アルゴリズム 84－2
（2002．5．23）

有向グラフの推移閉包問題に対する Fully Dynamic アルゴリズムの再定式化と実装

伊藤 剛志　今井 浩
東京大学大学院情報理工学系研究科コンピュータ科学専攻

本論文では有向グラフの推移閉包問題に対する Demetrescu と Italiano の dynamic アルゴリズムについて考察する。このアルゴリズムではグラフが少しずつ更新されるときその反射推移閉包を更新のたびに一から再計算することなく求めることができる。本論文ではこのアルゴリズムの仕組みを明確にし、もとの形より簡略化したデータ構造を示す。また、この簡略化版のアルゴリズムを実装して実行時間を static アルゴリズムである Floyd-Warshall アルゴリズムと比較することにより性能を評価する。更新の回数に対する反射推移閉包の問い合わせの頻度が高いときは、頂点数が 512 程度というあまり大きくないグラフでもこの dynamic アルゴリズムが有効であることを示す。

Reformalization and Implementation of Fully Dynamic Algorithm for Transitive Closure on Directed Graphs

ITO Tsuyoshi　IMAI Hiroshi
Department of Computer Science,
Graduate School of Information Science and Technology, University of Tokyo

In this paper, Demetrescu and Italiano's dynamic algorithm for the transitive closure problem on a directed graph is considered. This algorithm computes the reflective transitive closure of a graph when the graph is updated repeatedly, without recomputing it from scratch every time the graph is updated. In this paper, the mechanism of the algorithm is clarified and a data structure simpler than the original one is presented. In addition, this simplified version is implemented to see its efficiency by comparing the execution time of it to that of static Floyd-Warshall algorithm. When many queries on the reflective transitive closure is performed compared to updates, the dynamic algorithm is efficient even for a graph with 512 vertices, which is not impractically large.

1 Introduction

In this paper, we consider the fully dynamic problem of computing the reflective transitive closure of a given directed graph. A *fully dynamic graph problem* is a problem to find an efficient way to represent a graph while edges are added to or removed from it repeatedly so that a certain property of it can be computed in a short time when asked.

In fact, a dynamic problem is not necessarily a problem on a graph. Generally, a dynamic algorithm for a problem is an online algorithm which accepts a sequence of the three kinds of operations:

- Initialization: a whole input to the original problem.
- Update: a change to the input to the original problem.
- Query: a request for the answer to the original problem.

A fully dynamic algorithm for a graph problem is a dynamic algorithm which supports both the addition and the removal of an edge as an update. A dynamic

graph algorithm which supports only either of the addtion or the removal of an edge is called an *incremental* or *decremental* algorithm, respectively.

We can make a trivial dynamic algorithm for a problem from any *static algorithm* (that is, an algorithm in a common sense which is not dynamic) for that problem in two ways. One way to do this is to store only the input to the original problem; we update this input according to the initializations and updates requested and use the static algorithm every time a query is given. This way, an initialization and an update are fast and a query is slow. We call this algorithm a *trivial dynamic algorithm with fast updates*. The other way is to store both the input and the answer to the original problem; when an initialization or an update is requested, we update not only the input but also the answer by using the static algorithm. This way, an initialization and an update are slow and a query is fast. We call this algorithm a *trivial dynamic algorithm with fast queries*. We are interested in a dynamic algorithm which can perform both an update and a query in a short time.

The time complexity of an online algorithm is often described in terms of the *amortized time complexity* [10], or the amount of time an operation requires for the whole sequence of operation to be completed. This is because we are mainly interested in its performance on a sequence of operations, not on a single operation.

The *reflective transitive closure* of a directed graph $G = (V, E)$ is a directed graph $G^* = (V, E^*)$ where for any two vertices s and t, $(s, t) \in E^*$ if and only if G has a directed path from s to t. We call the problem of computing the reflective transitive closure simply the *transitive closure problem*, or *TC*. TC is one of the fundamental problems on a directed graph and has application in many fields. TC can be seen as a special version of the shortest path problem where the weight of every edge is equal to 0 or ∞.

