2024

算法大作业报告

小组成员：

王明硕 任勇宁 刘一辰

一、作业要求：

查询图中两个顶点之间的最短距离的常见的方法有 BFS，Dijkstra 等。这两种方法通过**实时遍历**图上的点和边来获取两点之间的最短距离，因此在规模较大的图上用时较长。为解决这一问题，2-hop 索引方法【1】被提出，其核心思想是**事先计算一部分顶点之间的距离**，将其**储存在索引之中**，并在查询最短距离时，利用**最短路径的最优子结构特性**，使用两个储存的最短距离**组合出来需要查询的最短距离**（比如，储存 s-h、h-t 的最短距离，其中 h 在 s-t 的最短路径上，则 s-h、h-t 的最短距离之和为 s-t 的最短距离）。

在现实场景下，社交网络、知识图谱等图的结构会不断变化。当**图的结构发生变化**时，如何高效地**维护/修改 2-hop 索引**，以便使用 2-hop 索引准确地查询变化后的图中的最短距离，具有重要的现实作用。【2】中描述了前沿的 **2-hop 索引动态维护算法**。本次大作业将给出上述算法的不完整 C++代码，要求同学们补全这些代码，使得相应的测试程序可以无 bug 地运行。

二、论文阅读与伪代码分析：

（一）论文一：

论文一主要介绍了引入剪枝操作的 BFS，其使用的为无权无向图，主要是介绍了“2-hop 标签”的概念以及作用，来为论文二做铺垫。

这里提到了用到的符号。

Notation	Description
$G = (V, E)$	A graph
n	Number of vertices in graph G
m	Number of edges in graph G
$N_G(v)$	Neighbors of vertex v in graph G
$d_G(u, v)$	Distance between vertex u and v in graph G
$P_G(u, v)$	Set of all the vertices on the shortest paths between vertex u and v in graph G

由于背景为无权图，一定满足三角不等式，当 v 出现在 s 和 t 的最短路径上时（即 v 在集合 P 中），不等式取等号。

$$d_G(s,t) \leq d_G(s,v) + d_G(v,t), \quad (1)$$
$$d_G(s,t) \geq |d_G(s,v) - d_G(v,t)|. \quad (2)$$

We define $P_G(s,t) \subseteq V$ as the set of all vertices on the shortest paths between vertices s and t. In other words,

$$P_G(s,t) = \{v \in V \mid d_G(s,v) + d_G(v,t) = d_G(s,t)\}.$$

QUERY 函数是在 L(s)和 L(t)的交集中所有的顶点 v，找出最小的距离。根据最短路径最优子结构特性，可以组合出 s 和 t 的最短路径，这也是 2-hop 的核心思想。

$$\text{QUERY}(s,t,L) = \min\{\delta_{vs} + \delta_{vt} \mid (v, \delta_{vs}) \in L(s), (v, \delta_{vt}) \in L(t)\}.$$

Algorithm 1 Pruned BFS from $v_k \in V$ to create index L'_k.

1: **procedure** PRUNEDBFS(G, v_k, L'_{k-1})
2: $Q \leftarrow$ a queue with only one element v_k.
3: $P[v_k] \leftarrow 0$ and $P[v] \leftarrow \infty$ for all $v \in V(G) \setminus \{v_k\}$.
4: $L'_k[v] \leftarrow L'_{k-1}[v]$ for all $v \in V(G)$.
5: **while** Q is not empty **do**
6: Dequeue u from Q.
7: **if** QUERY(v_k, u, L'_{k-1}) $\leq P[u]$ **then**
8: **continue**
9: $L'_k[u] \leftarrow L'_{k-1}[u] \cup \{(v_k, P[v_k])\}$
10: **for all** $w \in N_G(v)$ s.t. $P[w] = \infty$ **do**
11: $P[w] \leftarrow P[u] + 1$.
12: Enqueue w onto Q.
13: **return** L'_k

Algorithm 2 Compute a 2-hop cover index by pruned BFS.

