# Chapter 9

# The Shortest Path Problem

## 9.1 Introduction

The single source shortest path (SSSP) problem has wide applications in transportation, networking and many other fields. The problem takes as input a weighted directed graph with $n$ vertices and $e$ edges. We are required to find $cost[x]$, the minimum cost of a path from the *source* vertex $v_0$ to all other vertices $x$ where the cost of a path is defined as the sum of edge weights along that path. Dijkstra's algorithm (or one of its variants) is the most popular single source shortest path algorithm used in practice.

Dijkstra's algorithm and the LLP algorithm assume that all edge weights are positive. Section 9.5 derives Johnson's algorithm from a lattice-linear predicate to convert the problem of a graph with negative weights (but no negative cost cycle) to a problem with non-negative weights. Section 9.6 discusses Bellman-Ford algorithm. Bellman-Ford algorithm works on any graph so long as there are no negative cost cycles.

Note that there are multiple parallel algorithms on *undirected* graphs which are not only efficient in terms of time but also in terms of the work. The situation is different for the directed graphs. Many of the problems in the directed graph require reachability. All the time-efficient parallel algorithms for reachability suffer from transitive-closure bottleneck. The transitive closure bottleneck refers to the problem that the *polylog* algorithms require a large number of processors making the parallel algorithm work inefficient (and thus impractical). In this chapter, we cover the problem of computing transitive closure of a matrix. This problem can be solved fast using the technique of *repeated squaring*. The same technique is applicable to solve all pair shortest path algorithm. However, these algorithms are far from being work-efficient. In any case, it is important to study these parallel algorithms to appreciate the transitive closure bottleneck.

In this chapter, we first describe Dijkstra's algorithm in Section 9.2. We then formulate the shortest path problem as searching for an appropriate element in a lattice and derive an LLP algorithm for the shortest path in Section 9.3. Section 9.4 discusses a parallel algorithm for the shortest path problem called the delta-stepping algorithm. Section 9.5 describes techniques that we can use when the directed graph may have edges with negative weights. In particular, it describes Johnson's algorithm that converts a weighted directed graph to another weighted graph that preserves the shortest paths and the resulting graph does not have negative weight edges. Section 9.6 describes Bellman-Ford algorithm that can be used to find the cost of the shortest path when the graph may have negative edge weights. We then consider the problem of finding costs of shortest paths for all pairs of vertices. Section 9.7 starts with an algorithm for matrix multiplication. Then, it extends the algorithm to the problem of computing the transitive closure of a binary matrix. Section 9.8 extends this problem to nonbinary matrices by viewing it as a computation of the cost of shortest path for all pairs of vertices.

## 9.2 Dijkstra's Algorithm

We consider a directed weighted graph $(V, E, w)$ where $V$ is the set of vertices, $E$ is the set of directed edges and $w$ is a map from the set of edges to positive reals (see Fig. 9.1(a) for a running example). To avoid trivialities, we assume that the graph is loop-free and every vertex $x$, except the source vertex $v_0$, has at least one incoming edge.

**Algorithm Dijkstra:** Finding the shortest costs from $v_0$ .

```
1  var dist: array[0...n-1] of integer initially ∀i : dist[i] = ∞;
2      fixed: array[0...n-1] of boolean initially ∀i : fixed[i] = false;
3      H: binary heap of (j, d) initially empty;// j is the vertex and d is the cost
4  dist[0] := 0;
5  H.insert((0,dist[0]));
6  while ¬H.empty() do
7      (j, d) := H.removeMin();
8      if (fixed[j]) continue;
9      fixed[j] := true;
10     forall k: ¬fixed(k) ∧ (j, k) ∈ E
11         if (dist[k] > dist[j] + w[j, k]) then
12             dist[k] := dist[j] + w[j, k];
13             H.add (k, dist[k]);
14 endwhile;
```

Dijkstra's algorithm maintains $dist[i]$, which is a cost to reach $v_i$ from $v_0$. Every vertex $x$ in the graph has initially $dist[x]$ equal to $\infty$. Whenever a vertex is discovered for the first time, its $dist[x]$ becomes less than $\infty$. We use the predicate $discovered(x) \equiv dist[x] < \infty$. The variable $dist$ decreases for a vertex whenever a shorter path is found due to edge relaxation.

In addition to the variable $dist$, a boolean array *fixed* is maintained. Thus, every discovered vertex is either *fixed* or *non-fixed*. The invariant maintained by the algorithm is that if a vertex $x$ is *fixed* then $dist[x]$ gives the final shortest cost from vertex $v_0$ to $x$. If $x$ is *non-fixed*, then $dist[x]$ is the cost of the shortest path to $x$ that goes only through fixed vertices.

A heap $H$ keeps all vertices that have been discovered but are non-fixed along with their distance estimates $dist$. We view the heap as consisting of tuples of the form $(j, dist[j])$ where the heap property is with respect to $dist$ values. The algorithm has one main *while* loop that removes the vertex with the minimum distance from the heap with the method $H$.removeMin(), say $v_j$, and marks it as fixed. It then *explores* the vertex $v_j$ by relaxing all its adjacent edges going to non-fixed vertices $v_k$. The value of $dist[k]$ is updated to the minimum of $dist[k]$ and $dist[j] + w[j, k]$. This step is called *edge relaxation*. If $v_k$ is not in the heap, then it is inserted, else if $dist[k]$ has decreased then the label associated with vertex $k$ is inserted in the heap. We abstract this step as the method $H.\text{add}(k, dist[k])$. Since the vertex may already be on the heap, the insertion may cause a vertex to be present in a heap with different distances. Hence, when we remove a vertex from the heap, if it is already fixed, we simply go to the next vertex in the heap. The algorithm terminates when the heap is empty. At this point there are no discovered non-fixed vertices and $dist$ reflects the cost of the shortest path to all discovered vertices. If a vertex $j$ is not discovered then $dist[j]$ is infinity reflecting that $v_j$ is unreachable from $v_0$.

Observe that every vertex goes through the following states. Every vertex $x$ is initially *undiscovered* (i.e., $dist[x] = \infty$). If $x$ is reachable from the source vertex, then it is eventually *discovered* (i.e., $dist[x] < \infty$). A discovered vertex is initially *non-fixed*, and is therefore in the heap $H$. Whenever a vertex is removed from the heap it is a *fixed* vertex. A fixed vertex may either be *unexplored* or *explored*. Initially, a fixed vertex is unexplored. It is considered explored when all its outgoing edges have been relaxed.

