Interdisciplinary Information Sciences Vol. 21, No. 1 (2015) 25–35
©Graduate School of Information Sciences, Tohoku University
ISSN 1340-9050 print/1347-6157 online
DOI 10.4036/iis.2015.25

Experimental Evaluations of Dynamic Algorithm for Maintaining Shortest-Paths Trees on Real-World Networks

Takashi HASEGAWA, Takehiro ITO*, Akira SUZUKI and Xiao ZHOU

Graduate School of Information Sciences, Tohoku University, Sendai 980-8579, Japan

Received May 7, 2014; final version accepted September 17, 2014

For a digraph $G = (V, A)$ and a source vertex $s \in V$, suppose that we wish to compute a shortest directed path from s to every vertex $v \in V \setminus \{s\}$ (if exists) under several arc costs. Frigioni *et al.* (2000) proposed a dynamic algorithm which efficiently reuses the shortest-paths information computed for the previous arc costs. In this paper, we experimentally evaluate how such a dynamic algorithm works efficiently for real-world networks.

KEYWORDS: dynamic algorithm, graph algorithm, shortest-paths problem

1. Introduction

One of the most fundamental problems in computer science is the (*single-source*) *shortest-paths problem*: Given an arc-cost digraph (directed graph) $G = (V, A)$ and a source vertex $s \in V$, we wish to find a shortest directed path from s to every vertex $v \in V \setminus \{s\}$ if exists. This problem can be solved in time $O(m + n \log n)$ by the well-known Dijkstra algorithm [1, Section 24.3] [2] using Fibonacci heaps for its implementation [3], where n is the number of vertices in G and m is the number of arcs in G.

The shortest-paths problem has several applications even in real-world situations. As an interesting example, the problem is used to simulate traffic jams [5, 6]. In the simulation, each car on the street is assumed to move along a shortest route (path) from the current location to its destination. Then, finding a shortest route for each car corresponds to finding a shortest path in the corresponding digraph G. Note that, in the simulation, each arc cost of G corresponds to the transit time of the street (not the length of the street), and hence it would be changed according to the traffic jam. Therefore, the shortest routes for cars vary from hour to hour, even though the graph structure is the same. We thus need to solve the shortest-paths problem several times in the simulation.

Frigioni *et al.* [4] proposed a "dynamic" algorithm to cope with such a situation: when an update operation (e.g., a change of an arc cost) is applied to the graph, their algorithm maintains the shortest paths without recomputing everything from scratch. The algorithm efficiently reuses the shortest-paths information computed for the previous arc costs.

In this paper, we experimentally evaluate how such a dynamic algorithm works efficiently for real-world networks. We implement a dynamic algorithm which is a simplified version of the algorithm by Frigioni *et al.* [4], and apply it to the graph based on the map of Ishinomaki city, Miyagi, Japan. Therefore, the dynamic algorithm in this paper is not our original from the theoretical viewpoint, but we will experimentally show that the dynamic algorithm works very efficiently in practice.

2. Preliminaries

In this section, we first define some terms and notation which will be used throughout the paper. We then introduce the well-known Dijkstra algorithm with its implementation in Section 2.3.

2.1 Definitions

Let $G = (V, A)$ be a digraph with the vertex set V and the arc set A; we sometimes denote by $V(G)$ and $A(G)$ the vertex set and arc set of G, respectively. We write $e = (u, v)$ if e is an arc from a vertex u to a vertex v. For a vertex v in G, let $N_{\text{in}}(G, v) = \{u \in V(G) \mid (u, v) \in A(G)\}$ and $N_{\text{out}}(G, v) = \{w \in V(G) \mid (v, w) \in A(G)\}$. Let $\Delta_{\text{in}} = \max\{|N_{\text{in}}(G, y)| : y \in V(G)\}$, and let $\Delta_{\text{out}} = \max\{|N_{\text{out}}(G, y)| : y \in V(G)\}$. A directed path in G from v to w is simply called a (v, w)-*path* in G. We say that a vertex w is *reachable* in G from a vertex v if there is a (v, w)-path in G.

*Corresponding author. E-mail: takehiro@ecei.tohoku.ac.jp

Let G be a digraph such that each arc $(v, w) \in A(G)$ has a non-negative real number $c(v, w) \geq 0$, called the *cost* of (v, w). For two vertices v and w in G, a (v, w)-path P in G is *shortest* if the sum of costs of all arcs in P is minimum among all (v, w)-paths in G. For a vertex s in G, a directed subgraph T of G is called a *shortest-paths tree* of G rooted at s if the following three conditions (a)–(c) hold:

(a) $V(T)$ consists of all vertices in G that are reachable from s;
(b) T forms a rooted tree with root s; and
(c) for every vertex $v \in V(T) \setminus \{s\}$, the unique path in T from s to v is a shortest (s, v)-path in G.

The root s is also called a *source* vertex. For a vertex $v \in V(T) \setminus \{s\}$, we denote by $\mathsf{parent}(v)$ the parent of v in T. Then, the tree T can be represented by memorizing $\mathsf{parent}(v)$ for all vertices $v \in V(T) \setminus \{s\}$.

Given an arc-cost digraph G and a source vertex $s \in V(G)$, the *single-source shortest-paths problem* is to find a shortest-paths tree of G rooted at s.

2.2 Min-priority queue

We implement both the Dijkstra and our algorithms by using a well-known "min-priority queue" based on the heap data structure. A *min-priority queue* Q is a data structure which maintains a set S such that each element in S has a value, called the *key*, and it supports the following operations:

- INSERT(Q, x): this operation inserts a new element x into Q, and hence $S := S \cup \{x\}$;
- EXTRACT-MIN(Q): this operation removes and returns the element $x_{\min}$ in Q with the smallest key, and hence $S := S \setminus \{x_{\min}\}$; and
- DECREASE-KEY(Q, x, k): this operation replaces the key of the element x with the new value k, which is assumed to be at most the current key of x.

