $\mathcal{U}$. The interval tree $\mathcal{U}$ is a data structure for only computing $\min D$ of the unvisited components efficiently, and therefore the structure of $\mathcal{U}$ is quite far from the conceptual image of $\mathcal{U}$ (in Figure 5). In $\mathcal{U}$, we use another ordering of the vertices $\{u_1, ..., u_n\}$, which is a permutation of $\{1, ..., n\}$ satisfying the following condition: (i) the root component of $\mathcal{T}$ consists of $\{u_1, ..., u_n\}$, (ii) if a component consists of a sequence $\{u_i, u_{i+1}, ..., u_{j-1}, u_j\}$ and it has k children $X_1, ..., X_k$ (i.e. X_1 is the leftmost child and X_k is the rightmost child in $\mathcal{T}$), and $|X_h| = x_h$ $(1 \leq h \leq k$, $\sum_{h=1}^{k} x_h = j - k + 1)$, then X_h consists of a sequence $\{u_a, ..., u_b\}$ where $a = \sum_{\ell=1}^{h-1} x_h + i$, $b = a + x_h - 1$. For example, for $\mathcal{T}$ in Figure 2, the root $[1]_4$ consists of $\{u_1, ..., u_9\}$, and its children $[1]_3$, $[3]_3$, $[5]_0$ consist of $\{u_1, u_2\}$, $\{u_3, ..., u_8\}$, $\{u_9\}$ respectively, since $|[1]_3| = 2$, $|[3]_3| = 6$, $|[5]_0| = 1$. We call *segment*$[a, b]$ corresponds to a component X if X consists of a sequence $\{u_a, ..., u_b\}$.

Figure 4 shows the structure of $\mathcal{U}$ for $n = 9$ (it depends on only the number n). For example, a *piece*(1) (a rectangle with 1 and $[1, 3]$) is a super vertex for vertices u_1 to u_3 , and a *piece*(2) (a rectangle with 2 and $[4, 6]$) corresponds to vertices u_4 to u_6, and a *piece*(4) (a rectangle with 4 and $[1, 1]$) corresponds to vertex u_1 respectively. At the top level, a piece corresponds to $\lfloor \log_2 n \rfloor$ vertices, and at the bottom a piece corresponds to $\lfloor \log_2 \log_2 n \rfloor$ vertices. Furthermore, a circle with $[1, 3]$ means an interval contains pieces *piece*(1) to *piece*(3), and thus it contains a sequence $\{u_1, ..., u_9\}$. We call the circle *interval*$([1, 9])$, and call a circle with $[2,2]$ ($[7,8]$) contains a $piece(2)$ ($piece(7)$ and $piece(8)$) $intervals[4,6]$ ($interval([7,8])$), respectively. Note that in this figure since n is small, a piece in the bottom of the figure corresponds a vertex.

Now we discuss the size of the interval tree, $\mathcal{U}$. At the top level of $\mathcal{U}$ we deal with $\lfloor \log n \rfloor$ vertices as one piece, so that the number of the pieces of the top level is at most $n/\lfloor \log n \rfloor$. Then at the bottom level, we construct an interval tree for each piece of the top level, and in this tree we deal with $\lfloor \log \log n \rfloor$ vertices as one piece, so that the pieces of the bottom level are at most $n/\lfloor \log \log n \rfloor$. Thus, the total number of nodes (drawn as circles in Figure 4) in $\mathcal{U}$ is bounded by $2n$ since at the top level there are at most $2n/\lfloor \log n \rfloor$ nodes and at the bottom level there are at most $2n/\lfloor \log \log n \rfloor$ nodes. Thorup used a *Q-heap* [6] to implement each bottom piece (which contains at most $\log \log n$ vertices), and we can construct $\mathcal{U}$ in $O(n)$ time if we could implement the Q-heap on today's computers, but the Q-heap is too complicated to implement, and therefore we discuss this problem in Section 3.2.3.

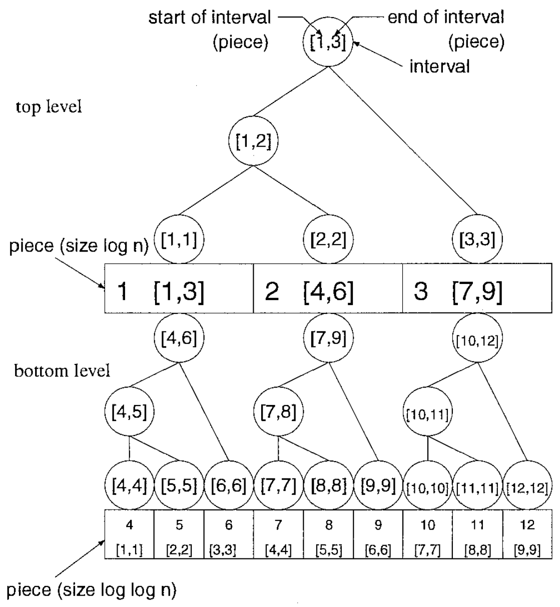


Figure 4: The interval tree $\mathcal{U}$ for $n = 9$.

The interval tree supports two operations called *change* and *split* to maintain $\min D([v]_i)$ of every components dynamically in the algorithm.

First, the *change* operation corresponds to the updates of D when a vertex is visited in Step 2(b) of Dijkstra's algorithm (thus, the total calls of *change* is at most m). When we visit a vertex v, for each vertex w adjacent to v we perform the *change*(w) operation to update $\min D([w]_i)$ where $[w]_i$ is a component in $\mathcal{U}$, and to relocate the component $[w]_i$ in the buckets of its parent.

