Generation and Maintenance of 2-hop Labels for Accelerating Group Steiner Tree Search on Graphs

The road map of this supplement is as follows.
- In Section S1, we provide proofs that are omitted in the main contents.
- In Section S2, we provide detailed discussions on complexities of algorithms.

S1. PROOFS OF THEOREMS

We demonstrate theorems and proofs as follows.

Theorem 1. *A set of 2-hop labels that satisfies the GST-customized 2-hop cover constraint supports the query of a shortest path between every pair of a vertex $v \in V$ and a candidate vertex group $g \in \Gamma_{all}$.*

Proof. We show that we can query the shortest distance/path between every pair of a vertex $v \in V$ and a candidate vertex group $g \in \Gamma_{all}$ using a set L of 2-hop labels that satisfies the above constraint as follows. There are two different cases:

1. Case 1: there is a path between v and g on the original graph. Since each mock edge weight M is larger than the total weight of all non-mock edges, a shortest path between v and v_g on the transformed graph contains a shortest path between v and g on the original graph. Since $|\{v, v_g\} \cap D| < 2$, there is a common hub vertex $u \in C(v) \cap C(v_g)$ in a shortest path between v and v_g. Thus, we can use L to query the shortest distance between v and v_g, which is smaller than $2M$ and equals the shortest distance between v and g on the original graph plus M. Hence, equivalently, we can use L to query the shortest distance between v and g. Furthermore, since there is no mock vertex in the middle of a shortest path between v and v_g on the transformed graph, we can also iteratively query the shortest path between v and g on the original graph by adding predecessors into labels.
2. Case 2: there is no path between v and g on the original graph. Thus, v and v_g are not properly connected, and there may not be a common hub vertex $u \in C(v) \cap C(v_g)$. If there is no common hub vertex $u \in C(v) \cap C(v_g)$, then we can directly deduce that there is no path between v and g on the original graph. If there is a common hub vertex $u \in C(v) \cap C(v_g)$, then since v and v_g are not properly connected on the transformed graph, there must be at least one mock vertex in the middle of any path between v and v_g, and as a result the queried distance of any path between v and v_g is no smaller than $2M$, which still indicates that there is no path between v and g on the original graph.

Hence, this theorem holds. □

Theorem 2. *A set of 2-hop labels that has the GST-customized canonical property is minimal for meeting the GST-customized 2-hop cover constraint.*

Proof. Consider a set L of 2-hop labels that has the GST-customized canonical property. For every pair of properly connected vertices v_i and v_j such that $|\{v_i, v_j\} \cap D| < 2$, let v_k be the vertex with the highest rank in all shortest paths between v_i and v_j. We have $v_k \notin D$ and $v_k \in C(v_i) \cap C(v_j)$, since v_k has the highest rank in all shortest paths between v_i and v_k, as well as between v_j and v_k, and no vertices in these shortest paths can be hubs of v_k, except itself. Thus, L meets the GST-customized 2-hop cover constraint. Moreover, consider an arbitrary label $(v_i, d_{v_i v_j}) \in L(v_j)$. We have (i) v_i and v_j are properly connected; (ii) $|\{v_i, v_j\} \cap D| < 2$; and (iii) the rank of v_i is the highest among all vertices in all shortest paths between v_i and v_j. Thus, there is no other vertex $v_k \in C(v_i) \cap C(v_j)$ in a shortest path between v_i and v_j, *i.e.*, deleting this arbitrary label makes L not satisfy the GST-customized 2-hop cover constraint any more. Hence, this theorem holds. □

Theorem 3. *The set L of 2-hop labels generated by* HL4GST *has the GST-customized canonical property, and thus is minimal for meeting the GST-customized 2-hop cover constraint.*

Proof. Let L^{can} be the set of labels that has the GST-customized canonical property. We prove that every label in L^{can} is also in L as follows.