One can easily implement a min-priority queue using the heap data structure so that each of the three operations above can be done in time $O(\log |S|)$. We do not explain how to construct such a data structure in this paper. (See e.g., [1, Section 6.5] for details.)

2.3 Dijkstra algorithm

In this subsection, we introduce the well-known Dijkstra algorithm.

Let G be a given digraph with an arc-cost function c, and let s be a given source vertex. In the Dijkstra algorithm, we maintain the status of each vertex v in G by two values $\mathsf{flag}(v)$ and $\mathsf{dist}(v)$, defined as follows.

(1) $\mathsf{flag}(v)$ takes one of the three statuses "unconsidered," "active" and "fixed," which represents the current status of v. If $\mathsf{flag}(v) = \mathsf{unconsidered}$, then the algorithm has not taken v into account yet; if $\mathsf{flag}(v) = \mathsf{active}$, then the algorithm is currently calculating a shortest (s, v)-path; and if $\mathsf{flag}(v) = \mathsf{fixed}$, then the algorithm has already found a shortest (s, v)-path.
(2) $\mathsf{dist}(v)$ takes a non-negative real number, which represents the current (temporary) distance from s to v; if $\mathsf{flag}(v) = \mathsf{fixed}$, then the value of $\mathsf{dist}(v)$ represents the shortest distance from s to v.

In the Dijkstra algorithm, we maintain all the active vertices v by a min-priority queue with the values of $\mathsf{dist}(v)$ as their keys. We call the procedure DIJKSTRA(G, c, Q) under the following initialization:

- $Q = \{s\}$;
- $\mathsf{flag}(s) = \mathsf{active}$ and $\mathsf{dist}(s) = 0$; and
- $\mathsf{parent}(v) = \mathsf{undefined}$, $\mathsf{flag}(v) = \mathsf{unconsidered}$, and $\mathsf{dist}(v) = +\infty$ for all vertices $v \in V(G) \setminus \{s\}$.

The procedure DIJKSTRA(G, c, Q) correctly constructs a shortest-paths tree of G rooted at s. Furthermore, the

Algorithm 1 DIJKSTRA(G, c, Q)

```
1: while $Q$ is not empty do
2:   $w :=$ EXTRACT-MIN$(Q)$
3:   $\mathsf{flag}(w) := \mathsf{fixed}$
4:   for each vertex $x \in N_{\mathrm{out}}(G, w)$ such that $\mathsf{flag}(x) \neq \mathsf{fixed}$ do
5:     if $\mathsf{dist}(x) > \mathsf{dist}(w) + c(w, x)$ then
6:       $\mathsf{dist}(x) := \mathsf{dist}(w) + c(w, x)$
7:       $\mathsf{parent}(x) := w$
8:       if $\mathsf{flag}(x) = \mathsf{unconsidered}$ then
9:         $\mathsf{flag}(x) := \mathsf{active}$
10:        INSERT$(Q, x)$
11:      else $\{\mathsf{flag}(x) = \mathsf{active}\}$
12:        DECREASE-KEY$(Q, x, \mathsf{dist}(x))$
13:      end if
14:    end if
15:  end for
16: end while
```

procedure runs in $O(m \log n) = O(n\Delta_{\text{out}} \log n)$ time, where $n = |V(G)|$ and $m = |A(G)|$. (See e.g., [1, Section 24.3] for details.)

3. Algorithms for Arc-Cost Update

In this section, we explain a dynamic algorithm for maintaining a shortest-paths tree of a digraph G rooted at a source vertex s. Assume in this paper that we can change only the cost of a single arc in G, as the update operation for G. More formally, let c_t be the arc-cost function of G which is obtained from an arc-cost function c_{t-1} of G by applying such an update operation; then, we have $|\{(u, v) \in A(G) : c_t(u, v) \neq c_{t-1}(u, v)\}| = 1$.

3.1 Static algorithm

One of the simplest static algorithm is to solve the single-source shortest-paths problem for the arc-cost function c_t by applying the Dijkstra algorithm in Section 2.3. We employ this simple static algorithm as a benchmark of the experimental evaluations in Section 4.

3.2 Dynamic algorithm

In this subsection, we describe a dynamic algorithm which is a simplified version of the algorithm given by Frigioni *et al.* [4]. Therefore, this algorithm is not our original from the theoretical viewpoint, as we have mentioned in Introduction.

Suppose that we have a shortest-paths tree T_{t-1} of G rooted at s for an arc-cost function c_{t-1}. We assume that each vertex v in T_{t-1} is associated with three values $\mathsf{parent}(v)$, $\mathsf{flag}(v)$ and $\mathsf{dist}(v)$ such that

(a) $\mathsf{parent}(v)$ is the parent of v in T_{t-1};
(b) $\mathsf{flag}(v) = \mathsf{fixed}$; and
(c) $\mathsf{dist}(v)$ is equal to the distance of a shortest (s, v)-path in G with respect to the arc-cost function c_{t-1}.

Notice that, for the first time, we can obtain such a shortest-paths tree by the Dijkstra algorithm in Section 2.3. From the second time, our algorithm constructs a shortest-paths tree T_t for the arc-cost function c_t which satisfies the conditions (a)–(c) above, by maintaining the current shortest-paths tree T_{t-1} for c_{t-1}.

Our algorithm takes different strategies depending on whether the updated cost is increased or decreased. To avoid the confusion, for each vertex v in G, we sometimes denote by $\mathsf{dist}_{t-1}(v)$ and $\mathsf{dist}_t(v)$ the distances of shortest (s, v)-paths for the arc-cost functions c_{t-1} and c_t, respectively. Similarly, we sometimes denote by $\mathsf{parent}_{t-1}(v)$ and $\mathsf{parent}_t(v)$ the parents of v in shortest-paths trees T_{t-1} and T_t, respectively.

3.2.1 Increase case

We first consider the case where the updated cost is increased. Let $e = (u, v)$ be the arc such that $c_t(u, v) > c_{t-1}(u, v)$.