Next, we describe the *split* operation, more difficult and complicated one. When we visit a component, components to be maintained by $\mathcal{U}$ (i.e., unvisited components having a visited parent) may change and we must recompute $\min D$ of new elements of $\mathcal{U}$ to place the unvisited components in proper buckets in $\mathcal{B}$. Figure 5 shows a conceptual image of the *split* operation. Components with asterisks in the left figure are maintained by $\mathcal{U}$ before we visit $[1]_3$. As soon as we visit $[1]_3$, the domain is changing to components with asterisks in the right figure. Then, we compute $\min D([1]_0)$ and $\min D([2]_0)$.

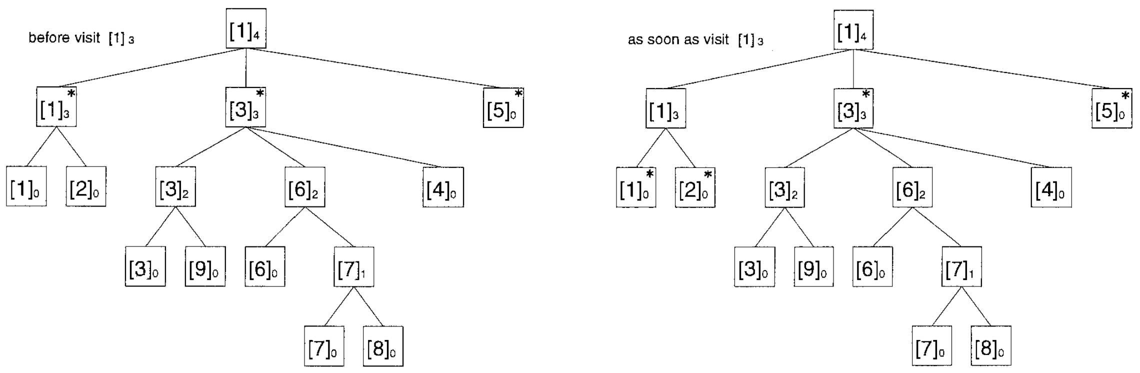


Figure 5: Illustration of the changes of the domain (i.e., *split*).

We explain an implementation of the *split* operation using the data structure of the interval tree. For the *split* operation we have to compute $\min D$ of new elements maintained by $\mathcal{U}$, by using interval tree. Since when we require to compute $\min D([v]_i)$ for some component $[v]_i$, instead of searching $D(w)$ for all vertices $w \in [v]_i$, we compute $\min D$ of some intervals. An $interval[a,b]$ is said to be *unbroken* when it is contained in some $segment[c,d]$ (i.e. $a \geq c$ and $b \leq d$) corresponding to some component maintained by $\mathcal{U}$. We can compute $\min D$ of components maintained by $\mathcal{U}$, by comparing all maximal unbroken intervals contained in segments corresponding to the components, and we can find the maximal unbroken intervals by searching down from the root. Let $\min D(interval[a,b])$ denote $\min_{a \leq j \leq b}\{D(u_j)\}$ and $\min D(piece(a))$ denote $\min_{u_j \in piece(a)}\{D(u_j)\}$. We can compute $\min D(interval[a,b])$, by computing $\min_{c \leq h \leq d} D(piece(h))$ if $piece(c)$ to $piece(d)$ are contained in the interval. Note that $\min D(piece(a))$ for $piece(a)$ in the bottom level can be obtained by a *Q-heap* in constant time though it has $\lfloor \log \log n \rfloor$ vertices.

For example, for $\mathcal{T}$ in Figure 2, at first only $[1]_4$ is maintained by $\mathcal{U}$, and a maximal unbroken interval is $interval[1,9]$, the root of the interval tree, and when we visit $[1]_4$, to place its children in the buckets, we split $[1]_4$. Then, $[1]_3$, $[3]_3$, $[5]_0$ (i.e. $segment[1,2]$, $segment[3,8]$, $segment[9,9]$ respectively) are maintained by $\mathcal{U}$ now, and by searching down the interval tree, maximal unbroken intervals are $interval[1,2]$, $interval[3,3]$, $interval[4,6]$, $interval[7,8]$, $interval[9,9]$. We compute $\min D[1]_3$ from $\min D$ of $interval[1,2]$, and $\min D[3]_3$ from $interval[3,3]$, $interval[4,6]$, $interval[7,8]$, and so on. Furthermore, when we split $[1]_3$, then $[1]_0$, $[2]_0$, $[3]_3$, $[5]_0$ are maintained by $\mathcal{U}$, and therefore maximal broken intervals become to $interval[1,1]$, $interval[2,2]$, $interval[3,3]$, $interval[4,6]$, $interval[7,8]$, $interval[9,9]$. Then we only compute $\min D$ of $interval[1,1]$ and $interval[2,2]$, and do not need to recompute the others.

3.1.5. Visit

For (e), this part is the search phase and a main routine in the algorithm. Let us call the part *visiting-part* from now on, since we visit components in the component tree. We can solve SSSP with the source s by calling **Visit**$([s]_r, r+1)$ recursively described below where r denotes the level of the root of $\mathcal{T}$.

Visit$([v]_i, j)$: We assume the parent of $[v]_i$ is $[v]_j$ $(j > i)$.

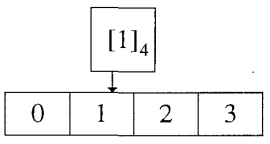
1. If $i = 0$, then for each vertex w adjacent to v, update $D(w)$ just like Dijkstra's algorithm, and perform the *change*(w) operation of $\mathcal{U}$. Then, exit the call.
2. If we have not called **Visit**$([v]_i, j)$ previously, do the following (a)-(c).
   (a) Perform the *split* operation of $\mathcal{U}$ to obtain $\min D([u]_h)$ of children $[u]_h$ of $[v]_i$.