Consider an arbitrary label $(u, d_{vu}) \in L^{can}(v)$. We observe that (i) u and v are properly connected; (ii) $|\{u, v\} \cap D| < 2$; and (iii) the rank of u is the highest among all vertices in all shortest paths between u and v. Thus, we have $d_{vu} < 2M$, and the generation of the above label cannot be pruned by GST-customized pruning technique in Line 11 of HL4GST. Further consider the labeling process for hub u in HL4GST. HL4GST generates labels with hub u by spreading u to other vertices, starting from u, via a Dijkstra-style process. Suppose that the spread of hub u to v along a shortest path between u and v stops at a middle vertex v' due to the query pruning technique in Line 6 of HL4GST. This means that, before inserting $(u, d_{v'u})$ into $L'(v')$, the queried distance between u and v' is no larger than $d_{v'u}$. As a result, there must be a common hub vertex $z \in C(u) \cap C(v')$ such that z is in a shortest path between u and v', and $r(z) > r(u)$. This contradicts with the fact that the rank of u is the highest among all vertices in all shortest paths between u and v'. Consequently, the spread of hub u to v along a shortest path between u and v cannot stop at a middle vertex, and as a result HL4GST inserts (u, d_{vu}) into $L'(v)$.

We further show that HL4GST also inserts (u, d_{vu}) into $L(v)$ as follows. When it checks $(u, d_{vu}) \in L'(v)$ in Line 18 during the cleaning process, it computes $d'(u, v)$ using $L'_{>r(u)}(v)$ and $L'(u)$. If $d'(u, v) \leq d_{vu}$, then there must be a common hub vertex $z \in C(u) \cap C(v)$ such that z is in a shortest path between u and v, and $r(z) > r(u)$. This contradicts with the fact that the rank of u is the highest among all vertices in all shortest paths between u and v. Consequently, $d'(u, v) > d_{vu}$ and HL4GST also inserts (u, d_{vu}) into $L(v)$. Hence, every label in L^{can} is also in L.

We further prove that every label not in L^{can} is not in L as follows. Suppose that there is a label $(u, d_{vu}) \in L(v) \setminus L^{can}(v)$. We observe that (i) u and v are not properly connected; or (ii) $|\{u, v\} \cap D| = 2$; or (iii) the rank of u is not the highest among all vertices in all shortest paths between u and v. If u and v are not properly connected or $|\{u, v\} \cap D| = 2$, then $d_{vu} \geq 2M$, and the generation of the above label will be pruned by GST-customized pruning technique in Line 11 of HL4GST, which means that $(u, d_{vu}) \notin L(v)$. Otherwise, we consider the case where the rank of u is not the highest among all vertices in all shortest paths between u and v as follows. Let z be the vertex with the highest rank in all shortest paths between u and v. We have $(z, d_{zv}) \in L'_{>r(u)}(v)$ and $(z, d_{zu}) \in L'(u)$. As a result, HL4GST computes $d'(u, v)$ as a value no larger than d_{vu} in Line 19, and does not insert (u, d_{vu}) into L. Hence, every label not in L^{can} is not in L. In conclusion, we have $L^{can} = L$. By Theorem 2, this theorem holds. □

Theorem 4. *Given a graph and a corresponding set of 2-hop labels that satisfies the GST-customized 2-hop cover constraint, after a batch of edge insertions and edge weight decreases, the maintained set of labels by* BatchM4GST-Insert *satisfies the above constraint for the updated graph.*

Proof. Consider a pair of vertices s and t. We use $p'(s, t)$ to denote a shortest path between s and t after the change. Let u' be the vertex with the highest rank in all shortest paths between s and t after the change, and $u' \in p'(s, t)$. Also let $d'(s, t)$ be the shortest distance between s and t after the change. Suppose that $|\{s, t\} \cap D| < 2$ and $d'(s, t) < 2M$, *i.e.,* s and t are properly connected after the change.

Let $E_{in} = \{(a_1, b_1), ..., (a_i, b_i)\}$ be the set of changed edges on $p'(s, t)$, *i.e.,* $E_{in} \subseteq E_c$. Notably, E_{in} could be empty. Without loss of generality, suppose that a_x is closer to s than b_x along $p'(s, t)$ for each $x \in [1, i]$, and u' is between (a_k, b_k) and (a_{k+1}, b_{k+1}), as illustrated in Figure S1. Since there are only edge insertions and edge weight decreases, $p'(u', b_k)$ is a shortest path between u' and b_k both before and after the change, and u' has the highest rank in all shortest paths between u' and b_k both before and after the change. We also have $|\{u', b_k\} \cap D| < 2$ and $d'(u', b_k) < 2M$.