Consider the case where $e \notin A(T_{t-1})$. Then, each vertex z in T_{t-1} has a shortest (s, z)-path (under the arc-cost function c_{t-1}) which does not pass through the arc (u, v). Since only the cost of (u, v) is updated from c_{t-1} and $c_t(u, v) > c_{t-1}(u, v)$, the (s, z)-path is a shortest (s, z)-path also for the arc-cost function c_t. We thus have $T_t = T_{t-1}$.

We then consider the case where $e \in A(T_{t-1})$. We denote by T^v_{t-1} the subtree of T_{t-1} which is rooted at v and is induced by v and its all descendants in T_{t-1}. (See Fig. 1(a).) Then, the same argument above implies that, for every vertex z in $T_{t-1} \setminus T^v_{t-1}$, the (s, z)-path in T_{t-1} is a shortest (s, z)-path also for c_t. (See the vertex z in Fig. 1.) Therefore, it suffices to update the shortest distances only for the vertices in T^v_{t-1}. For each vertex w in T^v_{t-1}, there are the following two possibilities to consider.

(i) The (s, w)-path in T_{t-1} is a shortest (s, w)-path also for c_t. (See the vertex w_1 in Fig. 1.) In this case, the distance from s to w is increased by $\delta = c_t(u, v) - c_{t-1}(u, v)$, that is, $\mathsf{dist}_t(w) = \mathsf{dist}_{t-1}(w) + \delta$.

(ii) The (s, w)-path in T_{t-1} is not a shortest (s, w)-path for c_t. (See the vertex w_2 in Fig. 1.) In this case, a shortest (s, w)-path for c_t does not pass through the arc (u, v), and has the distance less than $\mathsf{dist}_{t-1}(w) + \delta$. Then, such a shortest (s, w)-path can be divided into a shortest (s, x)-path and a shortest (y, w)-path such that $x \in$

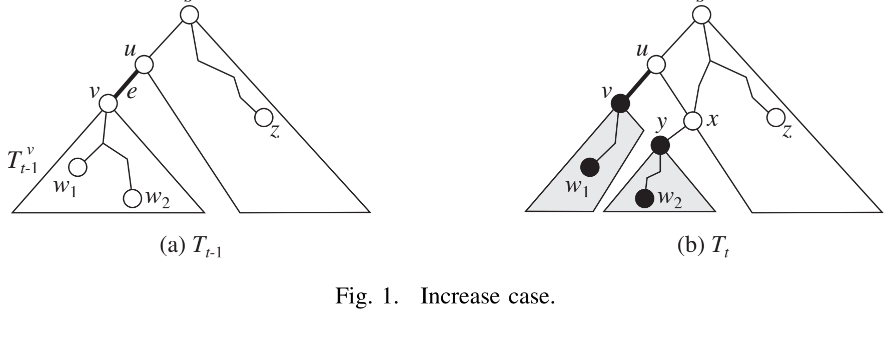

Fig. 1. Increase case.

Algorithm 2 INCREASE(G, c_t, T_{t-1})

```
1: $Q := \emptyset$
2: Let $e = (u, v)$ be the arc such that $c_t(u, v) > c_{t-1}(u, v)$, and let $\delta = c_t(u, v) - c_{t-1}(u, v)$
3: for each vertex $y$ in $T^v_{t-1}$ do
4:     $\mathsf{dist}(y) := \mathsf{dist}(y) + \delta$
5:     for each vertex $x \in N_{\text{in}}(G, y) \cap V(T_{t-1} \setminus T^v_{t-1})$ do
6:         if $\mathsf{dist}(y) > \mathsf{dist}(x) + c_t(x, y)$ then
7:             $\mathsf{dist}(y) := \mathsf{dist}(x) + c_t(x, y)$
8:             $\mathsf{parent}(y) := x$
9:         end if
10:    end for
11:    $\mathsf{flag}(y) := \mathsf{active}$
12:    INSERT$(Q, y)$
13: end for
14: DIJKSTRA$(G, c_t, Q)$
```

$V(T_{t-1} \setminus T^v_{t-1})$, $y \in V(T^v_{t-1})$ and there is an arc from x to y, as illustrated in Fig. 1(b).
To deal with the case (ii) above, we apply the Dijkstra algorithm to the vertices only in T^v_{t-1}. More formally, the procedure INCREASE(G, c_t, T_{t-1}) constructs a shortest-paths tree T_t of G rooted at s for the arc-cost function c_t.

We now estimate the running time of INCREASE(G, c_t, T_{t-1}). The updated arc $e = (u, v)$ can be found in time $O(m)$. Let n_{inc} be the number of vertices in the subtree T^v_{t-1}. For each vertex y in T^v_{t-1}, the lines 5–10 can be executed in time $O(\Delta_{\text{in}})$, and the line 12 can be done in $O(\log n_{\text{inc}})$. Therefore, the lines 4–12 of INCREASE(G, c_t, T_{t-1}) can be done in time $O(\Delta_{\text{in}} + \log n_{\text{inc}})$ for one vertex y in T^v_{t-1}. Since these lines are executed for all n_{inc} vertices in T^v_{t-1}, the lines 3–13 take time $O(n_{\text{inc}}\Delta_{\text{in}} + n_{\text{inc}} \log n_{\text{inc}})$. Then, the line 14 can be done in time $O(n_{\text{inc}}\Delta_{\text{out}} \log n_{\text{inc}})$. In this way, INCREASE$(G, c_t, T_{t-1})$ runs in time $O(m + n_{\text{inc}}\Delta_{\text{in}} + n_{\text{inc}}\Delta_{\text{out}} \log n_{\text{inc}})$ in total.

3.2.2 Decrease case

We then consider the case where the updated cost is decreased. Let $e = (u, v)$ be the arc such that $c_t(u, v) < c_{t-1}(u, v)$.