   (b) Place the children $[u]_h$ to the bucket of $([v]_i)$ along $\min D([u]_h) \downarrow (i-1)$.
   (c) Set *index* $:= \min D([v]_i) \downarrow (i-1)$.
3. Repeat the following (a) and (b) until we have visited all vertices in $[v]_i$ or *index* $\downarrow (j-i)$ is increased:
   (a) While there are children $[w]_h$ in a bucket with $\min D([w]_h^-) =$ *index*, do the following.
      i. Let $[w]_h$ be such a child component in the bucket.
      ii. **Visit**$([w]_h, i)$ (recursive visiting).
   (b) Increase *index* by one.
4. If we have visited all vertices in $[v]_i$, delete $[v]_i$ from the bucket of the parent.
5. Otherwise, replace $[v]_i$ in a proper position of the bucket of the parent using new $\min D([v]_i^-)$.

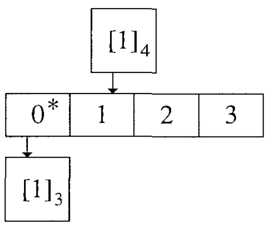
Figure 6 to 8 illustrate a beginning part of the algorithm **Visit** for G in Figure 1 and $\mathcal{T}$ in Figure 2. At first, when **Visit**$([1]_4, 5)$ is called, the buckets of $[1]_4$ are empty, and $D(1) = 0$, $D(v) = \infty$ for $2 \leq v \leq 9$ in the left part of Figure 6. Then, we perform *split* to place children of $[1]_4$ in the buckets, and therefore, $\mathcal{U}$ maintains $[1]_3$, $[3]_3$, $[5]_0$ now. Since only $[1]_3$ has a finite value $\min D([1]_3) \downarrow (4-1) = 0$, $[1]_3$ is placed in the 0-th bucket of $[1]_4$, and an index for the buckets of $[1]_4$ is set to 0. See the right part of Figure 6, the asterisk signifies the index.

Then **Visit**$([1]_3, 4)$ is called, and by *split* $\mathcal{U}$ maintains $[1]_0$, $[2]_0$, $[3]_3$, $[5]_0$ now, and $[1]_0$ is placed in the 0-th bucket of $[1]_3$, and an index for the buckets of $[1]_3$ is set to 0, as the left part of Figure 7. Then **Visit**$([1]_0, 3)$ is called, and by the *change*(1) operation and scanning edges adjacent to the vertex 1, $D(2) = 4$ and $D(8) = 9$ now. Since $[2]_0$ and $[3]_3$ in $\mathcal{U}$ have finite values $\min D([2]_0) \downarrow (3-1) = 1$, $\min D([3]_3) \downarrow (4-1) = 1$, respectively. Then since $[1]_0$ becomes empty, it is deleted from $\mathcal{B}$, and the index for the buckets $[1]_3$ increases by one, as the right part of Figure 7.

Then **Visit**$([1]_0, 3)$ is called, and by the *change*(2) operation and scanning edges adjacent to the vertex 2, $D(3) = 12$ now, but a value $\min D$ changes for no component in $\mathcal{U}$. Then since $[2]_0$ becomes empty, it is deleted from $\mathcal{B}$, and the index for the buckets $[1]_4$ increases by one, as Figure 8. We omit the reminder of the algorithm.



v	D(v)	5	Inf
1	0	6	Inf
2	Inf	7	Inf
3	Inf	8	Inf
4	Inf	9	Inf



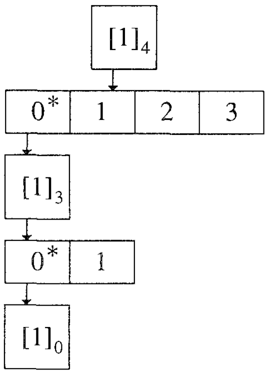
v	D(v)	5	Inf
1	0	6	Inf
2	Inf	7	Inf
3	Inf	8	Inf
4	Inf	9	Inf

Figure 6: Visit I and II

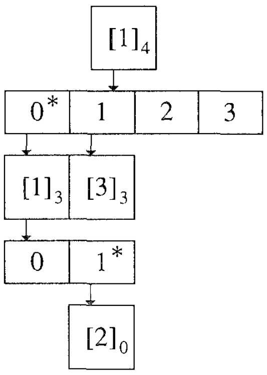
The time complexity of the algorithm is $O(n+m)$ since each call for **Visit** must determine $d(v)$ for some $v \in V$ or increase the index of the buckets by at least one, where the total number of buckets in $\mathcal{B}$ is at most $8n$, and the *split* and the *change* operation are called at most n and m times, which can be performed in $O(n)$ and $O(m)$ time respectively, as proved by Thorup.

3.2. Our modifications

We describe the problems arising in implementing it to run on today's computer, and then, we propose modifications. The main reason of difficulty in implementing Thorup's algorithm

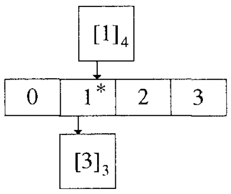


v	D(v)	5	Inf
1	0	6	Inf
2	Inf	7	Inf
3	Inf	8	Inf
4	Inf	9	Inf



v	D(v)	5	Inf
1	0	6	Inf
2	4	7	Inf
3	Inf	8	9
4	Inf	9	Inf

Figure 7: Visit III and IV



v	D(v)	5	Inf
1	0	6	Inf
2	4	7	Inf
3	12	8	9
4	Inf	9	Inf

Figure 8: Visit V

is data structures used in the linear-time MST algorithm. The data structures *atomic-heap* and *Q-heap*, which are proposed in [6], need a huge word-length.