The following lemma simply summarizes the well-known properties of Dijkstra's algorithm. We leave them as an exercise.

**Lemma 9.1** *The outer loop in Dijkstra's algorithm satisfies the following invariants.*
*(a) For all vertices x:* $fixed[x] \Rightarrow (dist[x] = cost[x])$.
*(b) For all vertices x:* $dist[x]$ *is equal to cost of the shortest path from* $v_0$ *to x such that all vertices in the path before x are fixed.*

For complexity analysis, let $n$ be the number of vertices and $m$ be the number of edges. We analyze the version of Dijkstra's algorithm in which instead of adjusting the key in the heap for a vertex, we simply insert the vertex in the heap. As a result the heap may have a vertex multiple times with different keys. When a vertex is removed, we check if it has already been fixed. If it is fixed, then we do not need to explore it and we can remove the next vertex in the heap. Since a vertex can be inserted in the heap only when an edge is relaxed, we get that there are at most $m$ insertions from the heap. We can also conclude that there are at most $m$ deletions from the heap. Since an insertion or a deletion from a heap takes $O(\log m) = O(\log n)$ time, we get the overall time complexity of $O(m \log n)$.

Here is a walkthrough of Dijkstra's algorithm on the graph, starting from vertex $v_0$.

1. Set $v_0$ as the source vertex and assign it a distance of 0. Assign all other vertices a tentative distance of infinity. We insert the source vertex $v_0$ in the heap $H$.
2. Since the heap is not empty, we remove the vertex at the top of the heap. It is $v_0$ with a distance of 0. We mark that vertex as *fixed*. We examine its neighbors, $v_1$ and $v_2$. For $v_0 \rightarrow v_1$, the existing distance to $v_1$ is infinity. The distance 0 ( distance to $v_0$) + 9 (edge weight from $v_0$ to $v_1$) is less than infinity, so we update the distance to $v_1$ to 9 and add it to the heap. For $v_0 \rightarrow v_2$, the existing distance to $v_2$ is infinity. The distance 0 + 2 is less than infinity, so update the distance to $v_2$ to 2 and add $v_2$ to the heap.
3. We remove the vertex at the top of the heap. The smallest distance among the unvisited vertices is 2 (for $v_2$), so set $v_2$ as the current vertex. We examine its neighbors, $v_3$ and $v_4$. For $v_2 \rightarrow v_3$, the existing distance to $v_3$ is infinity. Since 2 ( distance to $v_2$) + 6 (edge weight from $v_2$ to $v_3$) is less than infinity, we update the distance to $v_3$ to 8. For $v_2 \rightarrow v_4$, the existing distance to $v_4$ is infinity. Since $2 + 5$ is less than infinity, we update the distance to $v_4$ to 7.
4. Continuing in this manner until the heap becomes empty, we get the final distances as: $v_0$: 0, $v_1$: 9, $v_2$: 2, $v_3$: 8, $v_4$: 7.

## 9.3 Constrained Single Source Shortest Path Algorithm

In this section, we assume that all edge weights are *strictly positive*. We are required to find the minimum cost of a path from a distinguished *source* vertex $v_0$ to all other vertices where the cost of a path is defined as the sum of edge weights along that path. For any vertex $v$, let $pre(v)$ be the set of vertices $u$ such that $(u, v)$ is an edge in the graph. To avoid trivialities, assume that every vertex $v$ (except possibly the source vertex $v_0$) has nonempty $pre(v)$ and that all nodes in the graph are reachable from the source vertex.

As the first step of the predicate detection algorithm, we define the lattice for the search space. We assign to each vertex $v_i$, $G[i] \in \mathbf{R}_{\geq 0}$ with the interpretation that $G[i]$ is the cost of reaching vertex $v_i$. We call $G$, the *assignment* vector. The invariant maintained by our algorithm is: for all $i$, the cost of any path from $v_0$ to $v_i$ is greater than or equal to $G[i]$. The vector $G$ only gives the lower bound on the cost of a path and there may not be any path to vertex $v_i$ with cost $G[i]$. To capture that an assignment is feasible, we define *feasibility* which requires the notion of a *parent*. We say that $v_i$ is a parent of $v_j$ in $G$ (denoted by the predicate $parent(j, i, G)$) iff there is a direct edge from $v_i$ to $v_j$ and $G[j]$ is at least $(G[i] + w[i, j])$, i.e., $(i \in pre(j)) \wedge (G[j] \geq G[i] + w[i, j])$. A node may have multiple parents.

In Fig. 9.1, let $G$ be the vector $(0, 2, 3, 5, 8)$. Then, $v_0$ is a parent of $v_2$ because $G[2]$ is greater than $G[0]$ plus $w[0, 2] (i.e., 3 \geq 0 + 2)$. Similarly, $v_1$ is a parent of $v_4$ because $G[4] \geq G[1] + 2$. A node may have multiple parents. The node $v_2$ is also a parent of $v_4$ because $G[4] \geq G[2] + 5$.

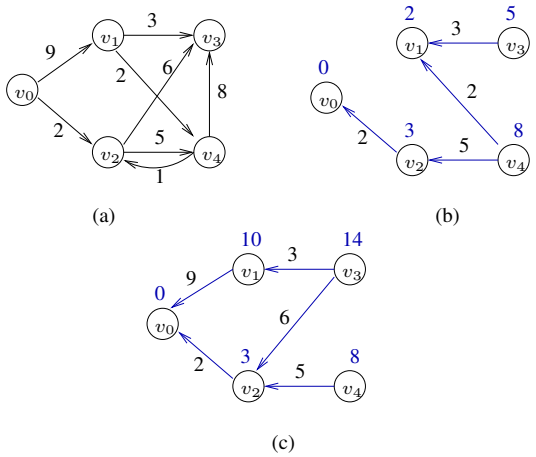



Figure 9.1: (a) A Weighted Directed Graph (b) The parent structure for $G = (0, 2, 3, 5, 8)$ (c) The parent structure for $G = (0, 10, 3, 14, 8)$. Since every non-source node has at least one parent, $G$ is feasible.

Since $w[i, j]$ are strictly positive, there cannot be a cycle in the parent relation. Now, feasibility can be defined as follows.

**Definition 9.2 (Feasible for paths)** *An assignment $G$ is* feasible for paths *iff every node except the source node has a parent. Formally, $B_{path}(G) \equiv \forall j \neq 0 : (\exists i : parent(j, i, G))$.*

Hence, an assignment $G$ is feasible iff one can go from any non-source node to the source node by following any of the parent edges. We now show that feasibility satisfies lattice-linearity.