Consider the case where $\mathsf{dist}_{t-1}(v) \le \mathsf{dist}_{t-1}(u) + c_t(u, v)$. Then, the (s, v)-path in T_{t-1} does not pass through the arc (u, v), and hence $e \notin A(T_{t-1})$. Therefore, the (s, v)-path in T_{t-1} is a shortest (s, v)-path also for c_t. We thus have $T_t = T_{t-1}$.

We then consider the case where $\mathsf{dist}_{t-1}(v) > \mathsf{dist}_{t-1}(u) + c_t(u, v)$. In this case, all vertices w in T^v_{t-1} decrease their shortest distances from the previous ones. (See the vertex w_1 in Fig. 2. Note that, however, this case may happen even when the arc (u, v) is not in T_{t-1}.) Furthermore, some vertices in $T_{t-1} \setminus T^v_{t-1}$ may change their shortest distances, too. (See the vertex w_2 in Fig. 2.) Let w be a vertex in $T_{t-1} \setminus T^v_{t-1}$ such that $\mathsf{dist}_t(w) < \mathsf{dist}_{t-1}(w)$. Then, there exists an arc (x, y) from a vertex $x \in V(T^v_{t-1})$ to a vertex $y \in V(T_{t-1} \setminus T^v_{t-1})$ such that y is an ancestor of w in T_{t-1}. Indeed, this update can be seen as an update in the Dijkstra algorithm: we thus apply the Dijkstra algorithm to all vertices in T_{t-1}, as follows.

- We take the vertex v as the source vertex. As the initialization, we set $\mathsf{flag}(v) = \mathsf{active}$ and $\mathsf{dist}(v) = \mathsf{dist}_{t-1}(u) + c_t(u, v)$ instead of setting $\mathsf{dist}(v) = 0$.
- Since $\mathsf{flag}(w) = \mathsf{fixed}$ for all vertices $w \in V(T_{t-1}) \setminus \{v\}$, we initialize their statuses to $\mathsf{flag}(w) = \mathsf{unconsidered}$. However, we do not change (initialize) the values $\mathsf{parent}(w)$ and $\mathsf{dist}(w)$ $(= \mathsf{dist}_{t-1}(w))$, and hence the values depict the parent of w in T_{t-1} and the shortest distance from s to w for c_{t-1}, respectively. Therefore, the Dijkstra algorithm updates $\mathsf{parent}(w)$ and $\mathsf{dist}(w)$ only when it finds a shorter (s, w)-path with respect to c_t.

Then, the procedure DECREASE(G, c_t, T_{t-1}) constructs a shortest-paths tree T_t of G for the arc-cost function c_t. It should be noted that the Dijkstra algorithm called in the line 9 of DECREASE(G, c_t, T_{t-1}) does not always re-compute the

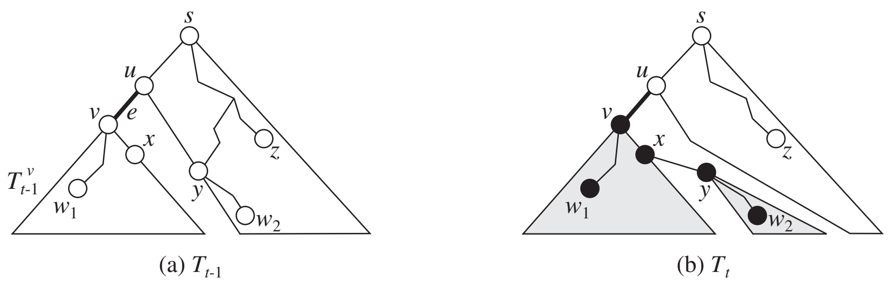

Fig. 2. Decrease case.

Algorithm 3 DECREASE(G, c_t, T_{t-1})

```
1: Let $e = (u, v)$ be the arc such that $c_t(u, v) < c_{t-1}(u, v)$
2: $\mathsf{dist}(v) := \mathsf{dist}(u) + c_t(u, v)$
3: $\mathsf{parent}(v) := u$
4: $\mathsf{flag}(v) :=$ active
5: $Q := \{v\}$
6: for each vertex $w$ in $V(T_{t-1}) \setminus \{v\}$ do
7:     $\mathsf{flag}(w) :=$ unconsidered
8: end for
9: DIJKSTRA$(G, c_t, Q)$
10: for each vertex $w$ in $V(T_{t-1}) \setminus \{v\}$ such that $\mathsf{flag}(w) :=$ unconsidered do
11:     $\mathsf{flag}(w) :=$ fixed
12: end for
```

shortest paths of all vertices in T_{t-1}, because $\mathsf{dist}_t(z) = \mathsf{dist}_{t-1}(z)$ holds for a vertex z which has no vertex $y \in N_{\text{in}}(G, z)$ such that $\mathsf{dist}_t(y) < \mathsf{dist}_{t-1}(y)$. Since the Dijkstra algorithm did not take such a vertex z into account, the vertex z remains $\mathsf{flag}(z) =$ unconsidered even after the execution of the Dijkstra algorithm in DECREASE(G, c_t, T_{t-1}). We thus need to back the status of z to $\mathsf{flag}(z) =$ fixed.

We estimate the running time of DECREASE(G, c_t, T_{t-1}). The updated arc $e = (u, v)$ can be found in time $O(m)$. Clearly, the procedure runs in time $O(n)$ except for the line 9. We thus estimate the running time of DIJKSTRA(G, c_t, Q) called in the line. Let n_{dec} be the number of vertices y such that $\mathsf{dist}_t(y) < \mathsf{dist}_{t-1}(y)$. Notice that, in the Dijkstra algorithm in Section 2.3, only such vertices are inserted into the min-priority queue Q. Therefore, the Dijkstra algorithm can be executed in time $O(n_{\text{dec}}\Delta_{\text{out}} \log n_{\text{dec}})$. Thus, DECREASE$(G, c_t, T_{t-1})$ runs in time $O(m + n + n_{\text{dec}}\Delta_{\text{out}} \log n_{\text{dec}})$ in total.

4. Experimental Evaluations

In this section, we experimentally evaluate the dynamic algorithm.