3.2.1. Modifications of atomic-heap and Q-heap

The atomic-heap is a list of heaps like the Fibonacci heap ([5]). An array called *forest buckets* maintains the root of heaps like the root-list of the Fibonacci heap. More precisely, the forest buckets consist of 12 buckets each with size $(\log n)^{1/5}$, and therefore, the number of elements in the forestbuckets can be $12(\log n)^{1/5}$. The problem is that in [6] a Q-heap, which can have at most $(\log n)^{1/4}$ items as described below, maintains the forest buckets, thus we need $n > 2^{12^{20}}$ to satisfy $(\log n)^{1/4} \geq 12(\log n)^{1/5}$. Thus we propose that we use a Q-heap for each bucket (i.e., 12 Q-heaps for the forest buckets). After that we require a cost larger by a constant factor 12 to find the minimum element from the atomic-heap, but we are free from the limitation of n.

Q-heap is a data structure like priority-queue, and to perform all of heap operations in constant time, although its size is limited by only $\lfloor(\log n)^{1/4}\rfloor$, and moreover, it uses too complicated a table-lookup method to implement. Thus, we propose that we use an array of size 2 instead of the Q-heap since the size $\lfloor(\log n)^{1/4}\rfloor \leq \lfloor b^{1/4}\rfloor \leq 2$ where b denotes the word length and $b = 32$ or 64 on today's computers. This modification does not break the constant-time complexity on today's computers.

3.2.2. Modifications in construction of $\mathcal{T}$

We have some small problems to implement the construction of $\mathcal{T}$. First, in the construction of $\mathcal{T}$, Thorup used a *packed merging sort* proposed in [2] and [1]. It is too complicated to implement, but in this problem it is used only for $n < b$ and for other n the bucket sort is used. Therefore, we assume $n \geq b$ to avoid the packed merging sort. This causes almost no problem for most applications. Next, Thorup used a linear-time *union and find* algorithm proposed in [7] to construct $\mathcal{T}$ from $\mathcal{M}$ of $(n-1)$ edges and n vertices. It is too complicated to implement and we decide to use the union with size and the find with *path compression* instead of the algorithm, since the time complexity of the path compression method is $O(\alpha(n,n)n)$, where $\alpha(n,n)$ is the functional inverse of the Ackerman function and $\alpha(n,n) \leq 4$ for all $n \leq 2^{2^{65536}}$.

3.2.3. Modification of $\mathcal{U}$

The remaining problem to implement the algorithm is the interval tree $\mathcal{U}$. The interval tree $\mathcal{U}$ is realized as a two-levels tree in [10]. The problem is that we must maintain the bottom super vertex with a Q-heap in [10] to perform all operations of heap in constant time. Recall that a Q-heap can maintain only at most $(\log n)^{1/4}$ elements (in our methods, the upper bound of the size is equal to 2 as above), and that it needs $(\log n)^{1/4} \geq \log\log n$. Since this inequality is satisfied only when $n > 2^{2^{16}}$, we cannot implement the interval tree with a naive method on today's computers.

To cope with this problem, we propose that we implement the interval tree as a three-levels tree. Under the second level of the tree, we attach a third level similarly in which each super vertex is for $\lfloor \log\log\log n \rfloor$ vertices. The Q-heaps maintain the super vertices in the third level only. Then, since $\lfloor \log\log\log n \rfloor \leq 2$ for $n \leq 2^{64}$, we can maintain the super vertex by a Q-heap with size 2 on today's computers.

3.3. Summary of experimental results for the whole algorithms

By using these modifications we are ready to implement Thorup's linear-time algorithm. In fact, we have actually implemented the algorithm together with the supported data structures and measured execution times of the implementation. We also implemented old algorithms based on Dijkstra's algorithm to compare the practical efficiencies. We note that the program (written with C++) is over 3,000 lines (c.f. naive Dijkstra program is only 100 lines), and this complexity of the program does not make merits in practice with comparing other implementations.

Here we only summarize results of the above experiments. For the whole execution time for SSSP (instead of search time), Thorup's algorithm take more time than the algorithm based on the Fibonacci heap in [5] (i.e., Dijkstra's algorithm with the Fibonacci heap which has time complexity $O(m + n\log n)$ and is the theoretically fastest algorithm based on the comparison computation model today). In particular, the part of the MST algorithm occupies most of the execution time of Thorup's algorithm. Table 1 shows the running times of Thorup's algorithm and Dijkstra's algorithm (naive one and one with Fibonacci heap) on random graphs $n = 50000$, $m = 175065, 299914, 424396$, respectively, and rows **MST, Data structures, Visit** show times to construct the MST $\mathcal{M}$, the other data structures $(\mathcal{T}, \mathcal{B}, \mathcal{U})$, and times for **Visit**$([s]_r, r+1)$, respectively. For experiments, we use Sun Ultra 30 workstation with 1GB main memory. Our programs are written in C++.

Table 1: The running times (sec) of Thorup's algorithm and Dijkstra's algorithm (naive and Fibonacci heap versions), $n = 50000$

m	175065	299914	424396
Dijkstra(naive)	327.53	326.63	326.97
Dijkstra(F-heap)	2.33	2.83	3.26
Our Thorup(total)	25.89	40.74	57.43
MST	23.33	35.24	56.60
Data structures	1.25	1.30	1.23
Visit	1.96	2.46	2.91

For space, we require at most $56n$ words for $\mathcal{T}$ and at most $32n$ for the buckets and at most $34n$ words for $\mathcal{U}$ in our implementation. Thus, our implementation requires at most $122n$ words (except the space for the graph and the MST). Note that we do not discuss memories for the MST, since the MST can be free after the construction of the MST.