**Lemma 9.3** *For any assignment vector $G$ that is not feasible, $\exists j : forbidden(G, j, B_{path}(G))$.*

**Proof:** Suppose $G$ is not feasible. Then, there exists $j \neq 0$ such that $v_j$ does not have a parent, i.e., $\forall i \in pre(j) : G[j] < G[i] + w[i, j]$. We show that $forbidden(G, j, B_{path}(G))$ holds. Pick any $H$ such that $H \geq G$. Since for any $i \in pre(j)$, $H[i] \geq G[i]$, $G[j] < G[i] + w[i, j]$ implies that $G[j] < H[i] + w[i, j]$. Therefore, whenever $H[j] = G[j]$, $v_j$ does not have a parent.

■

Since $B_{path}$ is a lattice-linear predicate, it follows from Lemma 2.3(c), that the set of feasible assignment vectors are closed under meets (the component-wise min operation). Hence, we can use LLP algorithm with $\alpha(G, j, B_{path}) = \min\{G[i] + w[i, j] \mid i \in pre(j)\}$.

For an unweighted graph (i.e., each edge has weight equal to 1), the above parallel algorithm requires time equal to the distance of the farthest node from the root. The LLP algorithm derived from $B_{path}$, however, may take time that depends on the weights because the advancement along a forbidden process may be small.

We now give an alternative feasible predicate that results in an algorithm that takes bigger steps. We first define a node $j$ to be *fixed* in $G$ if either it is the source node or it has a parent that is a fixed node, i.e., $fixed(j, G) \equiv (j = 0) \vee (\exists i : parent(j, i, G) \wedge fixed(i, G))$.

Observe that node $v_0$ is always fixed. Any node $v_j$ such that one can reach from $v_j$ to $v_0$ using parent relation is also fixed. We now define another feasible predicate called $B_{rooted}$, as $B_{rooted}(G) \equiv \forall j : fixed(j, G)$.

Even though it may first seem that the predicate $B_{rooted}$ is strictly stronger than $B_{path}$, the following Lemma shows otherwise.

**Lemma 9.4** *$B_{path}(G)$ iff $B_{rooted}(G)$.*

**Proof:** If $G$ satisfies $B_{rooted}$, then every node other than $v_0$ has at least one parent by definition of $fixed$, hence $B_{path}(G)$. Conversely, suppose that every node except $v_0$ has a parent. Since parent edges cannot form a cycle, by following the parent edges, we can go from any node to $v_0$.

■

It follows that the predicate $B_{rooted}$ is also lattice-linear. Then, the following threshold $\beta(G)$ is well-defined whenever the set of edges from the fixed vertices to non-fixed vertices is nonempty.

$$\beta(G) = \min_{(i,j):i \in pre(j)} \{G[i] + w[i,j] \mid fixed(i,G), \neg fixed(j,G)\}.$$

If the set of such edges is empty then no non-fixed vertex is reachable from the source. We call these set of edges *Heap* (because we would need the minimum of this set for advancement).

We now have the following result in advancement of $G$.

**Lemma 9.5** *Suppose* $\neg B_{rooted}(G)$.
*Then,* $\neg fixed(j,G) \Rightarrow forbidden(G, j, B_{rooted}, \beta(G))$.

**Proof:** Consider any assignment vector $H$ such that $H \geq G$ and $H[j] < \beta(G)$. We show that $H$ is not $B_{rooted}$. In particular, we show that $j$ is not fixed in $H$. Suppose $j$ is fixed in $H$. This implies that there is a path $W$ from $v_0$ to $v_j$ such that all nodes in that path are fixed. Let the path be the sequence of vertices $w_0, w_1, \ldots w_{m-1}$, where $w_0 = v_0$ and $w_{m-1} = v_j$. Let $w_l = v_k$ be the first node in the path that is not fixed in $G$. Such a node exists because $w_{m-1}$ is not fixed in $G$. Since $w_0$ is fixed, we know that $1 \leq l \leq m-1$. The predecessor of $w_l$ in that path, $w_{l-1}$ is well-defined because $l \geq 1$. Let $w_{l-1} = v_i$.

We show that $H[k] \geq \beta(G)$ which contradicts $H[j] < \beta(G)$ because $H[k] \leq H[j]$ as the cost can only increase going from $k$ to $j$ along the path $W$. We have $H[k] \geq H[i] + w[i,k]$ because $i$ is a parent of $k$ in $H$. Therefore, $H[k] \geq G[i] + w[i,k]$ because $H[i] \geq G[i]$. Since $i$ is fixed in $G$ and $k$ is not fixed in $G$, from the definition of $\beta(G)$, we get that $\beta(G) \leq G[i] + w[i,k]$. Hence, $H[k] \geq \beta(G)$.

■

By using the advancement Lemma 9.5, we get the algorithm *ShortestPath* shown in Fig. 9.2. In this algorithm, in every iteration we find all nodes that are forbidden (not fixed) and advance them. All nodes are advanced to $\alpha(G,j)$ that combines $\beta(G)$ with $\min\{G[i] + w[i,j] \mid i \in pre(j)\}$. Note that if a node is fixed, its parent is fixed and therefore any algorithm that advances $G[j]$ only for non-fixed nodes $j$ maintains that once a node becomes fixed it stays fixed.

By removing certain steps, we get Dijkstra's algorithm from the algorithm *ShortestPath*. It is clear that the algorithm stays correct if $\alpha(G,j)$ uses just $\beta(G)$ instead of $\max\{\beta(G), \min\{G[i] + w[i,j] \mid i \in pre(j)\}\}$. Secondly, the algorithm stays correct if we advance $G$ only on the node $j$ such that $(i,j)$ are in Heap edges and the node $j$ minimizes $G[i] + w[i,j]$. Finally, to determine such a node and $\beta(G)$, it is sufficient to maintain a min-heap of all non-fixed nodes $j$, along with the label that equals $\min_{i \in pre(j), fixed(i,G)} G[i] + w[i,j]$. On making all these changes to *ShortestPath*, we get Dijkstra's algorithm (modified to run with a heap).

