A Supplemental Material

A.1 Theoretical Analysis of Graph Abstraction-based Bound Construction

Now we prove the correctness of Case 1 in Theorem 1 and both Case 2 and Case 3 in Theorem 2.

Theorem 1. *The estimated bounds for vertices within the graph sketch (Case 1) are shorter than the actual shortest distances.*

Proof. We prove our claim through contradiction. Suppose there exist vertices in the graph sketch whose shortest paths are shorter than our estimated bounds. Among these, let v be the vertex corresponding to the shortest incorrect path, i.e., $F[v] > dist[src \rightarrow v]$. Assume that the shortest path from *src* to v is denoted as $P = \{src, v_1, v_2, \ldots, v_i, \ldots, v\}$. Next, we identify the closest vertex v_i to v such that $v_i \in P$ and v_i is part of the graph sketch. If v_i doesn't exist, we default to *src* with $F[src] = 0$. Given that v is the shortest incorrect vertex in the graph sketch, v_i must satisfy $F[v_i] \leq dist[src \rightarrow v]$.

The edges in the path $P_{v_i \rightarrow v} = \{v_i, \ldots, v\}$ belong to the graph partition set $P_G = \{G_i, \ldots, G_v\}$. For any partition $G_j \in P_G$, an edge is established from G_j to G_{j+1} with a weight set as the minimal edge weight in G_j, denoted as M_j. Since $F[v]$ can be updated as $F[v] = F[v_i] + M_i + \ldots + M_v$ and M_i is not larger than the weight of edge (v_i, v_{i+1}), it follows that the sum $M_i + \ldots + M_v$ is less than the weight of the path $P_{v_i \rightarrow v}$. Therefore $F[v] \leq F[v_i] + P_{v_i \rightarrow v}$. Given $F[v_i] \leq dist[src \rightarrow v_i]$, we can conclude that $F[v] \leq dist[src \rightarrow v_i] + P_{v_i \rightarrow v} = dist[src \rightarrow v]$. It is a contradiction to our claim ($F[v] > dist[src \rightarrow v]$). Therefore, this theorem is proven. □

Theorem 2. *The estimated bounds of vertices not in graph sketches (Case 2 & Case 3) are not longer than the actual shortest distances.*

Proof. For Case 2, in our procedure, the vertex v is the out vertex of graph partitions g, then $F[v] = min\{F[g]\}$. In light of Theorem 1 and by considering each graph partition as a vertex within the GA, $F[g]$ gets the correct bounds, then $F[g] \leq dist[src \rightarrow v]$ and we also get $F[src \rightarrow v] \leq dist[src \rightarrow v]$. For the backward bound, (1) if v is also the in vertex of g', then we can get $B[v] = B[g']$. According to Theorem 1, $B[g']$ is correct, i.e., $B[g'] \leq dist[dest \rightarrow v]$. Then we can get $B[v] \leq dist[dest \rightarrow v]$ which can also be correct. (2) if v is not in vertex of any graph partition, then we set $B[v] = min\{B[g] + g.min\_edge\}$. Because v is not an in vertex, then in backword GA, there must exist an edge (u, v, w) satisfying that $dist[dest \rightarrow v] = dist[dest \rightarrow u] + w$ with graph partition g' containing edge (u, v, w). Because $B[v] = min\{B[g] + g.min\_edge\}$, we can get $B[v] \leq B[g'] + g.min\_edge \leq B[g'] + w$. Because the distance $dist[dest \rightarrow v] = dist[dest \rightarrow u] + w$ together with $B[g'] \leq dist[dest \rightarrow u]$ (according to Theorem 1), we can also get $B[v] \leq B[g'] + w \leq dist[dest \rightarrow u] + w = dist[dest \rightarrow v]$. Therefore, Case 2 is proven.

For Case 3, in our procedure, the vertex v is only the in vertex. For forward bound, we set $F[v] = min\{F[g] + g.min\_edge\}$ with graph partition g's in vertex containing v. Because v is only the in vertex, then there must exist an edge (u, v, w) satisfying that $dist[src \rightarrow v] = dist[src \rightarrow u] + w$ with graph partition g' containing edge (u, v, w). Because u satisfies Case 2 which has been proven above, then $F[u] \leq dist[src \rightarrow u]$ and we can get $F[v] = min\{F[g] + g.min\_edge\} \leq F[g'] + g'.min\_edge \leq F[u] + w \leq dist[src \rightarrow u] + w = dist[src \rightarrow v]$. For the backward bound, the in vertex v is contained in graph partition g, then $B[v] = min\{B[g]\} \leq B[g] \leq dist[dest \rightarrow u]$. Therefore, Case 3 is also proven. □