Suppose that u' is not a hub of b_k before the maintenance. Since the given set of 2-hop labels satisfies the GST-customized 2-hop cover constraint, to correctly query $d'(u', b_k)$, there must be a vertex z such that $z \in C(u') \cap C(b_k)$, $r(z) > r(u')$, and z is in a shortest path between u' and b_k. This contradicts with the assumption that u' has the highest rank in all shortest paths between u' and b_k. Thus, $(u', d'(b_k, u')) \in L(b_k)$, and further $(u', d'(a_{k+1}, u')) \in L(a_{k+1})$, before the maintenance. BatchM4GST-Insert calls $DIFFUSE$ to re-spread hub u' from b_k to s, and from a_{k+1} to t, along $p'(s, t)$. Thus, $(u', d'(s, u')) \in L(s)$, and similarly $(u', d'(t, u')) \in L(t)$, after the maintenance. Hence, for every pair of properly connected vertices v_i and v_j such that $|\{v_i, v_j\} \cap D| < 2$, there is a common hub vertex $u \in C(v_i) \cap C(v_j)$ on a shortest path between v_i and v_j, *i.e.,* the set of labels maintained by BatchM4GST-Insert still satisfies the GST-customized 2-hop cover constraint. This theorem holds. □

Theorem 5. *Given a graph and a corresponding set of 2-hop labels that satisfies the GST-customized 2-hop cover constraint, after a batch of edge deletions and edge weight increases, the maintained set of labels by* BatchM4GST-Delete *satisfies the above constraint for the updated graph.*

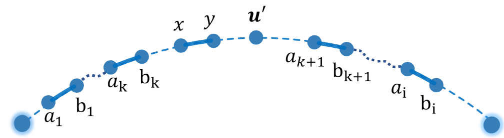

Fig. S1. An illustration for the correctness proofs of the proposed maintenance algorithms.

Proof. We use the notations in the proof of Theorem 4. Consider an arbitrary label $(z, d(z,x)) \in L(x)$ before the change. If this label corresponds to a path that passes through a changed edge, *i.e.*, $d'(z,x) \geq d(z,x)$, then BatchM4GST-Delete must call $SPREAD_1$ to set $L(x)[z]$ to ∞.

Subsequently, consider a pair of vertices s and t such that $|\{s,t\} \cap D| < 2$ and $d'(s,t) < 2M$, *i.e.*, s and t are properly connected after the change. We use $p'(s,t)$ to denote a shortest path between s and t after the change. Let u' be the vertex with the highest rank in all shortest paths between s and t after the change, and $u' \in p'(s,t)$.

Before the maintenance, let y be the vertex farthest to u' along $p'(u', b_k)$ such that u' is a hub of y, and the corresponding label-contained distance value is $d'(u', y)$. Let x be the neighbor of y along $p'(u', b_k)$ such that u' is not a hub of x, or u' is a hub of x but the corresponding label-contained distance value is not $d'(u', x)$.

If u' is a hub of x but the corresponding label-contained distance value is not $d'(u', x)$ before the maintenance, then we must have $d'(u', x) > d(u', x)$. In this case, BatchM4GST-Delete must call $SPREAD_1$ to set $L(x)[u']$ to ∞, and then call $SPREAD_2$ and $SPREAD_3$ to re-spread hub u' from y to x, and ultimately to s, along $p'(s,t)$, *i.e.*, to generate a new label $(u', d'(u', s)) \in L(s)$.

If u' is not a hub of x before the maintenance, then there must be a vertex z such that $z \in C(u') \cap C(x)$, $r(z) > r(u')$, and z is in a shortest path between u' and x, and $u' \in PPR[x,z]$ and $x \in PPR[u', z]$ before the maintenance. Since u' is the vertex with the highest rank in all shortest paths between u' and x after the change, z is not in a shortest path between u' and x after the change. To meet this condition, either $L(u')[z]$ or $L(x)[z]$ increases after the graph change. Thus, BatchM4GST-Delete must call $SPREAD_1$ to set either $L(u')[z]$ or $L(x)[z]$ to ∞. In either case, using the above PPR, it performs $SPREAD_2$ and $SPREAD_3$ to re-spread hub u' from y to x, and ultimately to s, along $p'(s,t)$, *i.e.*, to generate a new label $(u', d'(u', s)) \in L(s)$.