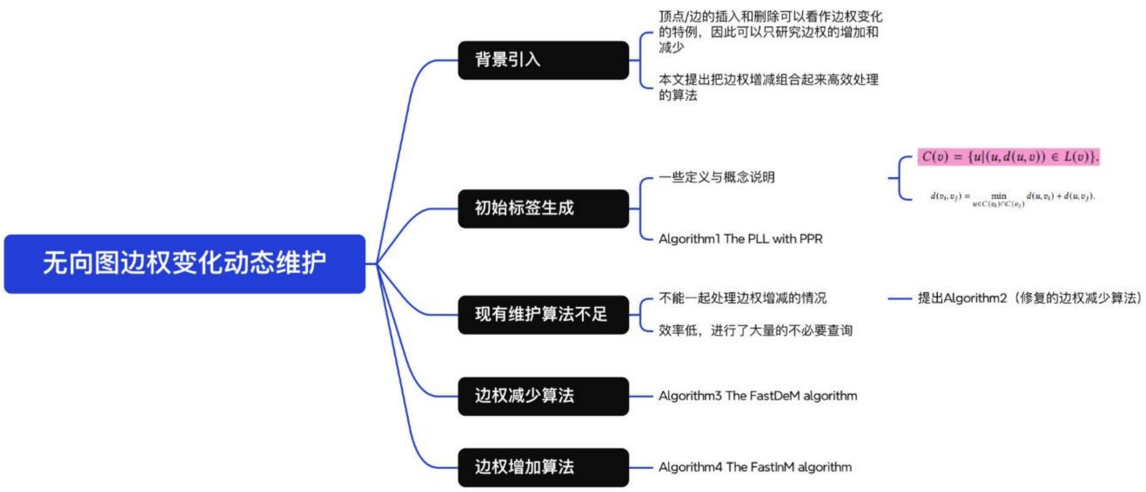
1: **procedure** PREPROCESS(G)
2: $L'_0[v] \leftarrow \emptyset$ for all $v \in V(G)$.
3: **for** $k = 1, 2, \ldots, n$ **do**
4: $L'_k \leftarrow$ PRUNEDBFS(G, v_k, L'_{k-1})
5: **return** L'_n

对伪代码的分析：

主体框架为广度优先搜索，对每个节点执行广度优先搜索，就形成了每个顶点的标签。不同之处在于加入了 purn 剪枝操作。具体来说为 Algorithm 1 的 7-9 行。如果能用现有标签组合出未求出的最短距离，则进行剪枝。最终该算法能生成每个顶点的二跳标签。

（二）论文二：

论文二的整体框架如下：



下面对主要部分做分析：

1. 初始标签生成部分：

这部分承接了论文一的 2-hop 标签的思想。变化之处在于：一方面把无权图变成了有权图（因此查询最短距离时的主体算法也从 BFS 变成了 Dijkstra），另一方面引入了 PPR 这一结构，便于快速识别受到边权增加影响的标签来提高维护效率。

下面分析伪代码：

Algorithm 1 The PLL algorithm incorporated with PPR

Input: a graph $G(V, E, w)$ **Output:** L and PPR
1: Initialize $L = PPR = \emptyset$
2: **for** each sorted vertex $u \in V$ **do**
3: Initialize $Q = \emptyset$, $d(u) = 0$, $d(v) = NIL$ for each $v \in V \setminus u$
4: Insert u into Q with the priority value of $d(u)$
5: **while** $Q \neq \emptyset$ **do**
6: Pop v out of Q with the priority value of $d(v)$
7: **if** $r(u) \geq r(v)$ **then**
8: **if** $Query(u, v, L) \leq d(v)$ **then**
9: $PPR[v, h_c].push(u)$, $PPR[u, h_c].push(v)$; Continue to Line 5
10: Insert $(u, d(v))$ into $L(v)$
11: **for** each vertex $x \in N(v)$ **do**
12: **if** $d(x) == NIL$ **then**
13: Insert x into Q with the priority value of $d(x) = d(v) + w(x, v)$
14: **else if** $d(x) > d(v) + w(x, v)$ **then**
15: Update x in Q with the priority value of $d(x) = d(v) + w(x, v)$
16: Return L and PPR

主体部分是 Dijkstra 算法（line 2：对每个顶点用一次）。这里遍历顺序为顶点度大的先遍历（line 2），因为度大的更可能出现在其他顶点的最短路径上（成为 hub），这样后续便于剪枝来提高效率。剔除 Dijkstra 的主体框架后，我们看到创新的主要在 8-10 行：这里是生成 L 和 PPR 的部分，在 query 结果小于等于 d 时（hc 在 u 和 v 的最短路径上），更新 PPR，不更新 L；否则更新 L（利用 Dijkstra 查询出的距离，而不是 2-hop）。这里之所以这样操作是因为 query 出的最短距离不仅包含了距离信息，还包含了路径上的顶点信息，利用 PPR 就可以把路径上的顶点信息保存下来，这是 L 做不到的。第 7 行的条件可以防止重复查询（因为遍历时是按照度的大小遍历的）。

2.边权减少维护算法:

Algorithm 3 The FastDeM algorithm

Input: the updated graph $G(V, E, w)$, L, PPR, (a, b) // $w_0 > w_1$
Output: the maintained L and PPR
1: $CL^c = \emptyset$; Conduct Lines 2-17 of Algorithm 2 without updating L
2: $DIFFUSE(CL^c)$
3: Return L and PPR

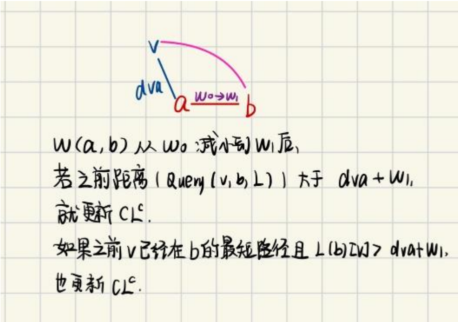
Procedure $DIFFUSE(CL^c)$
4: **for** each $(u, v, d_u) \in CL^c$ **do**
5: $Dis[u] = d_u$, $Dis[s] = -1\ \forall s \in V \setminus u$, $Q = \{(u \mid d_u)\}$
6: **while** $Q \neq \emptyset$ **do**
7: Pop $(x \mid d_x)$ out of Q, $L(x)[v] = d_x$
8: **for** each $x_n \in N(x)$ such that $r(v) > r(x_n)$ **do**
9: **if** $Dis[x_n] == -1$ **then** $Dis[x_n] = Query(x_n, v, L)$
10: **if** $Dis[x_n] > d_x + w(x_n, x)$ **then**
11: $Dis[x_n] = d_x + w(x_n, x)$; Insert (or update) $(x_n \mid Dis[x_n]) \in Q$
12: **else**
13: **if** $v \in C(x_n)$ & $\min\{L(x_n)[v], Q(x_n)\} > d_{new} = d_x + w(x_n, x)$ **then**
14: Insert (or update) $(x_n \mid d_{new}) \in Q$
15: $PPR[x_n, h_c].push(v)$, $PPR[v, h_c].push(x_n)$

2: **for** each label $(v, d_{va}) \in L(a)$ **do**
3: **if** $r(v) \geq r(b)$ **then**
4: **if** $Query(v, b, L) > d_{va} + w_1$ **then** // $w(a, b) = w_1$
5: ~~$L(b)[v] = d_{va} + w_1$~~, $CL^c.push((b, v, d_{va} + w_1))$
6: **else**
7: **if** $v \in C(b)$ & $L(b)[v] > d_{va} + w_1$ **then**
8: ~~$L(b)[v] = d_{va} + w_1$~~, $CL^c.push((b, v, d_{va} + w_1))$
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** $Query(v, a, L) > d_{vb} + w_1$ **then**
13: $L(a)[v] = d_{vb} + w_1$, $CL^c.push((a, v, d_{vb} + w_1))$
14: **else**
15: **if** $v \in C(a)$ & $L(a)[v] > d_{vb} + w_1$ **then**
16: $L(a)[v] = d_{vb} + w_1$, $CL^c.push((a, v, d_{vb} + w_1))$
17: $PPR[a, h_c].push(v)$, $PPR[v, h_c].push(a)$

算法步骤即 Algorithm 3 的 1-3 行。思想就是，既然 w（a，b）变化了，就以 a 和 b 为中心点开始遍历。

先执行算法 2 的 2-17 行（上图给出），但是不更新 L。对 a 和 b 做遍历时是一个对称

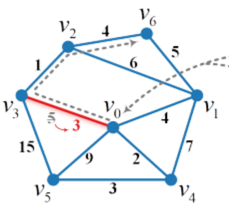
的过程。分出 if else 是因为 v 可能在 b 的 label 里也可能不在。对于 v 已经在 L（b）的情况，还需要更新 PPR。



接下来再执行 DIFFUSE:

该部分更新 L 和 PPR。利用刚刚得到的 CLc 中记录的信息来更新。利用优先队列和距离数组，DIFFUSE 确保所有过时的标签都能通过更新的边直接传播，以便在后续边权重增加时进行进一步的维护。

比如下图，当 v0 和 v3 的权重从 5 减小到 3，CLc 会存入（v3，v0, 3）。执行 DIFFUSE 时，一开始队列里是（v3,3），这样就会把 L(v3)[v0]更新为 3，即（v0,3）。



$L(v_0)$:	$(v_0, 0)$			
$L(v_1)$:	$(v_0, 4)$	$(v_1, 0)$		
$L(v_2)$:	$(v_0, 4)$	$(v_1, 6)$	$(v_2, 0)$	
$L(v_3)$:	$(v_0, 3)$	$(v_1, 7)$	$(v_2, 1)$	$(v_3, 0)$
$L(v_4)$:	$(v_0, 2)$	$(v_4, 0)$		
$L(v_5)$:	$(v_0, 5)$	$(v_4, 3)$	$(v_5, 0)$	
$L(v_6)$:	$(v_0, 8)$	$(v_1, 5)$	$(v_2, 4)$	$(v_6, 0)$

Figure 4: An example of FastDeM.