The static algorithm by Munro [7] computes the reflective transitive closure of a given graph in $O(n^\omega + n^2)$ worst-case time, where n is the number of vertices and ω is the exponent of matrix multiplication over a ring (currently $\omega < 2.38$ [1]). At the top of our knowledge, Munro's algorithm is the fastest static algorithm currently known when the worst-case complexity expressed as a function of n is concerned. On the other hand, many algorithms with $O(nm + n)$ worst-case time have been proposed, where m is the number of the edges in the graph. Among them is [8], which contains references to many of them.

Though many dynamic algorithms for graph problems have been found since the dynamic tree algorithm by Sleator and Tarjan [9], there was no fully dynamic algorithm for TC or any kind of shortest path problems on a directed graph until Henzinger and King [5] found a randomized algorithm for TC with one-side error.

Three deterministic fully dynamic algorithms are now currently known for TC. The first deterministic fully dynamic algorithm for TC was devised by King [6]. King's algorithm uses a forest of partial depth-first-search trees and supports updates in $O(n^2 \log n)$ amortized time and queries in $O(1)$ worst-case time. An update in King's algorithm is called a *generalized update* since it supports as one update the addition of arbitrary number of edges originating from or terminating at a vertex, or the removal of arbitrary number of edges in the graph. Both of the other two algorithms were proposed by Demetrescu and Italiano [2, 3]. These two algorithms use the reduction to the multiplication of matrices and supports queries in $O(1)$ worst-case time. One of them takes $O(n^2 \log n)$ time for an update and the other takes $O(n^2)$ time. An update in Demetrescu and Italiano's algorithms is also a generalized update, the same operation as that in King's algorithm.

Demetrescu and Italiano's algorithms use a well-known relation between the reflective transitive closure of a graph and the matrix multiplication: the adjancency matrix of the reflective transitive closure G^* of G is equal to the *Kleene closure* of the adjancency matrix of G over the Boolean semiring.[*1] The Kleene closure X^* of an $n \times n$ Boolean matrix X is defined as

$$X^* = \sum_{i=0}^{\infty} X^i$$

and it satisfies

$$X^* = (X + I)^{n'}, \quad (^\forall n' \geqq n)$$

where I is the identity matrix.

In their papers, Demetrescu and Italiano first defined the *dynamic matrix multiplication problem*, then presented an efficient algorithm for this new problem, and finally described their fully dynamic TC algorithms, which use their dynamic matrix multiplication algorithm as a common subroutine.

It is usually a good idea to divide a problem into several easier problems. The right way of division

[*1] The semiring $(\{0, 1\}, +, \cdot)$ which satisfies

$$0 + 0 = 0,\ 0 + 1 = 1 + 0 = 1 + 1 = 1;$$
$$0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0,\ 1 \cdot 1 = 1$$

is called the *Boolean semiring* and denoted $\mathbb{B}$. A matrix over the Boolean semiring is called a Boolean matrix, and the semiring of all the $n \times n$ Boolean matrices is denoted $\mathrm{M}_n(\mathbb{B})$.

makes it easier to prove the correctness and complexity of the algorithm, to implement it, to improve efficiency in both theory and practice, and to reuse it in a different problem. Unfortunately, Demetrescu and Italiano's definition of the dynamic matrix multiplication problem does not seem to be specific enough to have these benefits of the division of the problem. We redefine the dynamic matrix multiplication problem in this paper.

The time complexity of Demetrescu and Italiano's dynamic TC algorithm suggests that it is faster than any static one currently known if only a small number of updates are requested between two adjacent queries. However, the time complexity tells only the asymptotic behavior of the algorithm with large input.

We implement Demetrescu and Italiano's algorithm and carry out the experiments on its performance to know the real behavior of the algorithm.

The rest of this paper is organized as follows. We first define the dynamic TC problem in section 2 and the dynamic matrix multiplication problem in section 3. Then we review Demetrescu and Italiano's first dynamic TC algorithm which uses the dynamic matrix multiplication algorithm as a black box, and show how our division clarifies the proof of its correctness and complexity in section 4. After that, we give a simple algorithm for the dynamic matrix multiplication problem in section 5. Section 6 shows the result of the experiment on the Demetrescu and Italiano's dynamic TC algorithm. Finally, section 7 summarises and concludes the whole paper.