Therefore, in either case, we have $(u', d'(u', s)) \in L(s)$, and similarly $(u', d'(u', t)) \in L(t)$ after the maintenance. Hence, for every pair of properly connected vertices v_i and v_j such that $|\{v_i, v_j\} \cap D| < 2$, there is a common hub vertex $u \in C(v_i) \cap C(v_j)$ on a shortest path between v_i and v_j, *i.e.*, the set of labels maintained by BatchM4GST-Delete still satisfies the GST-customized 2-hop cover constraint. This theorem holds. □

S2. COMPLEXITIES OF ALGORITHMS

A. Complexities of HL4GST

The time complexity of the proposed HL4GST is

$$O\big(|E| \cdot \delta \cdot (\delta + \log |V|)\big)$$

in a single thread environment, where δ is the average number of labels associated with each vertex. The details are as follows. First, the labeling process in Lines 1-14 takes $O(|E| \cdot \delta \cdot (\delta + \log |V|))$ time. The reason is that, for each label inserted into $L'(v)$ in Line 8, it may insert $|N(v)|$ elements into Q in Line 12, and for each element in Q, it takes $O(\log |V|)$ time to pop it out in Line 5 (*e.g.*, using Fibonacci heap [1]) and $O(\delta)$ time to query a distance in Line 6. Second, the sorting process in Lines 15-16 takes $O(|V| \cdot \delta \cdot \log \delta)$ time. Since we generally have $|E| \gg |V|$, the above complexity is covered by that of the labeling process. Third, the cleaning process in Lines 17-21 takes $O(|V| \cdot \delta^2)$ time, given that a distance query that costs $O(\delta)$ is required for cleaning each label. The cost of $O(|V| \cdot \delta^2)$ is also covered by that of the labeling process. On the other hand, the space complexity of HL4GST is $O(|V| \cdot \delta)$ without PPR, and $O(|E| \cdot \delta)$ with PPR.

B. Complexities of BatchM4GST-Insert

The proposed BatchM4GST-Insert has a time complexity of

$$O\big(|E_c| \cdot \delta^2 + \Upsilon \cdot (\log \Upsilon + d_a \cdot \delta)\big)$$

in a single thread environment, where Υ is the number of update labels, and d_a is the average degree of vertices that are associated with these labels. The details are as follows. First, populating

CL in Lines 1-18 takes $O(|E_c| \cdot \delta^2)$ time, since each distance query in Lines 5 and 13 costs $O(\delta)$. Second, $DIFFUSE$ takes $O(\Upsilon \cdot (\log \Upsilon + d_a \cdot \delta))$ time. The reason is that, since $O(|CL|) = O(\Upsilon)$, the initialization steps in Lines 21-23 takes $O(\Upsilon \cdot \log \Upsilon)$ time. Moreover, $DIFFUSE$ pops $O(\Upsilon)$ elements out of the priority queue. Each pop operation takes $O(\log \Upsilon)$ times. After each pop, it searches $O(d_a)$ neighbors, and a distance query that costs $O(\delta)$ may be conducted in each search. On the other hand, due to the cost of PPR, the space complexity of BatchM4GST-Insert is $O(|E| \cdot \delta)$.