It is illustrative to compare the algorithm *ShortestPath* with Dijkstra's algorithm. In Dijkstra's algorithm, the nodes become fixed in the order of the cost of the shortest path to them. In the proposed algorithm, a node may become fixed even when nodes with shorter cost have not been discovered. In Fig. 9.1, node $v_1$ becomes fixed earlier than nodes $v_3$ and $v_4$. This feature is especially useful when we are interested in finding the shortest path to a single destination and that destination becomes fixed sooner than it would have been in Dijkstra's algorithm.

Dijkstra's algorithm maintains a distance vector *dist* such that it is always feasible, i.e., for any vertex $v$ there exists a path from source to $v$ with cost less than or equal to $dist[v]$. We maintain the invariant that the cost of the shortest path from source to $v$ is guaranteed to be at least $G[v]$. Therefore, in Dijkstra's algorithm, $dist[v]$ is initialized to $\infty$ whereas we initialize $G$ to 0. Dijkstra's algorithm and indeed many algorithms for combinatorial optimization, such as simplex, start with a feasible solution and march towards the optimal solution, our algorithm starts with an extremal point in search space (even if it is infeasible) and marches towards feasibility. Also note

that in Dijkstra's algorithm, $dist[v]$ can only decrease during execution. In our algorithm, $G$ can only increase with execution.

The *ShortestPath* algorithm has a single variable $G$. All other predicates and functions are defined using this variable.

**Shortest path from node s: LLP algorithm**
**input**: $pre(j)$: list of $1..n$; $w[i,j]$: positive int for all $i \in pre(j)$
**init**: $G[j] := 0$;
**always**: $parent[j,i] = (i \in pre(j)) \wedge (G[j] \geq G[i] + w[i,j])$;
$fixed[j] = (j = s) \vee (\exists i : parent[j,i] \wedge fixed[i])$
$Heap = \{(G[i] + w[i,k]) | (i \in pre(k)) \wedge fixed(i) \wedge \neg fixed(k)\}$;

**forbidden**: $\neg fixed[j]$
**advance**: $(G[j] := \max\{\min Heap, \min\{G[i] + w[i,j] \mid i \in pre(j)\})$

Figure 9.2: **Algorithm** *ShortestPath* to find the minimum cost assignment vector less than or equal to $T$.

We get the following structural result on the assignment vectors that are feasible.

**Lemma 9.6** *Let $G$ and $H$ be two assignment vectors such that they satisfy $B_{rooted}$, then $min(G,H)$ also satisfies $B_{rooted}$.*

**Proof:** Follows directly from Lemma 2.3, because $B_{rooted}$ is a lattice-linear predicate.

■

The set of feasible assignment vectors is not closed under the join operation. In Fig. 9.1, the vectors $(0, 10, 3, 14, 8)$ and $(0, 9, 10, 12, 11)$ are feasible, but their join $(0, 10, 10, 14, 11)$ is not feasible.

We now consider the generalization of the shortest path algorithm with constraints. We assume that all constraints specified are lattice-linear. For example, consider the constraint that the cost of vertex $i$ is at most cost of vertex $j$. The predicate $B \equiv G[j] \geq G[i]$ is easily seen to be lattice-linear. If any cost vector $G$ violates $B$, then the component $j$ is forbidden (with $\alpha(G,j)$ equal to $G[i]$). The predicate $(G[i] = G[j])$ is also lattice-linear because it can be written as a conjunction of two lattice-linear predicates $(G[i] \geq G[j])$ and $(G[j] \geq G[i])$. The predicate $B \equiv (G[i] \geq k) \Rightarrow (G[j] \geq m)$ is also lattice-linear. If any cost vector violates $B$, then we have $(G[i] \geq k) \wedge (G[j] < m)$. In this case, the component $j$ is forbidden with $\alpha(G,j)$ equal to $m$. Again, from Lemma 2.6, the algorithm *LLP* can be used to solve the constrained shortest path algorithm by combining $forbidden$ and $\alpha$ for constraints with $B_{rooted}$. An application of the constrained shortest path problem is as follows. Suppose that there are $n$ dispatch trucks that start from the source vertex at time 0. Let $w[i,j]$ denote the time it takes for a truck to go from node $i$ to node $j$. An assignment vector $G$ is feasible if it is possible to design a tree rooted at the source vertex such that the path from the source vertex to vertex $i$ takes $G[i]$ units of time and $G$ satisfies specified constraints. In Fig. 9.1, the vector $(0, 9, 2, 8, 7)$ is feasible, but does not satisfy the constraint that $G[1]$ equals $G[2]$. The least vector that satisfies this additional constraint is $(0, 9, 9, 12, 11)$. LLP algorithm can be used to find this vector. When the set of additional constraints is empty, we get back the standard shortest path problem.

An example of a predicate that is not lattice-linear is $B \equiv G[i] + G[j] \geq k$. If the predicate is false for $G$, then we have $G[i] + G[j] < k$. However, neither $i$ nor $j$ may be forbidden. The component $i$ is not forbidden because if $G[i]$ is fixed but $G[j]$ is increased, the predicate $B$ can become true. Similarly, $j$ is also not forbidden.

## 9.4 Delta-Stepping Algorithm

A popular practical parallel algorithm for SSSP is $\Delta$-stepping algorithm due to Meyer and Sanders [MS03]. Meyer and Sanders also provide an excellent review of prior parallel algorithms in [MS03]. They classify SSSP algorithms

**Algorithm LLP-ShortestPath:** An Implementation of Algorithm $ShortestPath_b$ to find the minimum cost assignment vector greater than or equal to $F$ .

```
1  var G: array[0..n − 1] of real initially ∀i : G[i] = 0;
2  while ∃i : ¬fixed(i, G) do
3        E′ := { (i, k) ∈ E | fixed(i, G) ∧ ¬fixed(k, G)};
4        if (E′ = φ) then return "non-fixed nodes not reachable";
5        Let (i*, j*) ∈ E′ minimize G[i] + w[i, j];
6        G[j*] := β(G);
7        for all j ≠ j* such that forbidden(G, j) in parallel do:
8              G[j] := α(G, j);
9  endwhile;
10 return G;
11 parent(j, i, G) ≡ (i ∈ pre(j)) ∧ (G[j] ≥ G[i] + w[i, j])
12 fixed(j, G) ≡ (j = 0) ∨ (∃i : parent(j, i, G) ∧ fixed(i, G))
13 β(G) = min{G[i] + w[i, j] | (i, j) ∈ E′}
14 forbidden(G, j) ≡ ¬fixed(j, G)
15 α(G, j) = max{β(G), min{G[i] + w[i, j] | i ∈ pre(j)}}
```