3.边权增加算法：

Algorithm 4 The FastInM algorithm

```
Input: the updated graph G(V, E, w), L, PPR, (a, b), w0 // w0 < w1
Output: the maintained L and PPR
AL1 = AL2 = AL3 = ∅
for each label (v, d_va) ∈ L(a) do
    if r(v) ≥ r(b) & (v, d_va + w0) ∈ L(b) then AL1.push((b, v, d_va + w0))
for each label (v, d_vb) ∈ L(b) do
    if r(v) ≥ r(a) & (v, d_vb + w0) ∈ L(a) then AL1.push((a, v, d_vb + w0))
SPREAD1(AL1, AL2), SPREAD2(AL2, AL3), SPREAD3(AL3)
Return L and PPR

Procedure SPREAD1(AL1, AL2)
for each (u, v, d′) ∈ AL1 do
    Queue = {(u, d′)}
    while Queue ≠ ∅ do
        Queue.pop((x, d_x)), L(x)[v] = ∞, AL2.push((x, v))
        for each x_n ∈ N(x) such that r(v) > r(x_n) do
            if (v, d_x + w(x, x_n)) ∈ L(x_n) then Queue.push((x_n, d_x + w(x, x_n)))

Procedure SPREAD2(AL2, AL3)
for each (x, y) ∈ AL2 do
    for each t ∈ PPR[x, y] ∪ y do
        if r(t) > r(x) then
            d1(x, t) = min_{x_n∈N(x)} {L(x_n)[t] + w(x, x_n)}
            if Query(x, t, L) > d1(x, t) then AL3.push((x, t, d1(x, t)))
            else PPR[x, h_c].push(t), PPR[t, h_c].push(x)
        if r(x) > r(t) then
            d1(t, x) = min_{t_n∈N(t)} {L(t_n)[x] + w(t, t_n)}
            if Query(t, x, L) > d1(t, x) then AL3.push((t, x, d1(t, x)))
            else PPR[t, h_c].push(x), PPR[x, h_c].push(t)

Procedure SPREAD3(AL3)
for each (u, v, d_u) ∈ AL3 do
    if Q(u, v, L) ≤ d_u then PPR[u, h_c].push(v), PPR[v, h_c].push(u); Continue
    Dis[u] = d_u, Dis[s] = −1 for each s ∈ V \ u, Q = {(u | d_u)}
    while Q ≠ ∅ do
        Pop (x | d_x) out of Q, L(x)[v] = min(d_x, L(x)[v])
        for each x_n ∈ N(x) such that r(v) > r(x_n) do
            if Dis[x_n] == −1 then Dis[x_n] = Query(x_n, v, L)
            if Dis[x_n] > d_x + w(x_n, x) then
                Dis[x_n] = d_x + w(x_n, x); Insert (or update) (x_n | Dis[x_n]) ∈ Q
            else PPR[x_n, h_c].push(v), PPR[v, h_c].push(x_n)
```

下面这段话介绍了该算法的主要思想：

The FastInM **algorithm:** The algorithm inputs the updated graph $G(V, E, w)$, L, PPR, (a, b) and w_0. First, it initializes three empty sets AL_1, AL_2 and AL_3 (Line 1). It uses AL_1 to store labels in $L(a) \cup L(b)$ that correspond to paths that pass through (a, b). Based on AL_1, it finds all labels that correspond to paths that pass through (a, b), and then deactivates these labels, and also uses AL_2 to record these labels. After the deactivation, some originally pruned labels can be newly produced. It uses AL_3 to record these labels, and uses these labels as starting points to generate more labels.