C. Complexities of BatchM4GST-Delete

The proposed BatchM4GST-Delete has a time complexity of

$$O\big(|E_c| \cdot \delta + \Upsilon \cdot \big(\log \Upsilon + d_a \cdot \delta + \kappa \cdot (d_a + \delta)\big)\big)$$

in a single thread environment, where κ is the average number of PPR elements of each vertex-hub pair. The details are as follows. First, it pushes labels into AL_1 in Lines 2-8 in $O(|E_c| \cdot \delta)$ time. Then, it performs $SPREAD_1$ in $O(\Upsilon \cdot d_a)$ time, since there are $O(\Upsilon)$ labels deactivated, and each deactivation is followed by $O(d_a)$ neighbor searches. Subsequently, it performs $SPREAD_2$ in $O(\Upsilon \cdot \kappa \cdot (d_a + \delta))$ time, since for each of $O(\Upsilon)$ tuples in AL_2, it checks $O(\kappa)$ PPR elements in Line 19, while checking each PPR element takes $O(d_a + \delta)$ time, due to the cost of $O(d_a)$ for computing $d1(x,t)$ and $d1(t,x)$ in Lines 21 and 28, and the cost of $O(\delta)$ for querying distances in Lines 23 and 30. After that, similar to $DIFFUSE$ in BatchM4GST-Insert, it performs $SPREAD_3$ in $O(\Upsilon \cdot (\log \Upsilon + d_a \cdot \delta))$ time, given that $O(|AL_3|) = O(\Upsilon)$. In the end, due to the cost of PPR, the space complexity of BatchM4GST-Delete is $O(|E| \cdot \delta)$.

D. Comparison between the proposed BatchM4GST and state-of-the-art methods

As discussed above, the proposed batch-friendly label maintenance framework: BatchM4GST contains a new integrated label update process that takes

$$O\big(\Upsilon \cdot (\log \Upsilon + d_a \cdot \delta)\big)$$

time. In comparison, the label update process in state-of-the-art algorithms [2–4] takes

$$O\big(\Upsilon \cdot d_a^2 \cdot \delta\big)$$

time. Since we generally have

$$d_a^2 \cdot \delta \gg \log \Upsilon + d_a \cdot \delta \tag{S1}$$

in practice, BatchM4GST obtains a smaller time cost than state-of-the-art algorithms [2–4]. We present details of the label update process in state-of-the-art algorithms as follows.

We take DecreaseAsyn in [2] as an example to show the label update process in state-of-the-art algorithms. We refer to DecreaseAsyn as InsertAsyn in the main experiments. It maintains 2-hop labels after an edge weight decrease or an edge insertion, and can be parallelly implemented in batch cases [4]. IncreaseAsyn in [2] that deals with an edge weight increase or an edge deletion has a similar label update process with DecreaseAsyn. These two algorithms, including their parallel versions [4], are state-of-the-art 2-hop label maintenance methods.

We show the pseudo code of DecreaseAsyn as Algorithm S1. Suppose that the weight of an edge $(a,b) \in E$ decreases from $w_0(a,b)$ to $w_1(a,b)$. The idea of DecreaseAsyn is to update all outdated label-contained distance values that correspond to paths that pass through the old (a,b), first for labels of a and b, and then neighbors of a and b, and then neighbors of neighbors, *etc.*

DecreaseAsyn first initializes two empty sets: CL^c and CL^n (Line 1). It uses CL^c to record new labels of a and b in Lines 2-17. DecreaseAsyn uses CL^c to record new labels of b as follows. For each $(v, d_{va}) \in L(a)$, if $r(v) \geq r(b)$ (Line 3), which means that v could be a hub of b, then it checks whether the queried distance between v and b is larger than $d_{va} + w_1(a,b)$. If it is, then we can generate a new label $(v, d_{va} + w_1(a,b)) \in L(b)$. Thus, it sets $L(b)[v] = d_{va} + w_1(a,b)$, and pushes $(b, v, d_{va} + w_1)$ into CL^c (Line 5), for iteratively updating more labels in later processes. Otherwise, it conducts the above step when $v \in C(b)$ & $L(b)[v] > d_{va} + w_1(a,b)$ (Line 7), and then inserts v into $PPR[b, h_c]$, and also inserts b into $PPR[v, h_c]$, where h_c is the common hub responsible for the queried distance between v and b. The condition that $v \in C(b)$ & $L(b)[v] > d_{va} + w_1(a,b)$ means that v is already a hub of b, and $L(b)[v]$ is larger than, but should be decreased to, $d_{va} + w_1(a,b)$. This step is necessary for combining DecreaseAsyn and IncreaseAsyn together to deal with fully dynamic cases where edge weights may alternately increase and decrease. After that, it uses CL^c to record new labels of a similarly (Lines 10-17).