4.1 Machine spec and graph data

We wrote and complied both Dijkstra and dynamic algorithms by Microsoft Visual C++ 2012 under the optimization to maximize speed (/O2). The experiments were done on a Dual Intel Xeon E5645 Processor with 24 GB of RAM memory (DDR3-1333/PC3-10600), running Windows 7 Professional SP1 (64-bit). Each processor has 6 cores sharing a 12 MB L3 cache, and each core has a 256 KB private L2 cache and 2.4 GHz speed; we use only one core for experiments. We report CPU times measured with the timeGetTime function by employing the timeBeginPeriod and timeEndPeriod functions to obtain accuracy of 1 ms. All the running times reported in our experiments were averaged over ten different runs.

The graph data used in this paper is provided by i-Transport Lab. Co., Ltd. The graph is constructed from the map of Ishinomaki city, Miyagi, Japan, and has 15,379 vertices and 38,680 arcs, that is, $n = 15{,}379$ and $m = 38{,}680$. For the graph, $\Delta_{\text{out}} = 6$ and $\Delta_{\text{in}} = 5$.

We randomly chose 200 vertices from the graph as source vertices. We have assumed in this paper that we can change only the cost of a single arc, as the update operation. For every source vertex, we applied four different update operations to each of the 38,680 arcs: we changed the cost $c_{t-1}(u, v)$ of a single arc (u, v) to

- $c_t(u, v) = +\infty$.
- $c_t(u, v) = 2 \cdot c_{t-1}(u, v)$;
- $c_t(u, v) = 0.5 \cdot c_{t-1}(u, v)$; and
- $c_t(u, v) = 0$,

that correspond to the situations where the road is closed, becomes crowded, gets less crowded, and becomes empty, respectively. Therefore, the number of instances of our experiments is $200 \times 38{,}680 \times 4 = 30{,}944{,}000$ in total.

4.2 Experiments

As a preprocessing of our dynamic algorithm, we first apply the Dijkstra algorithm under the arc-cost function c_{t-1} to obtain a shortest-paths tree T_{t-1}. Note that this preprocessing is not counted in the running time. Then, we apply the dynamic algorithm under the arc-cost function c_t with the shortest-paths tree T_{t-1}.

Figures A·1–A·4 and Tables A·1–A·4 show the experimental results. Note that each of figures and tables shows one type of update operations, and hence it contains $200 \times 38{,}680 = 7{,}736{,}000$ instances. As a benchmark, we also apply the Dijkstra algorithm for each instance under the arc-cost function c_t.

- The x-coordinates of the figures and the "changed vertices" in the tables mean the numbers of vertices w such that $\mathsf{dist}_t(w) \neq \mathsf{dist}_{t-1}(w)$ or $\mathsf{parent}_t(w) \neq \mathsf{parent}_{t-1}(w)$, that is, their shortest distances under c_t or their parents in the shortest-paths tree T_t are changed from the previous ones under c_{t-1} by the arc-cost update operation.

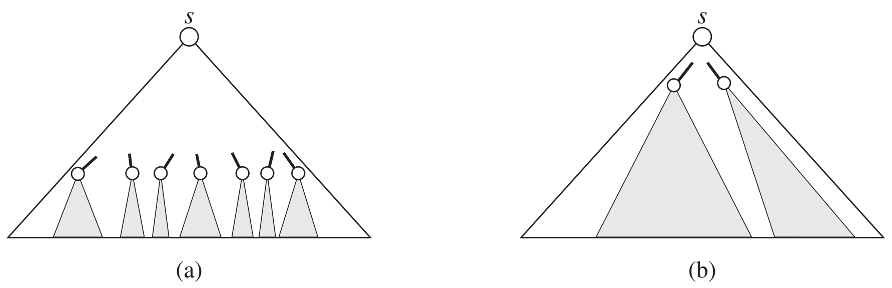

Fig. 3. Image of (a) arcs that are far from the source vertex s and (b) arcs that are close to s.

- The y-coordinates on the left side of the figures and the "Dijkstra" and "Ours" in the tables mean the running times of algorithms in millisecond. Note that, in the tables, the running time in each cell is averaged over the corresponding instances.
- The y-coordinates on the right side of the figures and the "integrated ratio" in the tables mean the integrated ratios (%) of the numbers of instances. For example, in Fig. A·1 and Table A·1, the integrated ratio is 99.63% when the number of changed vertices is 2,999; this means that $(7{,}640{,}943 + 46{,}915 + 19{,}766)/7{,}736{,}000 = 99.63\%$ instances have *at most* 2,999 changed vertices.

In the figures, each blue point represents the running time of the Dijkstra algorithm, while each orange point represents the one of our dynamic algorithm. Note that each figure contains $(200 \times 38{,}680 =)$ 7,736,000 blue points and 7,736,000 orange points, but many points are placed on the same positions. The gray line indicates the integrated ratio of instances.

4.3 Discussions

Since the Dijkstra algorithm recomputes everything from the scratch, its running time does not depend on the number of changed vertices; this can be seen from the figures and tables. On the other hand, the running time of our dynamic algorithm depends on the number of changed vertices as we have estimated in Section 3.2. By taking the summation over the red numbers in Tables A·1–A·4, our algorithm runs faster than the simple Dijkstra algorithm for 99.93% instances (among 30,944,000 total instances).

Among the four update operations, the two decrease cases shown in Tables A·3 and A·4 tend to be faster than the two increase cases shown in Tables A·1 and A·2. This is because, although both Algorithm 2 (for the increase case) and Algorithm 3 (for the decrease case) employ the Dijkstra algorithm, the increase case needs preprocessing steps heavier than the decrease case: the lines 3–13 of Algorithm 2 takes time $O(n_{\rm inc}\Delta_{\rm in} + n_{\rm inc}\log n_{\rm inc})$, while Algorithm 3 runs in time $O(n)$ except for executing the Dijkstra algorithm in the line 9.