2 Dynamic TC Problem

The fully dynamic TC problem is formalized as follows. The algorithm is to hold a graph $G = (V, E)$ internally. The set V of the vertices is given first and does not change during the operations. The problem is to perform the following operations.

- Initialization
  - `Init`(G') where $G' = (V, E')$ is a graph with n vertices:
    initializes G to the given graph G'.
- Updates
  - `Add`(v, S, T) where $v \in V$ and $S, T \subseteq V$:
    adds the edges (s, v) for $s \in S$ and (v, t) for $t \in T$ to G if G does not contain such edges. v is called the *center* of this update.
  - `Remove`(s, t) where $s, t \in V$:
    removes the edge (s, t) if G contains it.
- Query
  - `Reachable`(s, t) where $s, t \in V$:
    returns whether a directed path from s to t exists in G.

The operations are specified in an arbitrary order except the first operation must be an `Init`.

This formalization is different from that by Demetrescu and Italiano: we define `Remove` as it removes one edge at a time while Demetrescu and Italiano defined an operation which removes arbitrary number of edges. The fact that our operation takes $O(\log n)$ amortized time corresponds to the fact that Demetrescu and Italiano's takes $O(n^2 \log n)$ amortized time. This difference is for simplicity of the description of the problem and for ease of the complexity analysis of the algorithm.

3 Dynamic Matrix Multiplication Problem

Dynamic Boolean matrix multiplication problem is the one to find a data structure and algorithm which keeps track of $2h$ matrices $X_1^{(1)}, X_2^{(1)}, X_1^{(2)}, X_2^{(2)}, \ldots, X_1^{(h)}, X_2^{(h)} \in \mathrm{M}_n(\mathbb{B})$ to represent the *polynomial*

$$P = X_1^{(1)} X_2^{(1)} + X_1^{(2)} X_2^{(2)} + \cdots + X_1^{(h)} X_2^{(h)}$$

while a sequence of the operations are performed. Each $T^{(a)} = X_1^{(a)} X_2^{(a)}$ is called a *term* and each $X_b^{(a)}$ a *variable*.[*2] This algorithm is called a dynamic Boolean multiplication algorithm *representing* the polynomoial P.

The (s, t)-element $T^{(a)}[s, t]$ in $T^{(a)}$ shows whether or not any pair of elements $X_1^{(a)}[s, x]$ and $X_2^{(a)}[x, t]$ satisfies $X_1^{(a)}[s, x] = X_2^{(a)}[x, t] = 1$. We call the pair of elements $X_1^{(a)}[s, x]$ and $X_2^{(a)}[x, t]$ simply the *pair* $(a; s, x, t)$. The pair $(a; s, x, t)$ is said to be *incident to* the elements $X_1^{(a)}[s, x]$ and $X_2^{(a)}[x, t]$, and it is said to be *permitted* when $X_1^{(a)}[s, x] = X_2^{(a)}[x, t] = 1$.

Pairs have two states: *valid* or *invalid*. The state of pairs changes in response to update operations.

[*2] We use the terms *polynomial*, *term* and *variable* according to Demetrescu and Italiano's paper, though it is different from the common usage of the terms because the order of the variables in each term is significant in our polynomials.

The element $T^{(a)}[s,t]$ is said to be *valid* when a pair $(a;s,x,t)$ is valid for some $1 \leqq x \leqq n$. Otherwise, it is said to be *invalid*.

The *tracked value* of the polynomial P is a Boolean matrix P' such that $P'[s,t] = 1$ iff for some a, the element $T^{(a)}[s,t]$ is valid.