as either *label-setting*, or *label-correcting*. Label-setting algorithms, such as Dijkstra's algorithm, relax edges only for fixed vertices. Label-correcting algorithms may relax edges even for non-fixed vertices. Δ-stepping algorithm is a label-correcting algorithm in which eligible non-fixed vertices are kept in an array of buckets such that each bucket represents a distance range of Δ. During each phase, the algorithm removes all vertices of the first non-empty bucket and relaxes all the edges of weight at most Δ. Edges of higher weights are relaxed only when their starting vertices are fixed. The parameter Δ provides a trade-off between the number of iterations and the work complexity. For example, when Δ is ∞, the algorithm reduces to Bellman-Ford algorithm where any vertex that has its $D$ label changed is explored. When Δ equals 1 for integral weights, the algorithm is a variant of Dijkstra's algorithm. There are many practical large-scale implementations of the Δ-stepping algorithm (for instance, by Madduri et al [MBBC07]) in which authors have shown the scalability of the algorithm. Chakravarthy et al [CCM+17] give another scalable implementation of an algorithm that is a hybrid of the Bellman-Ford algorithm and the Δ-stepping algorithm.

We now give the pseudo-code of the Delta-stepping algorithm by Meyers and Sanders [MS03]. The algorithm uses a parameter Δ. We first classify all edges $(i, j) \in E$ to be *light* or *heavy* depending upon whether its weight is less than Δ or not. Formally,

$$E_{light} = \{(i, j) \in E \mid w[i, j] \leq \Delta\}$$

$E_{heavy}$ is simply the complement of $E_{light}$.

The variable $req$ is the set of all edges that are required to be relaxed. The variable $S$ is used to maintain the set of vertices whose outgoing light edges have been relaxed but not the heavy edges. The variable $B[i]$ denotes the bucket $B_i$.

The algorithm DeltaStepping explores buckets $B_i$ sequentially in the increasing order. The variable $i$ denotes the current bucket. The outer while loop iterates until there is some bucket. The inner while loop iterates until the current bucket becomes empty. To explore a bucket, the algorithm first explores all the light edges outgoing from the bucket. When these edges are relaxed, some additional vertices may move to the current bucket. Hence, the while loop is executed until all the light edges have been explored. We use $S$ to store all the vertices whose light edges have been explored but the heavy edges remain to be explored. Once the inner while loop terminates, we explore all the heavy edges outgoing from $S$. Note that exploration of all heavy edges can never result in introduction of any vertex into the current bucket. After the exploration of all heavy edges, we go to the outer while loop to explore the next bucket.

**Algorithm DeltaStepping:** A Parallel Delta Stepping Algorithm for the Shortest Path Problem

```
1  var
2      d: array[0..n-1] of int initially ∀i : d[i] = ∞
3      req, S: set of (int, int)
4      B: sequence of buckets (sets)
5  relax(0,0);
6  int i := 0;
7  while B ≠ ∅
8      S := {}
9      while B[i] ≠ {}
10         req := {(v', d[v] + w[v, v'] | v ∈ B[i] ∧ (v, v') ∈ E_light}
11         S := S ∪ B[i]
12         B[i] := {}
13         forall (v', d') ∈ req in parallel do: relax(v', d')
14     endwhile
15     req := {(v', d[v] + w[v, v'] | v ∈ S ∧ (v, v') ∈ E_heavy}
16     forall (v', d') ∈ req in parallel do: relax(v', d')
17     i := i + 1
18 endwhile
19 function relax(int u, int c)
20     if c < d[u]
21         d[u] := c
22         move u to the bucket B[c/Δ]
23 endfunction
```

## 9.5 Graphs with Negative Weights: Johnson's Algorithm

We now consider directed graphs which have negative weight edges. For such graphs, the shortest path may not be defined if there is a cycle with negative cost. By going around the cycle, one can decrease the cost of the path arbitrarily. Therefore, we assume that even though there are edges with negative costs, there are no negative cost cycles. Our strategy for finding the shortest path in such a graph $X$ would be to convert it into another graph $Y$ on the same set of vertices such that $Y$ has all non-negative edges and it preserves all shortest paths, i.e., a path is shortest in $X$ iff it is also a shortest path in $Y$. The graph $Y$ has the same set of vertices and edges as $X$. The weight of any edge $(i, j)$ is updated as follows:

$$w'[i, j] = w[i, j] + p[j] - p[i] \tag{9.1}$$

where $p$ is a *price* vector associated with vertices. A price vector $p$ is a non-negative vector such that when we compute new costs of edges, called *reduced* costs, we get that the new cost of every edges is at least 0. The advantage of updating weights using Equation 9.1 is that it preserves shortest paths.

**Lemma 9.7** *Let $s$ and $t$ be any two vertices in the graphs. The weight of any path in the graph $Y$ equals the weight in the graph $X$ plus $(p[t] - p[s])$.*

**Proof:** Let the path be $v_0, v_1, v_2, v_k$ where $v_0 = s$ and $v_k = t$. As we compute the cost of the path in $Y$, we add the price of every intermediate vertex once and subtract it once. Only the price of $v_0$ is subtracted exactly once and the price of $v_k$ is added exactly once giving us the lemma.

■

Since the cost of all paths between $s$ and $t$ are changed by the same amount, it follows that any shortest path in $X$ is a shortest path in $Y$ and vice-versa. Now our task is reduced to finding a price vector such that $w'[i, j]$ is at least 0 for all edges. We use LLP algorithm to find such a vector. Our feasibility predicate $B$ for pricing vector is

$$\forall (i, j) \in E : w[i, j] + p[j] - p[i] \geq 0$$

Furthermore, we require $p[i] \geq 0$ for all $i$. We first show that $B$ is lattice linear.

**Lemma 9.8** *Let $X$ be any graph such that the edge $(i, j)$ has weight $w[i, j]$ and every vertex $i$ has price $p[i]$. Consider the lattice of all non-negative price vectors. Then, the predicate*

$$B \equiv \forall (i, j) \in E : w[i, j] + p[j] - p[i] \geq 0$$

*is lattice linear.*

**Proof:** Since lattice linearity is closed under conjunction, it is sufficient to show that $B_e \equiv w[i, j] + p[j] - p[i] \geq 0$ is lattice linear for arbitrary edge $e = (i, j)$. $B_e$ can be rewritten as $p[j] \geq p[i] - w[i, j]$. The right hand side of this inequality is a monotone function on $p$ and hence from the Key Lemma of lattice-linearity, we get that $B_e$ is lattice linear.