AL1 能记录所有经过（a，b）的路径，停用这些标签并记录到 AL2 中。停用后，之前被修剪的标签可以重新生成，用 AL3 记录，并且通过这些标签作为起点来生成更多标签。

下图是是整体框架：

先填充 AL1，如果 v 到 b 的最短路径经过 a，存入 AL1；同理，如果 v 到 a 的最短路径经过 b，存入 AL1。这样 AL1 就记录了所有经过 (a,b) 的最短路径。spread1 基于 AL1 生成 AL2，spread2 基于 AL2 生成 AL3，spread3 基于 AL3 生成新标签。

Input: the updated graph $G(V, E, w)$, L, PPR, (a, b), w_0 // $w_0 < w_1$
Output: the maintained L and PPR

1: $AL_1 = AL_2 = AL_3 = \emptyset$
2: **for** each label $(v, d_{va}) \in L(a)$ **do**
3: **if** $r(v) \geq r(b)$ & $(v, d_{va} + w_0) \in L(b)$ **then** $AL_1.push((b, v, d_{va} + w_0))$
4: **for** each label $(v, d_{vb}) \in L(b)$ **do**
5: **if** $r(v) \geq r(a)$ & $(v, d_{vb} + w_0) \in L(a)$ **then** $AL_1.push((a, v, d_{vb} + w_0))$
6: $SPREAD_1(AL_1, AL_2)$, $SPREAD_2(AL_2, AL_3)$, $SPREAD_3(AL_3)$
7: Return L and PPR

下面对三个 spread 的关键点作说明：

Procedure $SPREAD_1(AL_1, AL_2)$
8: **for** each $(u, v, d') \in AL_1$ **do**
9: $Queue = \{(u, d')\}$
10: **while** $Queue \neq \emptyset$ **do**
11: $Queue.pop((x, d_x))$, $L(x)[v] = \infty$, $AL_2.push((x, v))$
12: **for** each $x_n \in N(x)$ such that $r(v) > r(x_n)$ **do**
13: **if** $(v, d_x + w(x, x_n)) \in L(x_n)$ **then** $Queue.push((x_n, d_x + w(x, x_n)))$

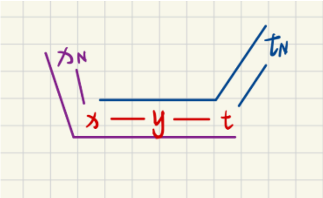
spread1 主要是通过将 L（x）[v]置为无穷来停用与（a，b）相关的标签，并将这些标签存入 AL2（对应于 line 11）。其余操作类似于广搜，实现遍历。

Procedure $SPREAD_2(AL_2, AL_3)$
14: **for** each $(x, y) \in AL_2$ **do**
15: **for** each $t \in PPR[x, y] \cup\ y$ **do**
16: **if** $r(t) > r(x)$ **then**
17: $d1(x, t) = \min_{x_n \in N(x)} \{L(x_n)[t] + w(x, x_n)\}$
18: **if** $Query(x, t, L) > d1(x, t)$ **then** $AL_3.push((x, t, d1(x, t)))$
19: **else** $PPR[x, h_c].push(t)$, $PPR[t, h_c].push(x)$
20: **if** $r(x) > r(t)$ **then**
21: $d1(t, x) = \min_{t_n \in N(t)} \{L(t_n)[x] + w(t, t_n)\}$
22: **if** $Query(t, x, L) > d1(t, x)$ **then** $AL_3.push((t, x, d1(t, x)))$
23: **else** $PPR[t, h_c].push(x)$, $PPR[x, h_c].push(t)$

spread2 在找出依赖 L（x）[y]生成的标签，这些标签需要被重新生成，这个时候就需要用到 PPR。

it enumerates each $t \in PPR[x, y] \cup y$ (Line 15). If $t \in PPR[x, y]$, then either $y \in C(x) \cap C(t)$ is responsible for originally pruning the spread of hub t from a neighbor of x to x (*i.e.*, $r(t) > r(x)$), or y is responsible for originally pruning the spread of hub x from a neighbor of t to t (*i.e.*, $r(x) > r(t)$). If $t = y$ (*i.e.*, $r(t) > r(x)$), then it may be possible to update $L(x)[y]$ by re-spreading hub y from a neighbor of x to x. Based on these analyses, it identifies

上图提到的就是 PPR 的作用原理，意思大致是下图所画：



Procedure $SPREAD_3(AL_3)$

24: **for** each $(u, v, d_u) \in AL_3$ **do**
25:     **if** $Q(u, v, L) \leq d_u$ **then** $PPR[u, h_c].push(v)$, $PPR[v, h_c].push(u)$; Continue
26:     $Dis[u] = d_u$, $Dis[s] = -1$ for each $s \in V \setminus u$, $Q = \{(u \mid d_u)\}$
27:     **while** $Q \neq \emptyset$ **do**
28:         Pop $(x \mid d_x)$ out of Q, $L(x)[v] = \min(d_x, L(x)[v])$
29:         **for** each $x_n \in N(x)$ such that $r(v) > r(x_n)$ **do**
30:             **if** $Dis[x_n] == -1$ **then** $Dis[x_n] = Query(x_n, v, L)$
31:             **if** $Dis[x_n] > d_x + w(x_n, x)$ **then**
32:                 $Dis[x_n] = d_x + w(x_n, x)$; Insert (or update) $(x_n \mid Dis[x_n]) \in Q$
33:             **else** $PPR[x_n, h_c].push(v)$, $PPR[v, h_c].push(x_n)$