We emphasize that 98.36% instances (among 30,944,000 total instances) have at most 999 changed vertices, and our dynamic algorithm recomputes the shortest-paths trees for these instances much faster than the simple Dijkstra algorithm. We note that the number of instances with small numbers of changed vertices is usually much larger than that of instances with large numbers of changed vertices. As illustrated in Fig. 3(a), there are many arcs (and hence many instances) that are far from the source vertex s. If we change the costs of such arcs, the numbers of changed vertices are usually small. (See the gray subtrees in Fig. 3(a).) On the other hand, the numbers of changed vertices would be large if we change the costs of arcs that are close to s, but there are only a few such arcs in the graph. (See Fig. 3(b).) Therefore, we conclude that our dynamic algorithm runs faster than the Dijkstra algorithm for many cases.

5. Conclusion

In this paper, we have experimentally evaluated the dynamic algorithm when applied to a real-world network. Our algorithm runs faster than the simple Dijkstra algorithm for 99.93% instances. It remains open to deal with the case where more than one arc-cost is changed at the same time.

Acknowledgments

We are grateful to i-Transport Lab. Co., Ltd. for providing us the graph data for our experiments. We would like to express our gratitude for fruitful discussions with Masao Kuwahara, JinYoung Kim, Kentaro Wada, Takeshi Ohata, Hisatomo Hanabusa, Katsuaki Koide, Kei Uchizawa, and Kazuki Yomogida. We also thank anonymous referees for their helpful suggestions. This work is partially supported by Grant-in-Aid for Scientific Research Grant Numbers 25106504 and 25330003 (T. Ito) and 26730001 (A. Suzuki).

REFERENCES

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, third edition, The MIT Press (2009).

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

[3] Fredman, M. L., and Tarjan, R. E., "Fibonacci heaps and their use in improved network optimization algorithms," *J. ACM*, **34**: 596–615 (1987).

[4] Frigioni, D., Marchetti-Spaccamela, A., and Nanni, U., "Fully dynamic algorithms for maintaining shortest paths trees," *J. Algorithms*, **34**: 251–281 (2000).

[5] Tamamoto, G., Kuwahara, M., Horiguchi, R., Chung, E., Tanaka, S., Satou, K., and Shiraishi, T., "Application of traffic simulation SOUND to the Tokyo metropolitan network," *Proc. of 11th World Congress on Intelligent Transport Systems*, **Nagoya**: (2004).

[6] Yoshii, T., and Kuwahara, M., "SOUND: A traffic simulation model for oversaturated traffic flow on urban expressways," *Proc. of 7th World Conference on Transportation Research*, **Sydney**: (1995).

A Experimental Results

Table A·1. The update operation $c_t(u, v) = +\infty$.

Changed vertices	Dijkstra (ms)	Ours (ms)	Number of instances	Integrated ratio (%)
0–999	3.25	0.50	7,640,943	98.77
1,000–1,999	3.27	0.94	46,915	99.38
2,000–2,999	3.25	1.36	19,766	99.63
3,000–3,999	3.27	1.79	8,605	99.74
4,000–4,999	3.25	2.23	7,718	99.84
5,000–5,999	3.28	2.62	2,436	99.88
6,000–6,999	3.27	3.18	1,338	99.89
7,000–7,999	3.29	3.63	761	99.90
8,000–8,999	3.23	4.06	1,219	99.92
9,000–9,999	3.21	4.59	1,233	99.93
10,000–10,999	3.21	5.00	1,953	99.96
11,000–11,999	3.22	5.58	1,132	99.97
12,000–12,999	3.24	6.13	903	99.99
13,000–13,999	3.24	6.64	303	99.99
14,000–14,999	3.24	7.21	588	99.99
15,000–15,379	3.23	7.70	187	100.00

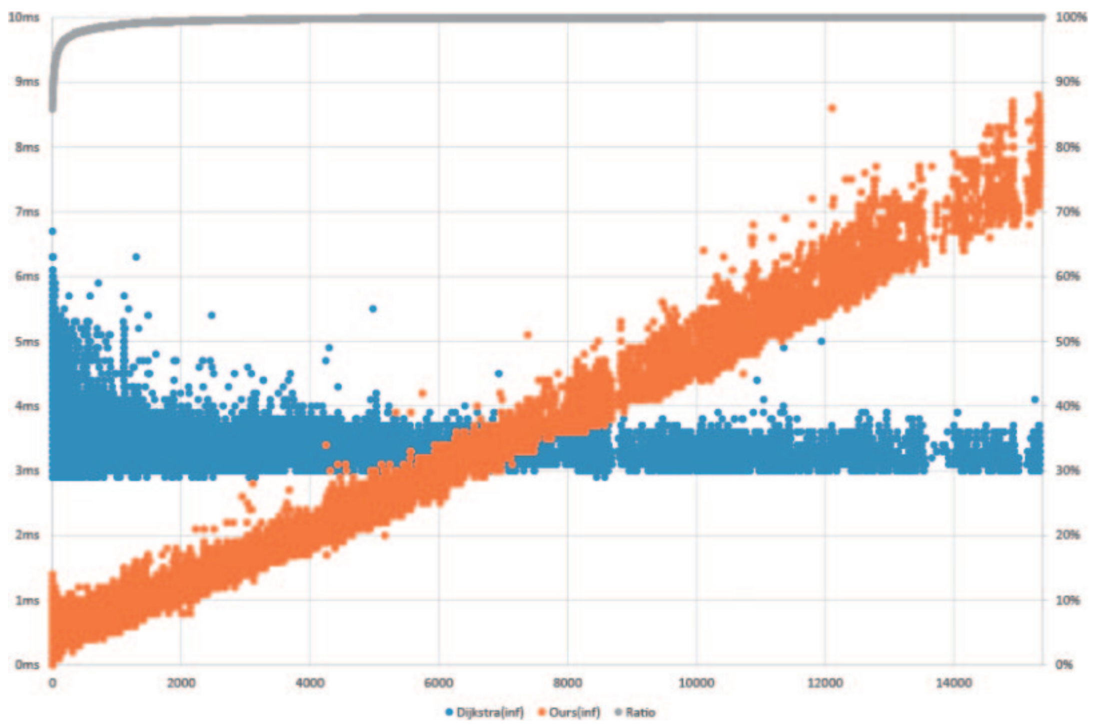

Fig. A·1. The update operation $c_t(u, v) = +\infty$.