■

By applying LLP algorithm, we get the following method to find the price vector.

Since we have not provided the stopping point in Algorithm LLP-Johnson, the method may not terminate. Indeed, if there is a cycle in the graph with negative cost, the above algorithm will not terminate. We show

**Lemma 9.9** *Algorithm LLP-Johnson terminates iff there is no negative cost cycle in the graph $X$.*

**Algorithm LLP-Johnson:** Finding the minimum price vector.

1 **input**: $pre(j)$: list of $1..n$; $w[i,j]$: int for all $i \in pre(j)$
2 **init**: $G[j] := 0$;
3 **ensure**: $G[j] \geq \max\{G[i] - w[i,j] \mid i \in pre(j)\}$

**Proof:**

We first show that if there is a negative cost cycle, then the algorithm never terminates. By considering $s$ and $t$ to be the same vertex, Lemma 9.7 implies that the cost of any cycle remains unchanged after relabeling. Hence, if there is any negative cost cycle in $X$, then for all pricing vectors the cost of the cycle is negative in $Y$. For any cycle to have a negative cost, at least one edge must be negative. Hence, for all pricing vectors there exists $i, j$ such that $w'[i,j] = w[i,j] + p[j] - p[i] < 0$. This implies that there exists $j$ such that $p[j] < p[i] - w[i,j]$. Hence, $forbidden(p,j)$ holds.

Now assume that there is no negative cost cycle in the graph. Since LLP algorithm finds the least feasible element of the lattice that satisfies the feasibility predicate $B$, it is sufficient to show that there exists at least one feasible solution. Consider an additional vertex $z$ added to the graph such that $z$ has a directed edge to every node in the graph with cost 0. Since there is no incoming edge to $z$, adding $z$ cannot introduce a negative cost cycle in the graph. Since there are no negative cost cycles, there are finite number of paths with the least cost from $z$ to any vertex $x$. The cost of the shortest path from $z$ to any vertex is 0 or smaller because there is a direct edge from $z$ to any vertex with weight 0. Let $d[z,x]$ be the distance of any vertex $x$ from $z$. We define $p[x]$ to be $-d[z,x]$ and claim that $w[i,j] + p[j] - p[i] \geq 0$ for all edges $(i,j)$. If not, we get that $w[i,j] - d[z,j] + d[z,i] < 0$. However, this implies that $d[z,j] > d[z,i] + w[i,j]$ which violates that $d[z,j]$ and $d[z,i]$ correspond to the cost of the shortest paths from $z$ since by first going through $i$, the cost of reaching $j$ can be reduced.

■

The proof of Lemma 9.9 shows that there always exists a price vector that makes edge weights non-negative. From the Key Lattice-Linearity Lemma, we know that the set of all feasible price vectors are closed under meets. It can also be observed that if the price vector $p$ is feasible, and that if we add a fixed positive constant to all prices, then all prices stay positive and all the reduced weights stay the same. This observation also implies that the least price vector $p$ that satisfies $B$ must have at least one component which is zero. Otherwise, we can subtract the least price from all prices and still get a feasible price vector contradicting that $p$ is the least feasible price vector.

We now bound the number of iterations in the LLP algorithm required to find the price vector.

**Lemma 9.10** *The LLP algorithm requires at most n iterations before termination.*

**Proof:** We claim that after $i$ iterations of LLP algorithm, $p[x]$ is negative of the cost of any path from any vertex $y$ to $x$ with at most $i$ edges. Initially, when $i$ equals 0, $p[x]$ equals 0 which is equal to the negative of the shortest path (from $x$ to itself). Assume that the claim holds for $i-1$ iteration. On $i^{th}$ iteration, we show that $p[x]$ is updated appropriately. Consider any shortest path of length $i$ with $y$ as the predecessor vertex in that path. If this path is shorter than any path of length $i-1$, we have that $-p[y] + w[y,x] \leq -p[x]$. However, this condition is equivalent to $p[x] < p[y] - w[y,x]$ guaranteeing that $p[x]$ is forbidden and increased so that $p[x] = p[y] - w[y,x]$.

Since any shortest path can have at most length $n$, the lemma follows.

■

Since each LLP iteration can be computed in $O(m)$ time, we get that the graph can be "reweighted" in $O(mn)$ time. Now we can use Dijkstra's algorithm to find the shortest path in the transformed graph for a single source shortest path problem. Since the algorithm for reweighting the graph dominates Dijkstra's shortest path algorithm, we get an algorithm with $O(mn)$ complexity to find shortest paths from a vertex to all vertices in an weighted directed graph possibly with negative costs. Furthermore, all pairs shortest path problem can be solved by using

Dijkstra's algorithm from every vertex after reweighting the graph. This gives us an all pairs shortest path algorithm with the time complexity of $O(mn + n(m + n \log n) = O(mn + n^2 \log n)$. This algorithm has lower time complexity than $O(n^3)$ Floyd-Warshall algorithm whenever the graph is not dense.

## 9.6 Bellman-Ford's Algorithm

In this section we give an algorithm that computes the shortest path tree from any given vertex without first converting it into graph with non-negative edges. We assume that the graph does not have any negative cost cycle. Given an such graph and a distinguished vertex $v_0$, we know that the distance of $v_0$ to itself can never be less than 0. Hence, without loss of generality, we will assume that $v_0$ does not have any incoming edge (or, simply delete those edges). Now, we consider the distance of other vertices from $s$. To apply LLP algorithm, we view finding the optimal distance vector as finding the largest $d[x]$ for every vertex $x \neq v_0$ which satisfies the following feasibility predicate:

$$\text{for all edges } (i,j) \in E : d[j] \leq d[i] + w[i,j]$$

If we define $B_e$ for any edge $e = (i,j)$ as $d[j] \leq d[i] + w[i,j]$. The feasibility predicate $B$ can be written as

$$B \equiv \wedge_{e \in E} B_e$$

We will view the search for optimal distance vector as searching for $d$ vector in a lattice defined as follows. The lattice has the bottom element as $(M, M, \ldots, M)$ where $M$ equal $n-1$ times the largest weight edge in the graph. Since we are looking for the largest distance vector satisfying $B$, it can never be greater than $M$. In this lattice $d \leq_L d'$ iff $\forall i : d[i] \geq d'[i]$. Thus, finding the least vector in this lattice corresponds to finding the largest vector $d$ that satisfies $B$.