The results above concern with the single query mode problem. Thus, it is too early to say that Thorup's algorithm is not useful in practical applications since it can be used as the shortest path search problem. In the next section, we analyze the visiting-part (search phase) and clarify the possibility of the usefulness of Thorup's algorithm .

4. Analysis of the Visiting-part

The visiting-part is important for practical use since only the visiting-part is repeated for each query source in the shortest path search problem. If the execution time of the visiting-part is less than the whole execution times of other algorithms based on Dijkstra's one, our implementation based on Thorup's algorithm will be useful for the shortest path search problem. In this section we show the details of the visiting-part and analyze it theoretically and experimentally.

We can divide the visiting-part into the 3-parts as follows. Refer section 3 for the algorithm of the visiting-part.

1. Main routine of visiting (Visiting-main or VM for short)
   Scanning the buckets of the component from left to right, we visit the children components in the buckets (we call the procedure Visiting-main-body or VMBD for short). Then, if necessary, we replace the component in the appropriate bucket of its parent as described in page 9, step 5 of **Visit** algorithm (we call the procedure Visiting-main-bucketing or VMBC for short). Quantities concerning the procedure VM are the total number of the buckets (at most $8n$) and the number of the components (at most $2n$). Note that the number of calls of the procedure depends on the order of visiting (and also depends on the source).
2. Visiting the leaves (Visiting-leaves or VL for short)
   When visiting a vertex (leaf of $\mathcal{T}$), we scan the vertices adjacent to the vertex and changes D of the vertices properly as in Dijkstra's algorithm (Visiting-leaves-Dijkstra or VLD for short). The total calls of the procedure is always n, and the total number of the changing D is bounded by m. Concerning the changing D, we perform the operations *change* of $\mathcal{U}$ (Visiting-leaves-change or VLC for short), and then, if necessary, we replace components which have updated $\min D$ in buckets of their parents (Visiting-leaves-bucket or VLB for short).
3. First bucketing for each component (Expand or EX for short)
   When we visit a component for the first time, we place its children components to its buckets (Expand-bucket or EXB for short). At this time, we perform the operation *split* of $\mathcal{U}$ (Expand-split or EXS for short).
   The total calls of the procedure and *split* is equal to a difference between the number of the components and the number of the leaves in $\mathcal{T}$, that is, (the number of the components - n). Note that the number of the elements of $\mathcal{U}$ (*intervals, pieces, mui* in Figure 15) concerns the total execution time of the procedure. For details of the elements of $\mathcal{U}$, see [10].

Note that the execution time of the whole of the visiting-part (VW for short) is equal to the sum of the execution times of VM, VL, and EX.

We measure the execution times and the number of calls of the three parts and the sub procedures and the numbers of the elements by several experiments. In addition, we compare the execution times of the visiting-part with the execution times of Dijkstra's algorithm with the Fibonacci heap (F-heap, for short) and the algorithm with the binary heap (B-heap, for short). We note that Dijkstra's algorithm with the binary heap proposed in [11] is very popular in practice and it has $O(m \log n)$ time complexity for the worst case. Refer [3], [8], and [9] to acquire the efficiency of the binary heap comparing with other algorithms and data structures.

We use five random graphs and five $k \times k$ grid graphs in the experiments. We make the random graphs for given n ($V = \{1, ..., n\}$) and m as follows (in the following experiments, we fix $m = 5n$).

1. We assume that there are edges $(i, i+1)$ for all $i = 1, ..., n-1$. The assumption is for connectivity, and it does not lose the generality.
2. For the remaining $m - n + 1$ edges, there are $n(n-1)/2 - (n-1) = (n-1)(n-2)/2$ combinations (i, j). Let LE denotes a number of remaining edges and LV denotes a number of remaining combinations.
3. For $(1, 3)$ to $(n-2, n)$, we give a possibility LE/LV for an edge.

We use uniform random numbers from 1 to 100 for the weights of the edges of the random graphs and the grid graphs. We note that the graphs are sparse ($m = O(n)$) since if $m = O(n^2)$ the time complexities of any algorithms are equal and if $m = O(n \log n)$ Thorup's algorithm and Dijkstra's algorithm with the Fibonacci heap have the same time complexity.

The results obtained by the experiments are given below with the boxplots. In the figures, the left graphs correspond to the grid graphs and the right graphs correspond to the random graphs.

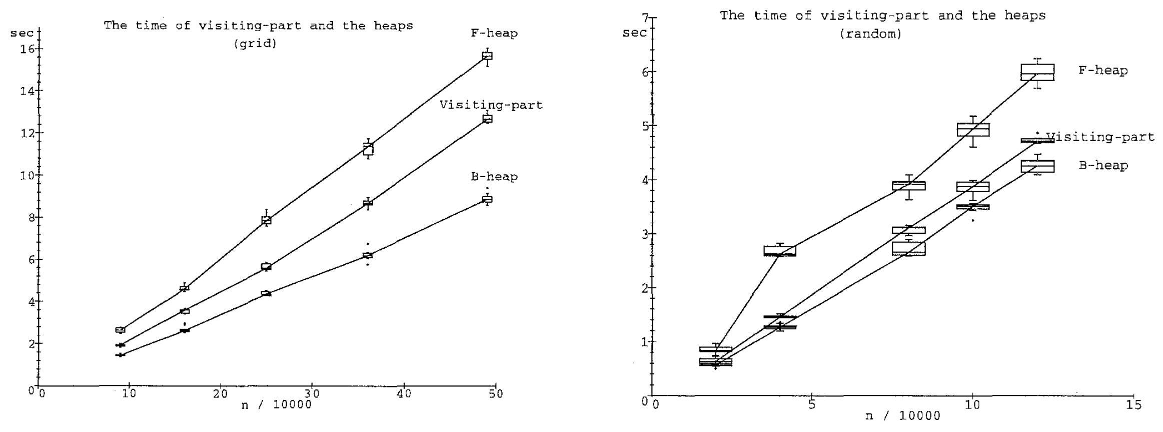


Figure 9: The execution times of the visiting-part, the Fibonacci heap algorithm (F-heap) and the binary heap algorithm (B-heap).