A.2 Gem Case Study: Shortest Path

Figure 23 explains how Gem computes the 6→4 in four major steps: First, we initialize $dist[src \rightarrow i]$ to $+\infty$ for all vertices except for vertex 6 (*src*) which is set to 0. Second, we derive that no graph partitions are inactive because $F[G_i] + B[G_i] < dist[src \rightarrow dest]$ (Step 4) and $\min\{dis[src \rightarrow u]\} + B[G_i] < dist[src \rightarrow dest]$ (Step 5) for each G_i. Third, we derive the graph partition loading priority. We opt to load G_3 first because $F[G_3] + B[G_3]$ offers the smallest values. Fourth, we build a new graph, including both the graph sketch and the loaded graph partition, and perform steps 1 - 3 as follows:

Graph sketch + G_3. For out vertices 4, 5, and 6 of the current in-memory graph, because $dist[src \rightarrow dest] = +\infty$, we obtain $F[v] + B[v] < dist[src \rightarrow dest]$ for all v's, Hence no vertex is pruned in step 1. Now moving to step 2, for edges from out vertex 4 and 5, because $dist[src \rightarrow 4] + B[4] \geq dist[src \rightarrow dest]$ and $dist[5] + B[5] \geq dist[src \rightarrow dest]$, we do not update out edges from 4 and 5 in this turn. But $dist[src \rightarrow 6] + B[6] < dist[src \rightarrow dest]$, so vertex 6 advances to step 3. Then we check $dist[src \rightarrow 6]$ with edge $e_{(6,5)}$. Since $dist[src \rightarrow 6] + w_{(6,5)} + B[5] < dist[src \rightarrow 4]$ (Step 3), we can update $e_{(6,5)}$, i.e., $dist[src \rightarrow 5] = min\{dist[src \rightarrow 5], dist[src \rightarrow 6] + w_{(6,5)}\} = 12$. Now vertex 5 is updated and active.

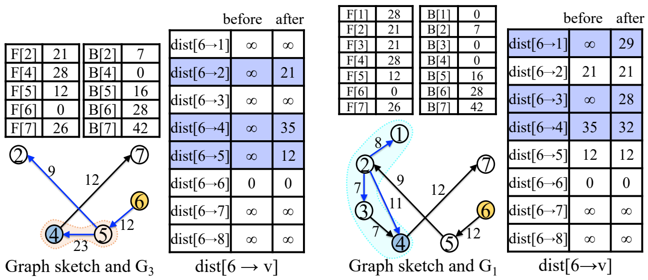
Then, we check the out edges from active vertex 5 for steps 1-3 again, e.g., $e_{(5,2)}$ and $e_{(5,4)}$. For step 1, since $dist[src \rightarrow dest]$ remains as $+\infty$, we cannot skip vertex 5. For step 2, $dist[src \rightarrow 5] + B[5] < dist[src \rightarrow dest]$. Then we move to step 3 with edge $e_{(5,2)}$ and $e_{(5,4)}$: $dist[src \rightarrow 5] + w_{(5,2)} + B[2] < dist[src \rightarrow dest]$ and $dist[src \rightarrow 5] + w_{(5,4)} + B[4] < dist[src \rightarrow dest]$, therefore $e_{(5,2)}$ and $e_{(5,4)}$ can be updated, i.e., $dist[src \rightarrow 2] = min\{dist[src \rightarrow 2], dist[src \rightarrow 5] + w_{(5,2)}\} = 21$ and $dist[src \rightarrow 4] = min\{dist[src \rightarrow 4], dist[src \rightarrow 5] + w_{(5,4)}\} = 35$. After that vertex 2 and 4 are updated and set as active.

For active vertex 2, it has no out edges in the current in-memory graph. We let it remain active. For active vertex 4, we find $dist[src \rightarrow 4] + B[4] \geq dist[src \rightarrow 4]$ which means vertex 4 will be neither updated nor active. After loading graph partition G_3, $dist[src \rightarrow 2]$, $dist[src \rightarrow 4]$, and $dist[src \rightarrow 5]$ have been updated, and they are all active vertices for loading the next graph partitions.

G_2 and G_4 become inactive. Although all partitions contain active edges, but $F[G_2] + B[G_2] \geq dist[src \rightarrow dest]$ (Step 4 for G_2) and $F[G_4] + B[G_4] \geq dist[src \rightarrow dest]$ (Step 4 for G_4), so G_2 and G_4 is marked as inactive.

We opt to load G_1 since its priority (noted as $\{dist[src \rightarrow 2]\} + B[G_1] = 21$) is the smallest, which is shown in Figure 23b. Then we check Step 1, Step 2, and Step 3 for graph sketch and G_1 as follows.