The operations permitted in the dynamic Boolean matrix multiplication problem are:

- `Init`$(X_1'^{(1)}, X_2'^{(1)}, X_1'^{(2)}, X_2'^{(2)}, \ldots, X_1'^{(h)}, X_2'^{(h)})$ where $X_b'^{(a)} \in \mathrm{M}_n(\mathbb{B})$ $({}^\forall X_b'^{(a)})$:
  initializes each variable $X_b^{(a)}$ according to $X_b'^{(a)}$. In addition, it validates all the permitted pairs and invalidates the others.

- `LazySet`(i,j,a,b) where $1 \leqq i,j \leqq n$, $1 \leqq a \leqq h$ and $1 \leqq b \leqq 2$:
  changes the value of $X_b^{(a)}[i,j]$ to 1. It does not change the states of the pairs.

- `Reset`(i,j,a,b) where $1 \leqq i,j \leqq n$, $1 \leqq a \leqq h$ and $1 \leqq b \leqq 2$:
  changes the value of $X_b^{(a)}[i,j]$ to 0, invalidates all the pairs incident to $X_b^{(a)}[i,j]$, and returns the list of (s,t) such that $P'[s,t] = 1$ before the update but $P'[s,t] = 0$ after the update.

- `FixRow`(s,a) where $1 \leqq s \leqq n$, $1 \leqq a \leqq h$:
  validates all the permitted pairs $(a;s,x,t)$ for $1 \leqq x,t \leqq n$.

- `FixMid`(x,a) where $1 \leqq x \leqq n$, $1 \leqq a \leqq h$:
  validates all the permitted pairs $(a;s,x,t)$ for $1 \leqq s,t \leqq n$.

- `FixCol`(t,a) where $1 \leqq t \leqq n$, $1 \leqq a \leqq h$:
  validates all the permitted pairs $(a;s,x,t)$ for $1 \leqq s,x \leqq n$.

- `Lookup`(s,t) where $1 \leqq s,t \leqq n$:
  returns the value of $P'[s,t]$.

Any sequence of these operations is permitted except that the first operation must be an `Init`.

Note that these operations are designed to maintain the following invariant: *after each operation, a valid pair is always permitted*. This is why we use the term *permitted*.

This invariant and the definition of the tracked value of a polynomial give the following lemma.

Lemma 3.1. *Let P' be the tracked value of P. After each operation, it holds $P' \subseteq P$.*[*3]

[*3]For two Boolean matrices X and Y, $X \subseteq Y$ means that $X[i,j] = 1$ implies $Y[i,j] = 1$.

Note that when all the permitted pairs are valid, it holds $P = P'$. This means `Lookup` returns the value of an element of P just after an `Init`.

The algorithm for this problem is described in section 5. We only summarize the result here. In terms of the worst-case time complexity, an `Init` takes $O(hn^\omega + hn^2)$ time. Each of `LazySet` and `Lookup` takes $O(1)$ time, and each of `Reset`, `FixRow` and `FixCol` takes $O(n^2)$ time in the worst case. In terms of the amortized time complexities, an `Init` takes $O(hn^3)$ time, each of `LazySet`, `Reset` and `Lookup` takes $O(1)$ time, and each of `FixRow` and `FixCol` takes $O(n^2)$ time. The algorithm requires $O(hn^2)$ space.

4 Dynamic TC Algorithm

In this section, a reformalized and simplified version of Demetrescu and Italiano's TC algorithm with $O(n^2 \log n)$ update time is presented.

Let G be the graph which the algorithm represents, and X be the adjacency matrix of G over $\mathbb{B}$. In addition, Let $P_0 = X + I$ where I is the identity matrix, that is, P_0 is equal to 1 at the diagonal elements and equal to X at the other elements.

Let $l = \lceil \log_2 n \rceil$. The data structure for this algorithm consists of l pairs of polynomials $Q_1, P_1; Q_2, P_2; \ldots; Q_l, P_l$. Each Q_k has one term and each P_k has two terms, thus they can be expressed as

$$Q_k = A_k B_k \tag{1}$$
$$P_k = L_k C_k + D_k R_k. \tag{2}$$

We denote the tracked values of Q_k and P_k by Q_k' and P_k' respectively. For easy notation, let $P_0' = P_0$.