Algorithm S1. The DecreaseAsyn algorithm

Input: the original $G_0(V, E_0, w_0)$, the updated $G_1(V, E_1, w_1)$, (a, b), L, PPR
Output: the maintained L and PPR

1: $CL^c = CL^n = \emptyset$
2: **for** each label $(v, d_{va}) \in L(a)$ **do**
3:   **if** $r(v) \geq r(b)$ **then**
4:     **if** $Query(v, b, L) > d_{va} + w_1(a, b)$ **then**
5:       $L(b)[v] = d_{va} + w_1(a, b), CL^c.push((b, v, d_{va} + w_1(a, b)))$
6:     **else**
7:       **if** $v \in C(b)$ & $L(b)[v] > d_{va} + w_1(a, b)$ **then**
8:         $L(b)[v] = d_{va} + w_1(a, b), CL^c.push((b, v, d_{va} + w_1(a, b)))$
9:       $PPR[b, h_c].push(v), PPR[v, h_c].push(b)$
10: **for** each label $(v, d_{vb}) \in L(b)$ **do**
11:   **if** $r(v) \geq r(a)$ **then**
12:     **if** $Query(v, a, L) > d_{vb} + w_1(a, b)$ **then**
13:       $L(a)[v] = d_{vb} + w_1(a, b), CL^c.push((a, v, d_{vb} + w_1(a, b)))$
14:     **else**
15:       **if** $v \in C(a)$ & $L(a)[v] > d_{vb} + w_1(a, b)$ **then**
16:         $L(a)[v] = d_{vb} + w_1(a, b), CL^c.push((a, v, d_{vb} + w_1(a, b)))$
17:       $PPR[a, h_c].push(v), PPR[v, h_c].push(a)$
18: **while** $CL^c \neq \emptyset$ **do** $ProDecrease(CL^c, CL^n), CL^c = CL^n, CL^n = \emptyset$
19: Return L and PPR

Procedure $ProDecrease(CL^c, CL^n)$
20: **for** each $(u, v, d_u) \in CL^c$ **do**
21:   **for** each $u_n \in N(u)$ **do**
22:     **if** $r(v) > r(u_n)$ **then**
23:       **if** $Query(v, u_n, L) > d_{new} = d_u + w_1(u, u_n)$ **then**
24:         $L(u_n)[v] = d_{new}, CL^n.push((u_n, v, d_{new}))$
25:       **else**
26:         **if** $v \in C(u_n)$ & $L(u_n)[v] > d_{new}$ **then**
27:           $L(u_n)[v] = d_{new}, CL^n.push((u_n, v, d_{new}))$
28:         $PPR[u_n, h_c].push(v), PPR[v, h_c].push(u_n)$

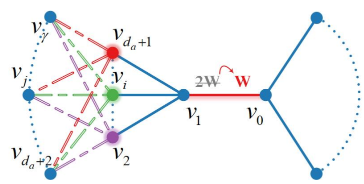

Fig. S2. An example to show the label update process in DecreaseAsyn.

Subsequently, while $CL^c \neq \emptyset$, DecreaseAsyn iteratively uses the $ProDecrease$ procedure to update more labels (Line 18). CL^c and CL^n are the sets of updated labels in the last and current iterations, respectively. To perform these iterations, it sets $CL^c = CL^n$ and $CL^n = \emptyset$ after using $ProDecrease$ in each iteration. In $ProDecrease$, for each $(u, v, d_u) \in CL^c$ (Line 20), it checks each neighbor u_n of u (Line 21). If $r(v) > r(u_n)$, which means that v could be a hub of u_n, then it checks whether the queried distance between v and u_n is larger than $d_{new} = d_u + w_1(u, u_n)$ (Line 23). If it is, then we can generate a new label $(v, d_{new}) \in L(u_n)$. Thus, it updates $L(u_n)[v] = d_{new}$, and pushes (u_n, v, d_{new}) into CL^n (Line 24). Otherwise, it conducts the above step when $v \in C(u_n)$ & $L(u_n)[v] > d_{new}$, and then inserts v into $PPR[u_n, h_c]$, and also inserts u_n into $PPR[v, h_c]$ (Line 28), where h_c is the common hub responsible for the queried distance between v and u_n. The update of PPR is for maintaining 2-hop labels in later edge weight increase or edge deletion cases. In the end, DecreaseAsyn returns the updated L and PPR (Line 19).