spread3 基本和前面 decrease 算法的 diffuse 一致，有以下两点不同：第一，spread3 首先检查查询的 u 和 v 之间的距离是否不大于 du（第 25 行），因为新生成的标签可能会修剪其他标签的生成；第二，当 Dis[xn] ≤ dx +w（xn，x）时，spread3 不会以 dx + w（xn，x）的优先级将 xn 的元素插入（或更新）到 Q 中，因为所有过时的标签都已在 spread1 中停用。

三、C++代码补全与正确性测试：

这部分需要补全四个函数，分别对应着伪代码里的 diffuse，spread1、2、3。基本都是直接根据伪代码翻译即可。需要把定义好的变量与结构以及查询函数了解清楚。需要认真阅读已有的代码，学到一些处理技巧，比如判断浮点数相等不能直接用等号。另外，这几段代码的很多操作是可以互相借鉴的，比如 diffuse 与 spread3 基本是一样的。

下面是补全的代码：

DIFFUSE：

```cpp
void DIFFUSE(graph_v_of_v_idealID& instance_graph, vector<vector<two_hop_label_v1>>* L, PPR_type* PPR, std::vector<affected_label>& CL,
    ThreadPool& pool_dynamic, std::vector<std::future<int>>& results_dynamic) {
    int vex_num = instance_graph.size();
    double* Dis = new double[vex_num];
    for (auto affected_l : CL) //{u,v,du}
    {
            pair<weightTYPE, int> dis_vex = graph_hash_of_mixed_weighted_two_hop_v1_extract_distance_no_reduc2(*L, affected_l.first, affected_l.second);

            fill_n(Dis, vex_num, -1);
            Dis[affected_l.first] = affected_l.dis;
            PLL_dynamic_node_for_sp node = { affected_l.first, affected_l.dis }; //{vertex, val}
            boost::heap::fibonacci_heap<PLL_dynamic_node_for_sp> Q;
            vector<graph_hash_of_mixed_weighted_HL_PLL_v1_handle_t_for_sp> Q_handles(vex_num);
            Q_handles[affected_l.first] = Q.push(node);
            bool exsit[vex_num]={false};
            exsit[affected_l.first]=true;
            while (Q.size() > 0)
            {
                node = Q.top();
                Q.pop();
                exsit[node.vertex]=false;
                int x = node.vertex;
                weightTYPE dx = node.priority_value;
                insert_sorted_two_hop_label((*L)[x],affected_l.second,dx);
                for (auto xn:instance_graph[x])
                {
                    if (affected_l.second < xn.first)
                    {
                        if (Dis[xn.first] == -1)
                        {
                            Dis[xn.first] = graph_hash_of_mixed_weighted_two_hop_v1_extract_distance_no_reduc(*L, xn.first, affected_l.second);
                        }
                        double newVal = dx + graph_v_of_v_idealID_edge_weight(instance_graph, xn.first, x);
                        if (Dis[xn.first]>newVal)
                        {
                            Dis[xn.first]=newVal;
                            node={xn.first, Dis[xn.first]};
                            if (exsit[xn.first])
```

```cpp
                        {
                            Q.update(Q_handles[xn.first], node);
                        }
                        else Q_handles[xn.first]=Q.push(node), exsit[node.vertex]=true;
                    }
                    else
                    {
                        auto search_result = search_sorted_two_hop_label((*L)[xn.first], affected_l.second);
                        if(search_result !=MAX_VALUE){
                            weightTYPE min_val = std::min(search_result, Dis[xn.first]);
                            if (min_val > newVal)
                            {
                                Dis[xn.first]=newVal;
                                node={xn.first, Dis[xn.first]};
                                if (exsit[xn.first])
                                {
                                Q.update(Q_handles[xn.first], node);
                                }
                                else Q_handles[xn.first]=Q.push(node), exsit[node.vertex]=true;
                            }
                        }
                        PPR_insert(*PPR, xn.first, dis_vex.second, affected_l.second);
                        PPR_insert(*PPR, affected_l.second, dis_vex.second, xn.first);
                    }
                }
            }
        }
    }
    delete[] Dis;
}
```