Though many variables are used in the equations (1) and (2), things are not as complicated as it appears because it holds after every operation that

$$A_k = B_k = C_k = D_k = P_{k-1}', \quad L_k = R_k,$$

for $1 \leqq k \leqq l$.

Now we describe how each operation works.

`Init`(G') sets all the variables in the polynomials so that the equations

$$A_k = B_k = C_k = D_k = P_{k-1}',$$
$$L_k = R_k = Q_k'$$

are satisfied for $1 \leqq k \leqq l$, by calling `Init` on Q_1, $P_1, Q_2, P_2, \ldots, Q_l$ and P_l in this order.

`Add`(v,S,T) first sets the (s,v)-elements and the (v,t)-elements in A_1, B_1, C_1 and D_1 to 1 for all $s \in S$ and $t \in T$ by calling `LazySet` on Q_1 and P_1. Then

it calls FixRow and FixCol on the vth row and the vth column in Q_1. Next, in order to propagate the changes in Q_1 to L_1 and R_1, for each of those elements on the vth row and the vth column in Q_1 on which Lookup returns 1, it sets the corresponding elements in L_1 and R_1 to 1 by calling LazySet on P_1, and calls FixMid on both terms in P_1. Then it propagates the changes in P_1 to A_2, B_2, C_2 and D_2 by calling LazySet on Q_2 and P_2 for each element in P_1 on which Lookup returns 1, and calls FixRow and FixCol on the vth row and the vth column in Q_2. The following steps are the propagation of the changes in Q_2 to L_2 and R_2, the propagation of those in P_2 to A_3, B_3, C_3 and D_3 and so on, until those in Q_l are propagated to P_l. These steps are similar to the previous steps.

Remove(s, t) does not do anything if $s = t$. If $s \neq t$, it first resets the (s, t)-element in A_1, B_1, C_1 and D_1 to 0 by calling Reset on Q_1 and P_1. Then according to the lists which the calls of Reset on Q_1 return, it resets the elements in L_1 and R_1 to 0 by calling Reset on P_1. The lists which the four of the calls of Reset on P_1 return enable it to reset the appropriate elements in A_2, B_2, C_2 and D_2 to 0. Similar calls of Reset continue until the appropriate elements in the A_k, B_k, C_k, D_k, L_k and R_k are reset to 0 for $1 \leqq {}^{\forall}k \leqq l$.

Reachable(s, t) calls Lookup(s, t) on P_l and returns true if the Lookup on P_l returns 1 or false otherwise.

We obtain another dynamic algorithm for TC by replacing the elements in the variables by the set of actual paths in the graph. Though this new algorithm is hardly practical to implement, it is quite useful to prove the following lemma, which guarantees the correctness of the algorithm.

Lemma 4.1. *After each operation, $P'^2_{k-1} \subseteq P'_k \subseteq P'^3_{k-1}$ for $1 \leqq k \leqq l$.*

The proof is done by induction with regard to operations.

Theorem 4.2. *In terms of the worst-case time complexity, an* Init *takes $O((n^\omega + n^2) \log n)$ time, where ω is the exponent for matrix multiplication on a ring. A* Reachable *takes $O(1)$ time, and each of* Add *and* Remove *takes $O(n^2 \log n)$ time in the worst case. In terms of the amortized time complexities, it takes $O(n^3 \log n)$ time for an* Init*, $O(n^2 \log n)$ for a* Add*, $O(\log n)$ for a* Remove *and $O(1)$ for a* Reachable*. The algorithm requires $O(n^2 \log n)$ space.*

Confirming Theorem 4.2 is easy.

5 Dynamic Matrix Multiplication Algorithm

A simple algorithm for the dynamic Boolean matrix multiplication problem is obtained just by maintaining which pairs are valid using lists of the valid pairs.

In this algorithm, we maintain the following data structure.