We now claim that

**Theorem 9.11** *$B$ is a lattice linear predicate.*

**Proof:** It is sufficient to show that $B_e$ is lattice linear because any conjunction of linear predicates is also lattice linear. $B_e$ for $e = (i,j)$ is equivalent to $d[j] \leq d[i] + w[i,j]$. This is equivalent to $d[j] \geq_L d[i] + w[i,j]$. Since the right hand side is a monotonic function of $d$, we get that $B_e$ is lattice linear.

■

**Algorithm LLP-Edge-Relaxation:** Finding the minimum distance vector satisfying $B$

```
1  input: pre(j): list of 1..n; w[i,j]: int for all i ∈ pre(j)
2  init: if (j = s) then d[j] := 0 else d[j] := maxint;
3  ensure: d[j] ≤ min{d[i] + w[i,j] | i ∈ pre(j)}
```

Algorithm LLP-Edge-Relaxation gives the correct answer; however, it may result in large computation complexity if $d$ are lowered in a non-disciplined manner. We now optimize this algorithm. Algorithm LLP-BellmanFord chooses forbidden indices more carefully. Whenever there is a forbidden vertex, all forbidden vertices are advanced. The *while* loop checks if there is any forbidden vertex. The *forall* loop advances all forbidden vertices.

Let us analyze the sequential time complexity of Algorithm LLP-BellmanFord. There are at most $n$ iteration of the *while* loop. The *forall* loop can be implemented with $O(m)$ time complexity giving us the overall sequential time complexity of $O(mn)$. For the parallel time complexity, it is sufficient to observe that the *forall* loop can be implemented in $O(\log n)$ parallel time. Thus, the parallel time complexity is $O(n \log n)$.

Earlier we had assumed that the graph does not have any negative cost cycle. What if the user input is such that it has a negative cost cycle? The algorithm LLP-BellmanFord will never terminate in that case. Problem 2 asks you to modify the algorithm so that it detects if the graph has a negative cost cycle and then outputs such a message.

**Algorithm LLP-BellmanFord:** Finding the minimum distance vector satisfying $B$

```
1 P_j: Code for thread j
2 var d: array[0..n − 1] of real initially (d[0] = 0) ∧ (∀i ≠ 0: d[i] = ∞)
3 always
4     forbidden(d, j) ≡ ∃i ∈ pre(j) : d[j] > d[i] + w[i, j]
5 while ∃j : forbidden(d, j)
6     forall j : forbidden(d, j) in parallel do
7         d[j] := min{d[i] + w[i, j] | i ∈ pre(j)};
```

## 9.7 Transitive Closure of a Matrix

We start with the problem of matrix multiplication of two $n \times n$ square matrices $A$ and $B$, i.e., we want to compute $C = AB$. The entry $C[i, j]$ can be computed as

$$C[i, j] = \sum_k A[i, k] * B[k, j].$$

We first compute an array $D_{i,j}$ such that $D_{i,j}[k] = A[i, k] * B[k, j]$. If we have $n^3$ processors, then we can assume that $n$ processors can be used for each entry $C[i, j]$. Since we have $n$ processors, this array can be computed in $O(1)$ time. Now, $C[i, j]$ is simply computed using the reduce operation on the array $D_{i,j}$ resulting in $O(\log n)$ time complexity for the algorithm on a CREW PRAM. This algorithm takes $O(n^3)$ work. Here, we have used the standard sequential algorithm for matrix multiplication. By using more advanced algorithms, one can reduce the exponent for the work. For example, by using Coppersmith and Winograd's algorithm, the work can be reduced to $O(n^{2.376})$. However, for simplicity, we will continue to use the standard matrix algorithm in this chapter.

Now consider the case when the matrix is boolean. In this case, we can compute $C[i, j]$ from the array $D_{i,j}$ using the *OR* operation. On a common CRCW PRAM, this can be achieved in $O(1)$ time although on a CREW PRAM we would still need $O(\log n)$ time.

Let $A[i, j]$ be a boolean matrix of size $n$ by $n$. This can also be viewed as a directed graph on $n$ vertices such that there is an edge from $i$ to $j$ iff $A[i, j]$ equals 1.

We now define the reflexive transitive closure of a matrix as

$$A^* = A^0 + A^1 + A^2 + \ldots A^n$$

The entry $A^*[i, j]$ equals 1 iff there is any path from $i$ to $j$. Our goal is to compute the reflexive transitive closure of $A$. We can apply the technique of *repeated squaring* to compute the transitive closure. The ideas is to compute $A^{2^k}$ by multiplying $A^{2^{k-1}}$ with itself. Now, it is a simple matter to compute the reflexive transitive closure. We first compute a matrix $C = I + A$. This matrix simply corresponds to adding self-loops in the graph at every vertex. Now, it is easy to show that $A^* = C^n = C^{2^{\log n}}$. Hence, reflexive transitive closure can be obtained in $O(\log^2 n)$ time on a CREW PRAM.

We now show that the problem of reflexive transitive closure can also be solved using the LLP technique. We view the problem as as searching for the least vector $G$ indexed by tuple $(i, j)$ where $i, j \in [0..n-1]$ such that

$$B_{closure} \equiv \forall i, j : G[i, j] \geq \max(A[i, j], \max\{G[i, k] \wedge G[k, j] \mid k \in [0..n-1]\}).$$

Since the right hand side of the inequality is a monotone function of $G$, it follows that $B_{closure}$ is lattice-linear. We can apply $LLP$ algorithm to compute the transitive closure. We have a boolean lattice of all vectors of size $O(n^2)$. Our goal is to find the least vector in this lattice that satisfies $B_{closure}$. The corresponding $LLP$ algorithm is shown in Fig. LLP-Transitive-Closure

Let us translate this high level algorithm to a parallel computer with $n^3$ processors. A processor is indexed by $(i, j, k)$ in the algorithm shown in Fig. 9.3. The processor checks if $G[i, j]$ is less than $G[i, k] \wedge G[k, j]$ and sets $G[i, j]$