Table A·2. The update operation $c_t(u, v) = 2 \cdot c_{t-1}(u, v)$.

Changed vertices	Dijkstra (ms)	Ours (ms)	Number of instances	Integrated ratio (%)
0–999	3.25	0.50	7,642,339	98.79
1,000–1,999	3.27	0.89	46,047	99.38
2,000–2,999	3.25	1.27	19,664	99.64
3,000–3,999	3.27	1.66	8,528	99.75
4,000–4,999	3.25	2.07	7,628	99.85
5,000–5,999	3.29	2.41	2,411	99.88
6,000–6,999	3.24	2.91	1,312	99.90
7,000–7,999	3.27	3.36	745	99.91
8,000–8,999	3.21	3.79	1,163	99.92
9,000–9,999	3.22	4.21	1,217	99.94
10,000–10,999	3.22	4.62	1,889	99.96
11,000–11,999	3.23	5.19	1,100	99.97
12,000–12,999	3.22	5.65	893	99.99
13,000–13,999	3.24	6.07	298	99.99
14,000–14,999	3.24	6.77	581	99.99
15,000–15,379	3.21	6.92	185	100.00

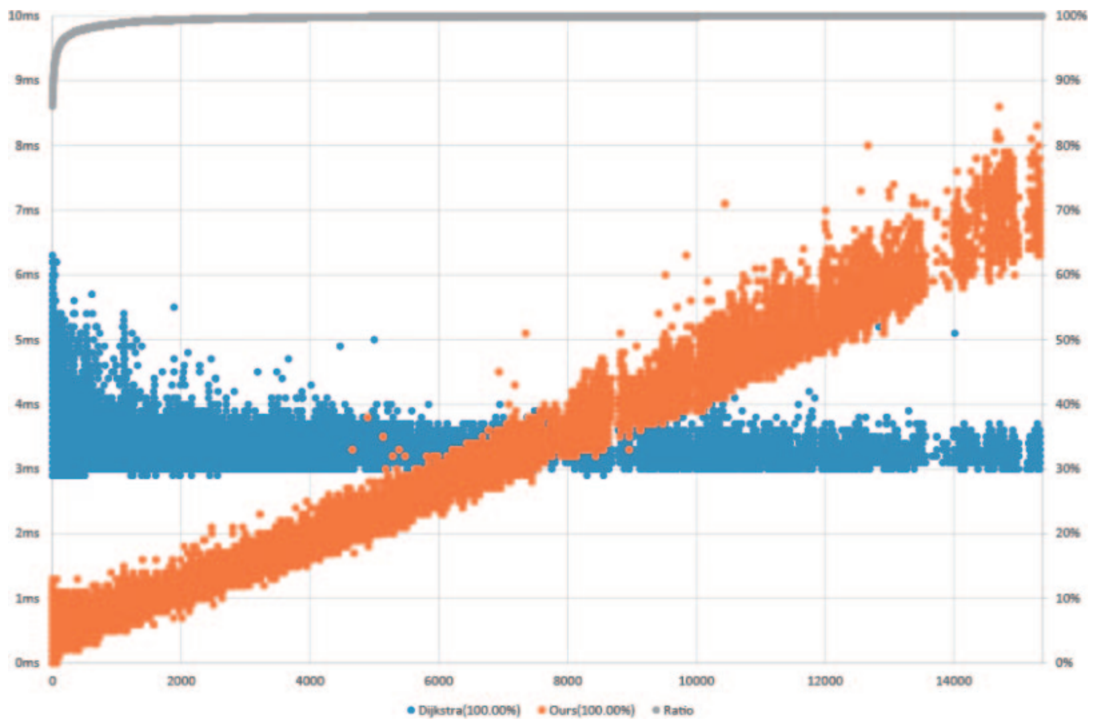

Fig. A·2. The update operation $c_t(u, v) = 2 \cdot c_{t-1}(u, v)$.

Table A·3. The update operation $c_t(u, v) = 0.5 \cdot c_{t-1}(u, v)$.

Changed vertices	Dijkstra (ms)	Ours (ms)	Number of instances	Integrated ratio (%)
0–999	3.25	0.50	7,604,020	98.29
1,000–1,999	3.27	0.72	64,593	99.13
2,000–2,999	3.25	0.91	27,250	99.48
3,000–3,999	3.28	1.13	11,683	99.63
4,000–4,999	3.25	1.33	10,259	99.76
5,000–5,999	3.30	1.53	3,426	99.81
6,000–6,999	3.27	1.77	2,152	99.84
7,000–7,999	3.27	2.01	1,395	99.85
8,000–8,999	3.24	2.23	1,654	99.88
9,000–9,999	3.22	2.42	1,980	99.90
10,000–10,999	3.20	2.62	2,805	99.94
11,000–11,999	3.21	2.93	1,666	99.96
12,000–12,999	3.22	3.10	1,543	99.98
13,000–13,999	3.21	3.38	483	99.99
14,000–14,999	3.25	3.75	859	99.99
15,000–15,379	3.24	3.90	232	100.00