Graph sketch + G_1. For out vertices 2, 3, 4, 5, and 6 of the current in-memory graph, because $dist[src \rightarrow dest] = 35$, we obtain $F[u] + B[u] < dist[src \rightarrow dest]$ for all out vertices. Hence no vertex is pruned in step 1. Now moving to step 2, for edges from out vertices 2, 3, 4, 5, and 6, because $dist[src \rightarrow 3] + B[3] \geq$



(a) Graph sketch + G_3. **(b) Graph sketch + G_1.**

Figure 23: Case study for shortest path query 6 → 4. In the graph, blue edges are updated during the current iteration (active vertices in the next iteration). For the $dist[src \to v]$ array, light blue boxes mark vertices that were updated in this iteration.

$dist[src \to dest]$, $dist[src \to 4] + B[4] \geq dist[src \to dest]$, we do not update out edges from 3 and 4 in this turn. But $dist[src \to 2] + B[2] < dist[src \to dest]$, so vertex 2 advances to step 3. Then we work on $dist[src \to 2]$ with $e_{(2,1)}$, $e_{(2,3)}$ and $e_{(2,4)}$. Since $dist[src \to 2] + w_{(2,1)} + B[1] < dist[src \to dest]$, $dist[src \to 2] + w_{(2,3)} + B[3] < dist[src \to dest]$ and $dist[src \to 2] + w_{(2,4)} + B[4] < dist[src \to dest]$, we can update $e_{(2,1)}$, $e_{(2,3)}$ and $e_{(2,4)}$, i.e., $dist[src \to 1] = min\{dist[src \to 1], dist[src \to 2] + w_{(2,1)}\} = 29$, $dist[src \to 3] = min\{dist[src \to 3], dist[src \to 2] + w_{(2,3)}\} = 28$, and $dist[src \to 4] = min\{dist[src \to 4], dist[src \to 2] + w_{(2,4)}\} = 32$, respectively. Now vertex 1, 3, and 4 are updated and active.

Then, we check out edges from active vertices 1, 3, and 4 again, e.g., $e_{(3,4)}$ and $e_{(4,7)}$. For step 1, since $dist[src \to dest]$ is 35 now, $F[3] + B[3] < dist[src \to dest]$, and $F[4] + B[4] < dist[src \to dest]$, so we cannot skip vertex 3 and vertex 4. For step 2, since $dist[src \to 3] + B[3] < dist[src \to dest]$, and $dist[src \to 4] + B[4] \geq dist[src \to dest]$, so we cannot skip vertex 3. Then we move to step 3 with edge $e_{(3,4)}$: $dist[src \to 3] + w_{(3,4)} + B[4] \geq dist[src \to dest]$, therefore $e_{(3,4)}$ can not be updated.

For active vertex 1, it will not be updated because it has no out edges in the current in-memory graph. For active vertex 4, we find $dist[src \to 4] + B[4] \geq dist[src \to 4]$ which means vertex 4 will be neither updated nor active. After loading graph partition G_1, $dist[src \to 1]$, $dist[src \to 3]$, and $dist[src \to 4]$ have been updated, and they are all active vertices for loading the next graph partition. In that case, only G_1 contains the active out vertices and can pass Step 4 and Step 5, but G_1 is updated in the previous round, so there is no update anymore. The optimal path from the source vertex 6 to the destination vertex 4 has been discovered. We have already found the shortest path 6 → 4.

A.3 Application Programming Interfaces

Table 7 presents the application programming interfaces (APIs) defined in Gem. *Select* is used to choose the top X edges from the original graph during preprocessing, creating the graph sketch in a process referred to as edge selection. *ForwardBound* estimates forward bounds for each vertex, taking the graph abstraction (GA) and graph partition set $\mathbb{G}$ (each graph partition is viewed as a vertex) as input. *BackwardBound* estimates backward bounds for each vertex, using the transpose of graph abstraction and $\mathbb{G}$ as input. During execution, *ProcessEdge* performs the relaxation operation on edge e, i.e., updating the distance of the destination vertex of e based on the distance of the source vertex of e. *UpdatePriority* updates the priority of each graph partition in each iteration. *VertexProgram* queries the result of each vertex. With these APIs, Gem can be user-friendly for various applications.

Table 7: APIs of Gem.

Functions	Usage
Select (Size_t X)	Generate GS with the size as X.
ForwardBound (GA, $\mathbb{G}$)	Estimate forward bounds by GA and $\mathbb{G}$.
BackwardBound (GA, $\mathbb{G}$)	Estimate backward bounds by GA and $\mathbb{G}$.
ProcessEdge (Edge e)	Process edge e and perform relaxation.
UpdatePriority (Grid g)	Update the priority of graph partition g.
VertexProgram (Vertex u)	Query the result of vertex u.