After a change, DecreaseAsyn first updates labels of a and b, and then updates labels of neighbors of a and b, and then neighbors of neighbors etc., until no label needs to be updated. The above label update process, *i.e.*, the iterative call of $ProDecrease$, takes $O(\Upsilon \cdot d_a^2 \cdot \delta)$ time. The details are as follows. First, note that, the above label update process updates a label-contained distance value $L(x)[v]$ by spreading hub v from a or b to x in a breadth first search way. This process performs a distance relaxation in Lines 23-24 for each searched edge. As a result, this process may update a label-contained distance value $L(x)[v]$ at most $O(d_a)$ times in the breadth first search process. Thus, this process could perform $O(\Upsilon \cdot d_a)$ label update operations. Each label update operation in an iterative call of $ProDecrease$ induces $O(d_a)$ distance queries in the next iterative call of $ProDecrease$. Since each distance query takes $O(\delta)$ time [5, 6], the above label update process takes $O(\Upsilon \cdot d_a^2 \cdot \delta)$ time. We show an example as follows.

Consider the graph in Figure S2, where $W \gg \Upsilon \gg d_a^2$; $w_0(v_0, v_1) = 2W$; $w_0(v_1, v_i) = 1$ for each $i \in [2, d_a + 1]$; there is a simple path between v_i and v_j for every pair of $i \in [2, d_a + 1]$ and $j \in [d_a + 2, \Upsilon]$, and this path contains $i - 1$ edges and has a total weight of $d_a - i + 2$. Suppose that the weight of (v_0, v_1) decreases from $2W$ to W. DecreaseAsyn maintains labels as follows. Initially, it updates $L(v_1)[v_0] = W$ and $L(v_i)[v_0] = W + 1$ for each $i \in [2, d_a + 1]$. Subsequently, for a certain $j \in [d_a + 2, \Upsilon]$, it sequentially updates $L(v_j)[v_0] = W + 1 + d_a - i + 2$ through the path between v_i and v_j for each $i \in [2, d_a + 1]$. As a result, they update label-contained distance values $O(\Upsilon \cdot d_a)$ times. Since each label update in an iterative call of $ProDecrease$ induces $O(d_a)$ distance queries in the next iterative call, DecreaseAsyn conducts $O(\Upsilon \cdot d_a^2)$ distance queries. Since each distance query takes $O(\delta)$ time, the label update process in DecreaseAsyn costs $O(\Upsilon \cdot d_a^2 \cdot \delta)$. In comparison, as discussed before, the integrated label update process in the proposed BatchM4GST has a smaller time cost of $O(\Upsilon \cdot (\log \Upsilon + d_a \cdot \delta))$.

REFERENCES FOR THE SUPPLEMENT

1. M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," Journal of the ACM **34**, 596–615 (1987).
2. M. Zhang, L. Li, W. Hua, and X. Zhou, "Efficient 2-hop labeling maintenance in dynamic small-world networks," in *2021 IEEE 37th International Conference on Data Engineering,* (IEEE, 2021), pp. 133–144.
3. M. Zhang, L. Li, and X. Zhou, "An experimental evaluation and guideline for path finding in weighted dynamic network," Proc. VLDB Endow. **14**, 2127–2140 (2021).
4. M. Zhang, L. Li, G. Trajcevski, A. Zufle, and X. Zhou, "Parallel hub labeling maintenance with high efficiency in dynamic small-world networks," IEEE Transactions on Knowl. Data Eng. (2023).
5. Y. Li, L. H. U, M. L. Yiu, and N. M. Kou, "An experimental study on hub labeling based shortest path algorithms," Proc. VLDB Endow. **11**, 445–457 (2017).
6. W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Distance labeling: on parallelism, compression, and ordering," The VLDB J. **31**, 129–155 (2021).