- $2h$ Boolean matrices $X_1^{(1)}, X_2^{(1)}, X_1^{(2)}, X_2^{(2)}, \ldots, X_1^{(h)}, X_2^{(h)}$,
- hn^2 lists of integers: for each $1 \leqq a \leqq h$ and each $1 \leqq s, x \leqq n$, the list of t such that the pair $(a; s, x, t)$ is valid,
- hn^2 lists of integers: for each $1 \leqq a \leqq h$ and each $1 \leqq x, t \leqq n$, the list of s such that the pair $(a; s, x, t)$ is valid, and
- hn^2 integers: for each $1 \leqq a \leqq h$ and each $1 \leqq s, t \leqq n$, the number of x such that the pair $(a; s, x, t)$ is valid.

This algorithm does not meet the time complexity requirement in that it requires $O(hn^3)$ worst-case time for Init, but it is not a serious disadvantage, because Init requires $O(hn^3)$ amortized time from the beginning.

To prove the amortized time complexity of the operations, we can use the function Φ defined as the number of the valid pairs $(a; s, x, t)$ as a potential function in the amortized complexity analysis.

Because the length of each list is at most n, the amount of space required is $O(hn^3)$. When a large graph is concerned, the space requirement is critical for all the required data to fit in the main memory.

To reduce required memory, we use carefully desinged time values instead of the lists of valid pairs. The idea is explained in Demetrescu and Italiano's papers. With this modification, the algorithm requires $O(hn^2)$ space and the worst-case time for Init reduces to $O(n^\omega + n^2)$.

6 Implementation and Experiments

The algorithm was implemented and experiments were carried out to see the practical performance of the dynamic TC algorithm, especially when compared to the of a static algorithm for the same problem.

First, note that the implementation is slightly different from the descriptiong in previous sections.

Implementation of `Init` on Polynomials In Demetrescu and Italiano's algorithm for the Boolean matrix multiplication, `Init` was supposed to use the fast matrix multiplication algorithm to achieve the $O(n^\omega)$ worst-case time for an `Init`. Though this is interesting in theoretical aspects, the fact the amortized complexity of an `Init` is $O(hn^3)$ implies that it would be little improve of practical performance if we used the fast matrix multiplication. For this reason, `Init` performs the multiplication of two matrices over $\mathbb{B}$ just as the definition for each term, spending $\Theta(hn^3)$ worst-case time in total, in our implementation.

Interface of `Add` on a Graph In our formalization of the TC algorithm, `Add` is passed a set of edges from or to the specified vertex. In our implementation, however, the operation `Add` is split into two methods: `LazyAdd` and `Fix`. To perform the operation `Add`, the caller calls `LazyAdd` as many times as the number of the edges to add. Each call of `LazyAdd` specifies one edge to add. After specifying all the edges to add, the caller calls `Fix` The center of the update is specified as the parameter passed to `Fix`.

The interface of `Add` could be just like the one presented in section 2, but it would require passing a set of vertices to a method, which could be an overload. To avoid this, we choose to split `Add` into `LazyAdd` and `Fix` so that the parameters to a method are kept to be of simple types. One fallback of this choice is that we have to be careful to call `LazyAdd` and `Fix` always in conjunction.

Experiments

The dynamic algorithm described so far was implemented, and routines of the trivial dynamic algorithm with fast queries was written using the Floyd-Warshall algorithm. A program was written to carry out the following steps.

1. A random graph with $2(n-1)$ edges is chosen.
2. This graph is passed to `Init` as the graph G, and the reflective transitive closure of G was obtained by calling (i, j) for all $1 \leqq i, j \leqq n$.
3. One vertex v and n edges from or to the vertex v to add to G were chosen randomly.
4. These edges were added to G by `LazyAdd`. A call of `Fix` with v specified as the center of update followed in order to make `LazyAdds` take effects, and the reflective transitive closure was obtained just like step 2.
5. n edges to remove from G were chosen randomly.
6. These edges were removed from the G by `Remove`, and the resulting transitive closure was obtained.
7. Steps 3 through 6 were repeated for 100 times.

The CPU time consumed during each of steps 2, 4 and 6 was mesured.

The measurements were performed for $n = 32, 64, 128, 256, 512$.