SPREAD1：

```cpp
void SPREAD1(graph_v_of_v_idealID& instance_graph, vector<vector<two_hop_label_v1>>* L,
    std::vector<affected_label>& al1, std::vector<pair_label>* al2, ThreadPool& pool_dynamic, std::vector<std::future<int>>& results_dynamic) {

    for (auto& label : al1) {
        std::queue<std::pair<int, weightTYPE>> Queue;
        Queue.push(std::make_pair(label.first, label.dis)); // 将（u, du）入队
        while (!Queue.empty()) {
            auto [x, dx] = Queue.front();
            Queue.pop();
            //统一改成使用函数改值(其实这里不用insert)
            //insert_sorted_two_hop_label((*L)[x], label.second, std::numeric_limits<weightTYPE>::max()); // 将（v, du）插入到 L(x) 中
            al2->emplace_back(x, label.second); // 添加到 al2 中
            for (auto xn : instance_graph[x]) { // 遍历 x 的邻居 xn
                if (label.second < xn.first) { // 如果标签的优先级高于 xn
                    //weightTYPE new_distance = dx + xn.second; // 计算新距离
                    double new_distance = dx + graph_v_of_v_idealID_edge_weight(instance_graph, x, xn.first);
                    // if ((*L)[xn.first][label.second].distance == new_distance) { // 如果新距离在 L(xn) 中
                    //  Queue.push(std::make_pair(xn.first, new_distance)); // 将（xn, 新距离）入队
                    // }
                    double search_res=search_sorted_two_hop_label((*L)[xn.first], label.second);
                    if (search_res!=std::numeric_limits<weightTYPE>::max() && abs(search_res - new_distance) < 1e-5) { // 如果新距离在 L(xn) 中
                        Queue.push(std::make_pair(xn.first, new_distance)); // 将（xn, 新距离）入队
                    }
                }
            }
        }
    }
}
```

SPREAD2:

```cpp
void SPREAD2(graph_v_of_v_idealID& instance_graph, vector<vector<two_hop_label_v1>>* L, PPR_type* PPR,
    std::vector<pair_label>& al2, std::vector<affected_label>* al3, ThreadPool& pool_dynamic, std::vector<std::future<int>>& results_dynamic) {
    std::vector<pair_label>::iterator it;
    for(it=al2.begin();it!=al2.end();it++)//for each (x,y)属于AL2
    {
        std::vector<int> PPR_xyandy = PPR_retrieve(*PPR,it->first,it->second);
        PPR_xyandy.push_back(it->second);
        for(std::vector<int>::iterator iIter = PPR_xyandy.begin();iIter!=PPR_xyandy.end();iIter++)
        {
            if(*iIter < it->first)//r(t)>r(x)
            {
                double d1 = std::numeric_limits<weightTYPE>::max();//inf
                for(std::vector<std::pair<int, double>>::iterator it2 = instance_graph[it->first].begin();it2!=instance_graph[it->first].end();it2++)
                {
                    //double d1_new =(*L)[it2->first][*iIter].distance+it2->second;
                    double d1_new = search_sorted_two_hop_label((*L)[it2->first], *iIter) + graph_v_of_v_idealID_edge_weight(instance_graph, it2->first, it->first);
                    d1=min(d1, d1_new);
                    // if(d1_new<d1)
                    // {
                    //  d1 =d1_new;
                    // }
                }// 计算d1(x,t)
                auto hc =graph_hash_of_mixed_weighted_two_hop_v1_extract_distance_no_reduc2(*L, it->first, *iIter);
                if(hc.first>d1)
                {
                    (*al3).push_back(affected_label(it->first,*iIter,d1));
                }
                else
                {
                    PPR_insert(*PPR,it->first,hc.second,*iIter);
                    PPR_insert(*PPR,*iIter,hc.second,it->first);
                }
            }
```

```cpp
            // else if(*iIter,it->first) ???
            else if(*iIter>it->first)
            {
                // double d1 = 1e12;???
                double d1 = std::numeric_limits<weightTYPE>::max();
                for(std::vector<std::pair<int, double>>::iterator it2 = instance_graph[*iIter].begin();it2!=instance_graph[*iIter].end();it2++)
                {
                    //double d1_new =(*L)[it2->first][it->first].distance+it2->second;
                    double d1_new = search_sorted_two_hop_label((*L)[it2->first], it->first) + graph_v_of_v_idealID_edge_weight(instance_graph, *iIter, it2->first);
                    d1=min(d1, d1_new);
                    // if(d1_new<d1)
                    // {
                    //  d1 =d1_new;
                    // }
                }
                pair<weightTYPE, int> hc = graph_hash_of_mixed_weighted_two_hop_v1_extract_distance_no_reduc2(*L,*iIter,it->first);
                if(hc.first>d1)
                {
                    (*al3).push_back(affected_label(*iIter,it->first,d1));
                }
                else
                {
                    PPR_insert(*PPR,it->first,hc.second,*iIter);
                    PPR_insert(*PPR,*iIter,hc.second,it->first);
                }
            }
        }//for each t属于PPR[x,y]并y
    }//遍历al2
}
```