From the experiments, we obtain the following observations.

- Visiting-part is as fast as the heaps (faster than F-heap and slower than B-heap). Note that the time complexity B-heap is now $O(n \log n)$ since we use $m = O(n)$. The times of them are roughly proportional to n in the experiments. More precisely, in the grid graphs, the execution time of the visiting-part is less than $0.30 \times 10^{-4} n$ (note that $m = 2n$) and the time of B-heap is more than $0.0090 \times 10^{-4} n \log n$, and thus, the visiting-part is faster than B-heap only when $\log n > 0.30/0.0090$, that is, roughly $n > 2^{34}$. In the random graphs, $m = 6n$, the execution time of the visiting-part is less

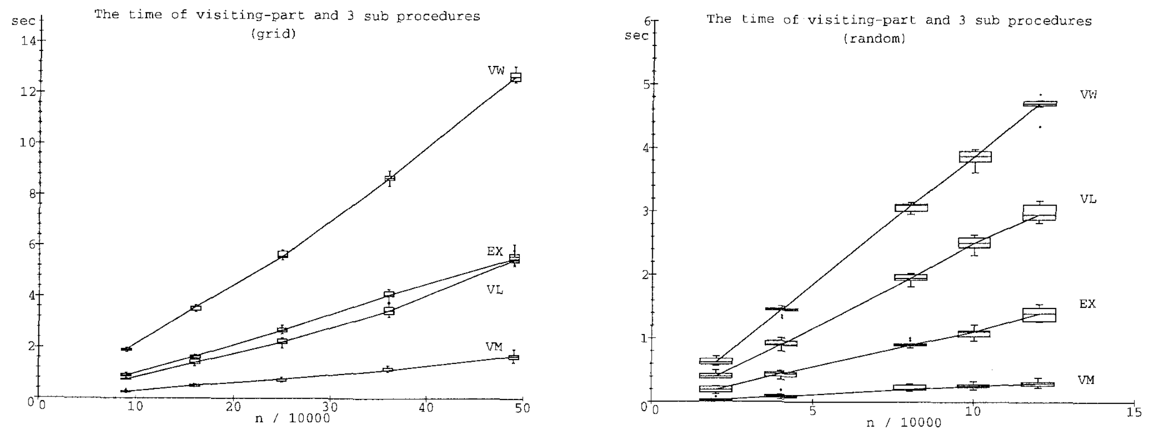


Figure 10: The execution times of the visiting-part (VW) and its 3 parts procedures VM, VL, EX.

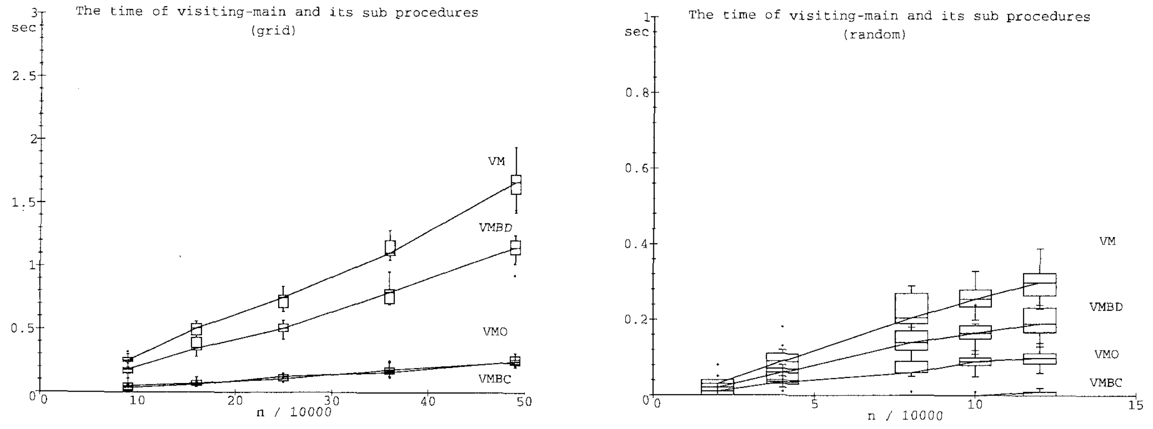


Figure 11: The execution times of Visiting-main (VM) and its sub procedures (VMBD, VMBC described as above, and VMO denotes the other remaining procedures in VM except VMBD and VMBC, that is, Visiting-main-others).

than $0.45 \times 10^{-4}n$ and the time of B-heap is more than $0.018 \times 10^{-4}n \log n$, and thus, the visiting-part is faster than B-heap only when $\log n > 0.45/0.018$, that is, $n > 2^{25}$. Note that in dense graphs B-heap has less advantages than in sparse graphs. As a result, on today's computers, the visiting-part is not faster than B-heap. Note that our implementation of Thorup's algorithm needs more than $100n$ words of memories as described in previous section.