The program was written in C++, compiled using GNU G++, and executed on Sun Ultra 60 with UltraSPARC-II 450 MHz and 2 GB memory.

Results and Discussion

Tables 1 and 2 show the average amount of CPU time consumed by each step for different values of n. Naturally the execution time of the Floyd-Warshall algorithm varied little for each graph size. However, it is surprising that the time varied little also in the dynamic algorithm. Especially, we remark the little variation in the CPU time consumed by `Remove` in the dynamic algorithm, because the fact the amortized complexity is much faster than the worst-case complexity suggested that the actual cost for each call varies much. This may be due to the few number of the changes in the transitive closure caused by `Remove`.

Figures 1 and 2 illustrate the increase in the execution time of each step in our benchmark program as the number of the vertices in G increased, and compare it betweem the dynamic and the static algorithm.

The little deviation in previous tables implies that the comparison of the efficiency between the dynamic and the static algorithms is meaningful for $n \geqq 256$ for `Add`, and for $n \geqq 512$ for `Remove`.

The dynamic algorithm consumed more time for the initialization than the Floyd-Warshall algorithm, but less time for the updates. This difference confirms that the fact the dynamic algorithm has an advantage for situations where many consecutive queries and small updates occur holds also in practical sense.

7 Concluding Remarks

Future Works

In our experiments, a random graph was used and a query on the whole transitive closure were performed after each update in our experiments. In real applications of TC, the property of the sequence of the operations performed on a graph may be different from our

setting. To examine the real performance of the algorithm, the experiments using more realistic settings of the operation sequences should be carried out.

Now that the efficiency of the dynamic algorithm is comparable to Floyd-Warshall algorithm, the performance comparison to faster static algorithms than the Floyd-Warshall algorithm seems interesting for large graphs.

The other dynamic TC algorithm Demetrescu and Italiano presented in [3] also uses the Boolean matrix multiplication algorithm but in a different way. Besides, the dynamic APSP algorithm they presented in [4] also uses a similar matrix multiplication algorithm. If we know exactly what makes these algorithms behave differently in terms of the complexity order, it might be possible to use the same subroutine in a yet different way to achieve another goal.

Conclusion

One of Demetrescu and Italiano's fully dynamic TC algorithms was given a different formalization from the original paper for simpler definition, better modularization, easier implementation and more precise complexity analyses, and a variant of their algorithm based on this reformalization was presented. Our formalization clarified the meaning of the algorithm, especially the connection and separation of the TC algorithm to the dynamic matrix multiplication algorithm.

In addition, our version of the algorithm was implemented and its performance was compared to that of the Floyd-Warshall algorithm. The results implied the usefulness of the dynamic TC algorithm at least in a situation with a dense graph and frequent queries.

Acknowledgment

This paper is based on the first author's senior thesis submitted to the Department of Information Science of the University of Tokyo.

References

[1] Coppersmith, D. and Winograd, S.: Matrix Multiplication via Arithmetic Progressions, *J. Symbolic Comput.*, Vol. 9, No. 3, pp. 251–280 (1990).

[2] Demetrescu, C.: Fully Dynamic Algorithms for Path Problems on Directed Graphs, *PhD Thesis*, Department of Computer and Systems Science, University of Rome "La Sapienza" (2001).

[3] Demetrescu, C. and Italiano, G. F.: Mantaining Dynamic Matrices for Fully Dynamic Transitive Closure, arXiv:cs.DS/0104001 (2001).

[4] Demetrescu, C. and Italiano, G. F.: Fully Dynamic All Pairs Shortest Paths with Real Edge Weights, in *Proc. 42nd IEEE FOCS*, pp. 260–267 (2001). Also *Technical Report ALCOM-FT, ALCOMFT-TR-01-150*.

[5] Henzinger, M. R. and King, V.: Fully Dynamic Biconnectivity and Transitive Closure, in *Proc. 36th IEEE FOCS*, pp. 664–672 (1995).

[6] King, V.: Fully Dynamic Algorithms for Maintaing All-Paris Shortest Paths and Transitive Closure in Digraphs, in *Proc. 40th IEEE FOCS*, pp. 81–89 (1999).