SPREAD3:

```cpp
void SPREAD3(graph_v_of_v_idealID& instance_graph, vector<vector<two_hop_label_v1>>* L, PPR_type* PPR, std::vector<affected_label>& al3,
    ThreadPool& pool_dynamic, std::vector<std::future<int>>& results_dynamic) {
    int vex_num=instance_graph.size();
    double *Dis=new double[vex_num];
    for (auto affected_l : al3)//{u,v,du} {first, second, dis}
    {
        pair<weightTYPE,int> dis_vex= graph_hash_of_mixed_weighted_two_hop_v1_extract_distance_no_reduc2(*L, affected_l.first, affected_l.second);
        if (dis_vex.first <= affected_l.dis)
        {
            PPR_insert(*PPR, affected_l.first, dis_vex.second, affected_l.second);
            PPR_insert(*PPR, affected_l.second, dis_vex.second, affected_l.first);
        }
        else
        {
            fill_n(Dis, vex_num, -1);
            Dis[affected_l.first]=affected_l.dis;
            PLL_dynamic_node_for_sp node={affected_l.first, affected_l.dis};//{vertex, val}
            boost::heap::fibonacci_heap<PLL_dynamic_node_for_sp> Q;
            vector<graph_hash_of_mixed_weighted_HL_PLL_v1_handle_t_for_sp> Q_handles(vex_num);
            Q_handles[affected_l.first] = Q.push(node);//Q={u,du}
            bool exsit[vex_num]={false};
            exsit[affected_l.first]=true;
            while (Q.size()>0)
            {
                node=Q.top();//pop (x|dx) out of Q
                Q.pop();
                exsit[node.vertex]=false;
                int x=node.vertex;
                weightTYPE dx=node.priority_value;
                //L[x][v]=min(dx,L[x][v])
                //(*L)[x][affected_l.second].distance=min((*L)[x][affected_l.second].distance, dx);
                double tmp=search_sorted_two_hop_label((*L)[x], affected_l.second);
```

```cpp
                if (dx<tmp)
                {
                    insert_sorted_two_hop_label((*L)[x], affected_l.second, dx);
                }
                for (auto xn:instance_graph[x])//for each xn in N(x)
                {
                    if (affected_l.second<xn.first)
                    {
                        if (Dis[xn.first]==-1)
                        {
                            Dis[xn.first]=graph_hash_of_mixed_weighted_two_hop_v1_extract_distance_no_reduc(*L, xn.first, affected_l.second);
                        }
                        double newVal=dx+graph_v_of_v_idealID_edge_weight(instance_graph, xn.first, x);
                        if (Dis[xn.first]>newVal)
                        {
                            Dis[xn.first]=newVal;
                            node={xn.first, Dis[xn.first]};
                            if (exsit[xn.first])
                            {
                                Q.update(Q_handles[xn.first], node);
                            }
                            else Q_handles[xn.first]=Q.push(node), exsit[node.vertex]=true;
                        }
                        else
                        {
                            PPR_insert(*PPR, xn.first, dis_vex.second, affected_l.second);
                            PPR_insert(*PPR, affected_l.second, dis_vex.second, xn.first);
                        }
                    }
                }
            }
        }
    }
    delete[] Dis;
}
```

下面来说明 debug 的过程：

为了得到代码具体的行为，我们将图与输入点固定位论文中的实例。我们将 generate_new_graph 设置为 0 并且将 simple_iterative_tests.txt 改变为我们根据论文实例定义的图来固定图。通过改变 graph_change_and_label_maintenance 函数中 while 循环内部被注释的 if 部分来模拟论文实例的边权变化。通过 mm.print_L()，mm.print_PPR()得到 L 以及 PPR 的变化，与论文中实例对比，最终在原图上能够得到正确的答案，并能扩展到更大的随机图上，运行出正确结果。

正确性：最终代码可以跑通，跑几百次可能会出现一次错误，基本可以实现动态维护功能。

四、小组成员分工：

前期：小组成员共同阅读论文与代码

补全代码：王明硕主要负责 decrease，任勇宁和刘一辰主要负责 increase

debug：主要由任勇宁和刘一辰完成

报告撰写：主要由王明硕完成

全部工作都在 git 上记录，可见 https://github.com/renyongning/lab