- VL (Visiting-leaves) and EX (Expand) occupy most of the time in the visiting-part, though VM (Visiting-main) takes small time (figure 10).
- VLD (Visiting-leaves-Dijkstra) and VLC (Visiting-leaves-change) occupy most of the time in VL, and VLD takes more costs than VLC (figure 12).
- EXS (Expand-split) occupies most of the time in EX (figure 13).
- As a result, VLD, VLC, and EXS occupy most of the time in the visiting-part. Note that VLC and EXS are the operations of the interval tree $\mathcal{U}$, and VLD is the same operation as update D in Dijkstra's algorithm, which implies we can hardly reduce the times of VLD.
- The total numbers of calls of all the procedures are proportional to n (figure 14) since we fix $m = O(n)$ for the experiments. This result confirms the theory. Note that calls

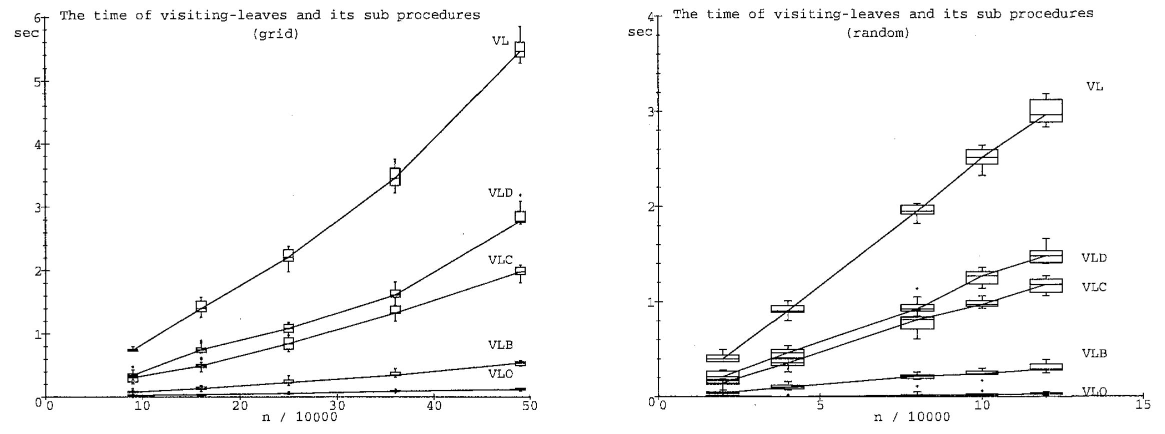


Figure 12: The execution times of Visiting-leaves (VL) and its sub procedures VLD, VLC, VLB, and Visiting-leaves-others (VLO) similar to VMO.

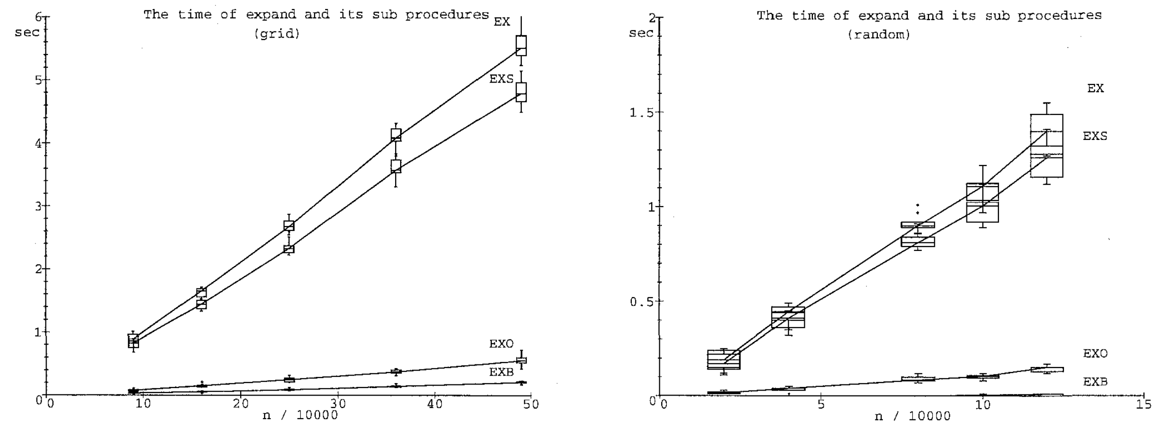


Figure 13: The execution times of Expand (EX) and its sub procedures EXS, EXB, and Expand-others (EXO).

of VM is more than calls of VLC in the grid graphs, on the other hand, in the random graphs calls of VM is far less than calls of VLC, since calls of *changes* increases when m is larger.

- The number of all the elements are proportional to n (figure 15). This result confirms the theory.

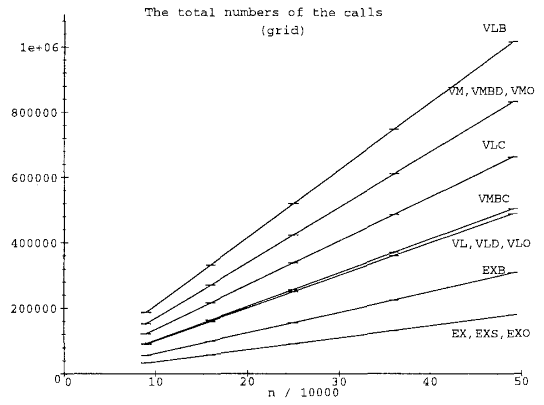
Finally, we discuss the results and future works in next section.

5. Concluding Remarks

In conclusion, our implementation of Thorup's linear-time algorithm is very slow for SSSP due to the time of the construction of the data structures. On the other hand, the *visiting-part*, which is repeated in the shortest path search problem, is as fast as the algorithms using the heaps.