[7] Munro, I.: Efficient Determination of the Transitive Closure of a Directed Graph, *Information Processing Letters*, Vol. 1, No. 2, pp. 56–58 (1971).

[8] Nuutila, E.: Efficient Transitive Closure Computation in Large Digraphs, *Acta Polytechnica Scandinavica*, Mathematics and Computing in Engineering Series No. 74, the Finnish Academy of Technology, Helsinki (1995).

[9] Sleator, D. D. and Tarjan, R. E.: A Data Structure for Dynamic Trees, *J. Comput. Syst. Sci.*, Vol. 26, No. 3, pp. 362–391 (1983).

[10] Tarjan, R. E.: Amortized Computational Complexity, *SIAM J. Alg. Disc. Meth.*, Vol. 6, No. 2, pp. 306–318 (1985).

Graph size	`Init`	`Add`			`Remove`		
		Quartile	Median	Quartile	Quartile	Median	Quartile
32	20	10	20	20	20	20	20
64	170	70	80	80	90	100	120
128	1450	350	360	370	470	510	560
256	11220	1650	1700	1750	2480	2660	2940
512	94320	8030	8240	8380	12480	13280	14320

Table 1: CPU time consumed by the dynamic algorithm for each step for different graph sizes, in miliseconds. Each time includes the time consumed by the succeeding calls of `Reachable`.

Graph size	`Init`	`Add`			`Remove`		
		Quartile	Median	Quartile	Quartile	Median	Quartile
32	0	0	0	10	0	0	10
64	30	30	30	30	20	30	30
128	200	220	240	260	190	200	210
256	1470	1720	1850	1970	1490	1570	1640
512	11800	13660	14940	16280	11860	12590	13240

Table 2: The average CPU time consumed by Floyd-Warshall algorithm, in miliseconds. The format is the same as Table 1.

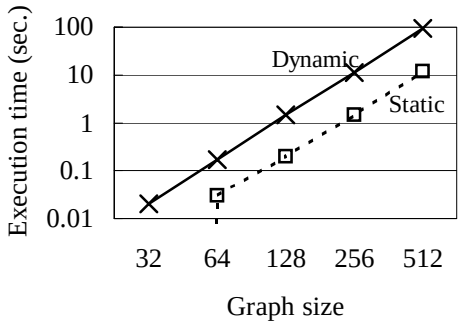

Figure 1: Comparison between the dynamic and the static algorithms of the increase in the total execution time of an `Init` and `Reachables` as the number of vertices of the graph increased. Both axes are shown in a logarithmic scale.

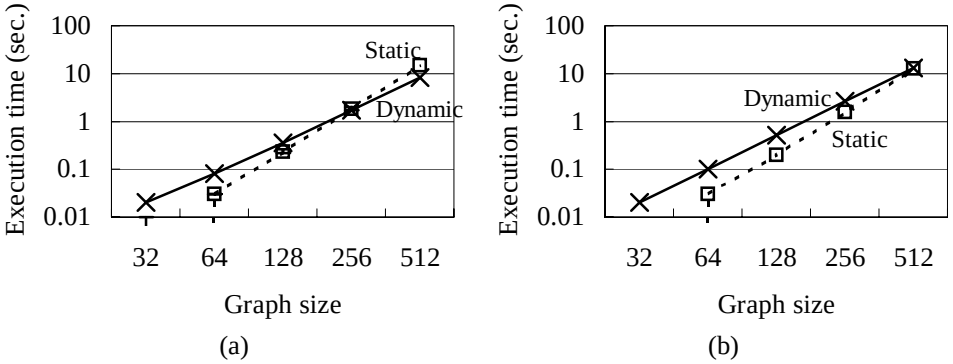

Figure 2: (a) Comparison between the dynamic and the static algorithms of the increase in the total execution time of a `Add` and `Reachables` as the number of vertices of the graph increased. Each case is represented by its median. Both axes are shown in a logarithmic scale. (b) Similar graph for `Removes` and `Reachables`.