**Algorithm LLP-Transitive-Closure:** A High-Level Algorithm to find the transitive closure of a matrix

```
1 input: A: matrix of boolean; // A[i,j] is 1 if there is an edge from i to j;
2 var G: matrix of boolean initially ∀i,j : G[i,j] = A[i,j];
3 forbidden(i,j): (∃k : G[i,j] < G[i,k] ∧ G[k,j])
4       advance: G[i,j] := 1;
```

to 1 whenever the condition is met. Since for the same value of $i$ and $j$, there may be multiple values of $k$ for which the condition is true, multiple processors may set $G[i,j]$ to 1 concurrently. We will assume Common CRCW PRAM model for this computation. In such a model, it is easy to see that after $\log n$ steps, $G$ matrix will correspond to the transitive closure. This is because the diameter of the graph will reduce by a factor of two after every step. On a CREW PRAM, we can compute the reflexive transitive closure in $O(\log^2 n)$ time.

```
input: A: matrix of boolean; // A[i,j] is 1 if there is an edge from i to j;
output: transitive closure of A;

var G: matrix of boolean initially ∀i,j : G[i,j] = A[i,j];
    for t := 1 to ⌈log n⌉ do
        forall i,j,k in parallel do
            if (G[i,j] < G[i,k] ∧ G[k,j]) then
                G[i,j] := 1;
        endforall;
    endfor;
return G; // the transitive closure
```

Figure 9.3: A Synchronous Parallel Algorithm with parallel time complexity $O(\log n)$ to find the transitive closure of a matrix A

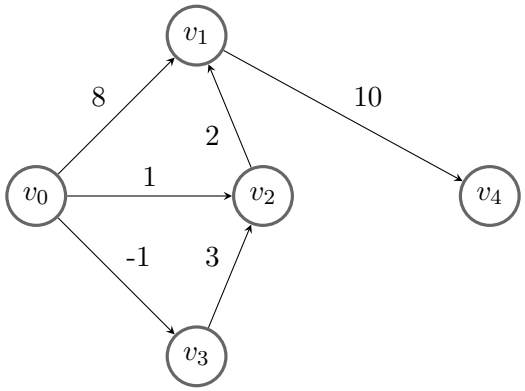
## 9.8 All Pairs Shortest Path

We now extend the algorithm to nonbinary matrices. Suppose that $A[i,j]$ represents the weight of the direct edge from $i$ to $j$ denoting the cost of going from $i$ to $j$ using at most one edge. We assume that all diagonal entries are zero and that there is no negative weight cycle. Our goal is to find a matrix $G[i,j]$ such that $G[i,j]$ is the least cost of going from $i$ to $j$ by using any number of edges.

Let us formulate this problem as finding the least vector of size $n^2$ in a distributive lattice. We initialize $G$ as $A$. This is the bottom element of our distributive lattice. Our goal is to find the least vector that satisfies the following constraint:

$$B_{Floyd-Warshall} \equiv \forall i,j : i \neq j : G[i,j] \leq \min_k \{G[i,k] + G[k,j]\}.$$

When $i$ equals $j$, $G[i,j]$ equals 0 and will stay fixed through the execution of the algorithm. The cost of going from $i$ to $j$ must be smaller than the cost of going from $i$ to $k$ and $k$ to $j$. Fig. 9.4 gives a high-level LLP algorithm for all pairs shortest path. Fig. 9.5 gives an implementation that takes $O(\log^2 n)$ time and $O(n^3 \log n)$ work with $O(n^3)$ processors.




**Algorithm LLP-Floyd-Warshall:** A High-Level Algorithm to find the shortest cost matrix for a directed graph

```
1 input: A: matrix of real; // A[i,j] is the weight of the edge from i to j;
2 var G: matrix of real initially ∀i,j : G[i,j] = A[i,j];
3 forbidden(i,j): (∃k : G[i,j] > G[i,k] + G[k,j])
4       advance: G[i,j] := min_k{G[i,k] + G[k,j]};
```

## 9.9 Summary

The following table lists all the algorithms discussed in this chapter.

```
input: A: matrix of real; // A[i,j] is the weight of the edge from i to j;
output: matrix G such that G[i,j] is the cost of the shortest path from i to j;

var G: matrix of real initially ∀i,j : G[i,j] = A[i,j];
    for t := 1 to ⌈log n⌉ do
        forall i,j in parallel do
            if (G[i,j] > min_k(G[i,k] + G[k,j])) then
                G[i,j] := min_k(G[i,k] + G[k,j]);
        endforall;
    endfor;
return G;
```

Figure 9.4: A Synchronous Parallel Algorithm with $O(\log^2 n)$ time and $O(n^3 \log n)$ work to find the Shortest Cost Matrix of a Graph

| Problem | Algorithm | Parallel Time | Work |
|---|---|---|---|
| Shortest Path | Dijkstra | $O(m)$ | $O(m \log n)$ |
| Shortest Path | LLP-Dijkstra | $O(m)$ | $O(m \log n)$ |
| Shortest Path | Delta-Stepping | $O(m)$ | $O(m \log n)$ |
| Graph Conversion | Johnson | $O(n)$ | $O(mn)$ |
| Shortest Path | LLP-Bellman-Ford | $O(n)$ | $O(mn)$ |
| All Pairs Shortest Path | Floyd-Warshall | $O(\log^2 n)$ | $O(n^3 \log n)$ |
| Transitive closure | LLP-Transitive-Closure | $O(\log^2 n)$ | $O(n^3 \log n)$ |

## 9.10 Problems

1. Prove Lemma 9.1.
2. Modify LLP-BellmanFord so that it detects if the input graph has a negative cost cycle and then outputs such a message.
3. Give an NC algorithm that that finds all strongly connected components in a directed graph.
4. Give an NC algorithm that checks whether the input directed graph is acyclic.
5. Give an NC algorithm to topologically sort an acyclic graph.
6. Give an NC algorithm to find all nodes reachable from a given vertex.

## 9.11 Bibliographic Remarks

The single source shortest path problem has a rich history. For the history of Dijkstra's algorithm, the reader is referred to the book by [Eri19]. One popular research direction is to improve the worst case complexity of Dijkstra's algorithm by using different data structures. For example, by using Fibonacci heaps for the min-priority queue, Fredman and Tarjan [FT87] gave an algorithm that takes $O(e + n \log n)$. There are many algorithms that run faster when weights are small integers bounded by some constant $\gamma$. For example, Ahuja et al [AMOT90] gave an algorithm that uses Van Emde Boas tree as the priority queue to give an algorithm that takes $O(e \log \log \gamma)$ time. Thorup [Tho00] gave an implementation that takes $O(n + e \log \log n)$ under special constraints on the weights. Raman [Ram97] gave an algorithm with $O(e + n\sqrt{\log n \log \log n})$ time. The LLP algorithm for the shortest path is taken from [AKG20]. Bellman and Ford's algorithm is from [Bel58] and [For56].