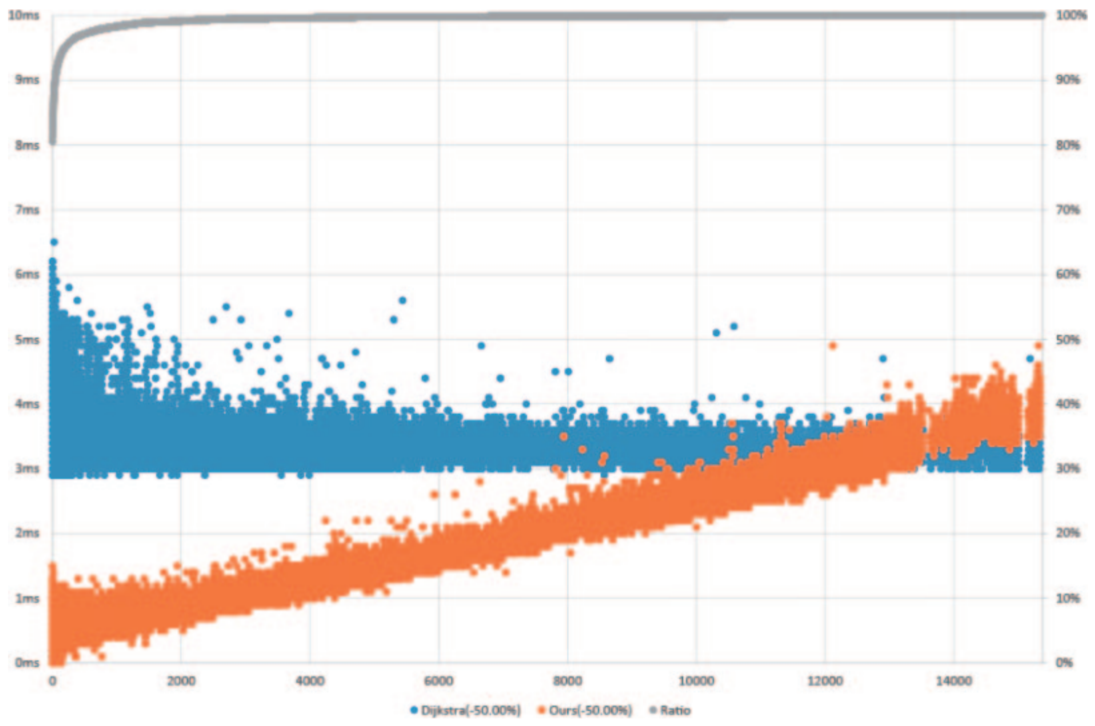

Fig. A·3. The update operation $c_t(u, v) = 0.5 \cdot c_{t-1}(u, v)$.

Table A·4. The update operation $c_t(u, v) = 0$.

Changed vertices	Dijkstra (ms)	Ours (ms)	Number of instances	Integrated ratio (%)
0–999	3.24	0.50	7,551,811	97.62
1,000–1,999	3.26	0.72	90,619	98.79
2,000–2,999	3.25	0.91	37,995	99.28
3,000–3,999	3.27	1.13	16,146	99.49
4,000–4,999	3.24	1.33	13,722	99.67
5,000–5,999	3.27	1.53	5,086	99.73
6,000–6,999	3.25	1.77	2,969	99.77
7,000–7,999	3.28	2.00	2,033	99.80
8,000–8,999	3.24	2.20	1,993	99.82
9,000–9,999	3.22	2.44	3,017	99.86
10,000–10,999	3.22	2.65	3,799	99.91
11,000–11,999	3.22	2.92	2,305	99.94
12,000–12,999	3.21	3.13	2,376	99.97
13,000–13,999	3.21	3.37	689	99.98
14,000–14,999	3.25	3.74	1,140	99.99
15,000–15,379	3.23	3.89	300	100.00

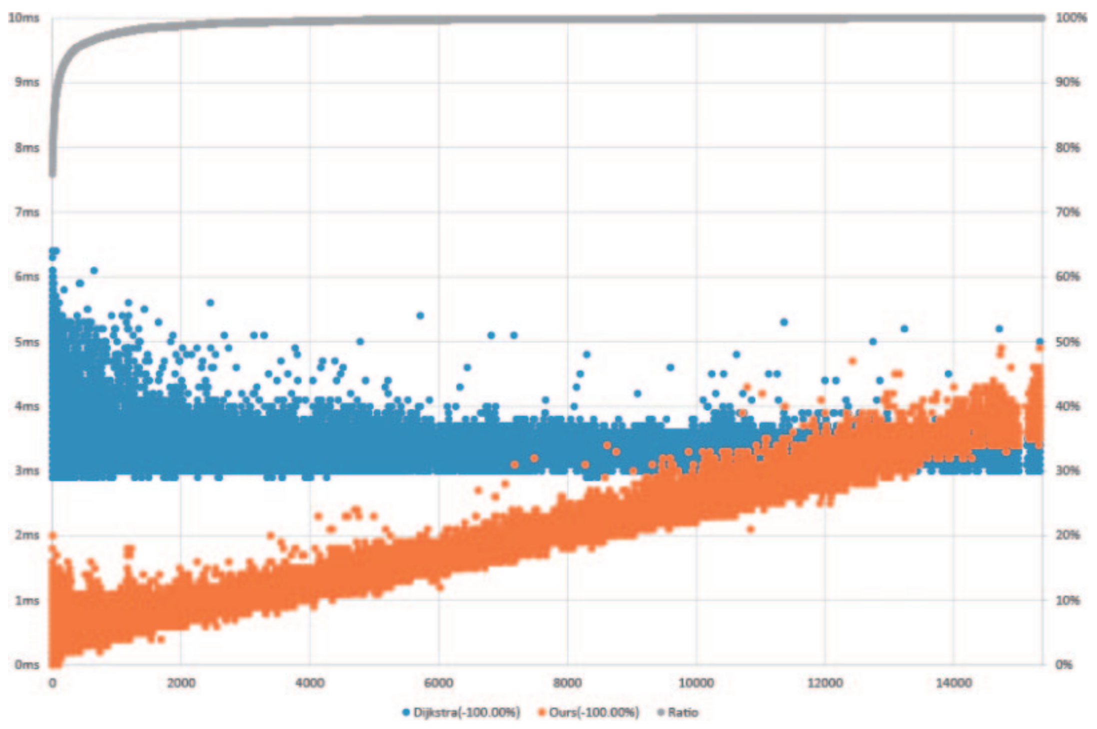

Fig. A·4. The update operation $c_t(u, v) = 0$.