Thus, due to the huge amount of memories and the complicated and large programs, our implementation of Thorup's linear-time algorithm is not useful in practice today.

However, we should make clear which parts in the visiting-part are slow for more improvements in the future. We note that the bottlenecks in the visiting-part are Visiting-leaves and Expand parts, and therefore, if we can improve the parts, we can acquire a faster implementation. In particular, we can modify the implementation of the interval tree $\mathcal{U}$ to



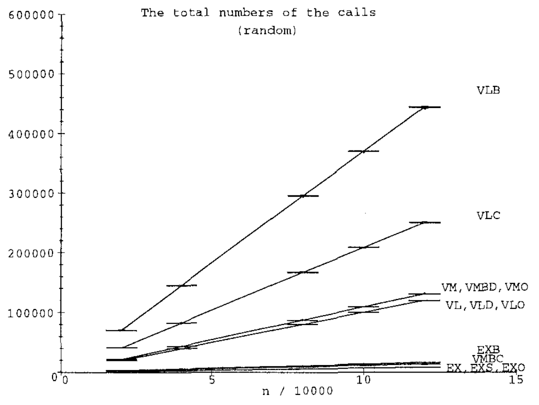
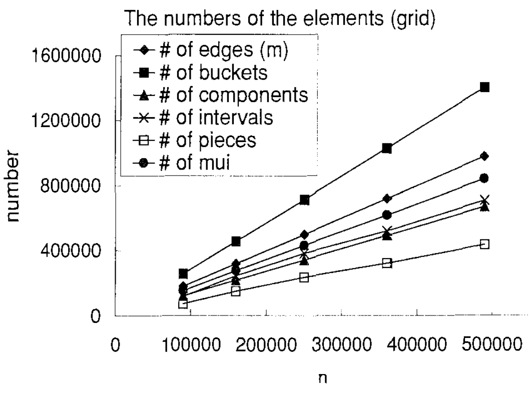


Figure 14: The total numbers of the calls. Note that the number of VM is equal to the ones of VMBD and VMO, similarly, VL is equal to VLD and VLO, EX is equal to EXS and EXO. Further, VL is equal to n.



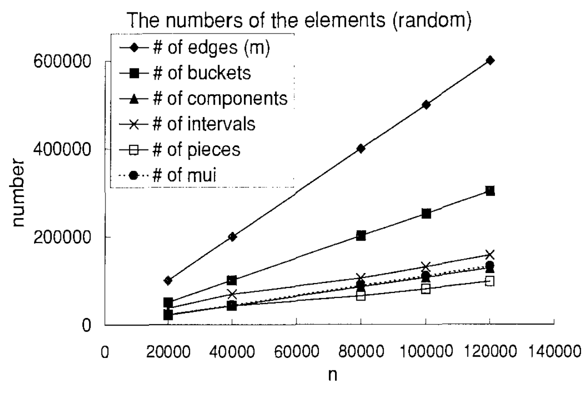


Figure 15: The numbers of the elements. Note that intervals, pieces (correspond to the super vertices), and mui (maximal unbroken intervals which belong to the components to find the minimum D by scanning, refer [10] for a detail) are the elements of the interval tree $\mathcal{U}$.

improve Expand part (i.e., *split*), for example, changing the structure. In this paper, we use the *three-levels* interval trees and its super vertices have sizes $\log n$, $\log\log n$, and 2, but we can implement it with other methods changing the number of levels and the sizes of the super vertices. Moreover, we will be able to reduce the amount of memories by using some minute techniques, in particular, in $\mathcal{T}$ and $\mathcal{U}$.

Finally, we describe the future works.

- Can we improve the implementation of Thorup's algorithm for practical use?
- Can we use the idea that we divide the algorithm into the *preprocess* constructions of the data structures and the visiting-part, to make a more efficient algorithm?
- Can we apply the data structures in [10] to other problems?

References

[1] S. Albers and T. Hagerup: Improved parallel integer sorting without concurrent writing. *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms*, (1992) 463–472.

[2] A. Andersson, T. Hagerup, S. Nilsson and R. Raman: Sorting in linear time? *Proceed-*

ings of 27th ACM Symposium on Theory of Computing, (1995) 427–436.

[3] B. V. Cherkassky, A. V. Goldberg and C. Silverstein: Buckets, heaps, lists, and monotone priority queues. *Proceedings of 8th ACM-SIAM Symposium on Discrete Algorithms*, (1997) 83–92.

[4] E. W. Dijkstra: A note on two problems in connection with graphs. *Numerische Mathematik*, **1** (1959) 269–271.

[5] M. L. Fredman and R. E. Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, **34** (1987) 596–615.

[6] M. L. Fredman and D. E. Willard: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, **48** (1994) 533–551.

[7] H. N. Gabow and R. E. Tarjan: A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, **30** (1985) 209–221.

[8] A. V. Goldberg and C. Silverstein: Implementations of Dijkstra's algorithm based on multi-level buckets. *Technical Report* **95-187** (NEC Research Institute, Prinston, NJ, 1995).

[9] H. Imai and M. Iri: Practical effeciencies of existing shortest-path algorithms and a new bucket algorithm. *Journal of the Operation Research Society in Japan*, **43** (1984) 43–58.

[10] M. Thorup: Undirected single source shortest paths in linear time. *Proceedings of the 38th Symposium on Foundations of Computer Science*, (1997) 12–21.

[11] J. Williams: Heapsort. *Communications of the ACM*, **7** (1964) 347–348.

Yasuhito Asano
Graduate School of Information Science
The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo 113-0033, Japan
E-mail: `asano@is.s.u-tokyo.ac.jp`