# Single-Source Shortest Path

**Single-Source Shortest Path**

A Single-Source Shortest Path (SSSP) is basically the path of least weight from a source vertex to every other vertex.

**Even More Definitions**

(Simple) Path: $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$

- Where $(v_i, v_{i+1}) \in E, \forall_{0 \le i \le (k-1)}$
- Simple = A path with no repeated vertex

Shortcut Notation: $v_0$ - squiggly line p -> $v_k$

- Means that p is a path from $v_0$ to $v_k$

Path Weight: $PW(p) = \sum_{1=0}^{k-1} w(v_i, v_{i+1})$

Shortest Path Weight from vertex a to b: $\delta(a, b)$

- Pronounced as delta a b

$$\delta(a,b) = \begin{cases} \min(PW(p)), & path\ between\ a, b \\ \infty, & b\ is\ unreachable \end{cases}$$

**Single-Source Shortest Path Problem:**

- Given G(V, E), w(a, b): E->R, and a source vertex s
- Find δ(s, b) (+best paths) from vertex s to each vertex b∈V

**Supporting Data Structures: Arrays**

We need the following supporting DSes to solve the SSSP problem:

- Array D of size V for distances
  - Initially, D[v] = 0 if v=s, else D[v] = ∞
  - D[v] decreases as we find better paths
  - D[v]≥δ(s,v) throughout the execution of the SSSP algorithm
  - D[v]=δ(s,v) at the end of the SSSP algorithm
- Array p of size V for predecessor
  - p[v] = the predecessor on best path from source s to v
  - p[s] = -1
  - Similar to BFS/DFS

**Edge Case: Negative Edge Weight and Cycles**

Sometimes, there are graphs with negative edge weights. Even worse, some graphs have negative total cycle weight, resulting in infinitely decreasing weight as we keep travelling around the cycle. This is a case that's difficult to catch without a proper algorithm. Let us analyse the different algorithms.

**Methods to Solve the SSSP Problem**

There are 7 ways to solve the SSSP Problem, depending on the graph:

- BFS for Unweighted Graphs
- General SSSP Algorithm (not going to be used ever)
- O(VE) Bellman Ford's SSSP Algorithm
- BFS/DFS for Trees
- One-pass Bellman Ford for DAG

- Original Dijkstra's Algorithm
- Modified Dijkstra's Algorithm

**Initialisation Step**
This is a step common to all SSSP algorithms. The pseudo code is as follows:

```
initSSSP(s)
      for each v ∈ V // initialization phase
            D[v] ← 1000000000 // use 1B to represent INF
            p[v] ← -1 // use -1 to represent NULL
      D[s]←0 // this is what we know so far
```

The concept is as follows:

- For all vertices, we initialise the distance from the source to that vertex to be some huge number
- We initialise all predecessors to be unknown
- Lastly, we initialise the distance from the source to the source to be 0.

**Relaxation Operation**
Reduce the **shortest distance estimate** to a vertex using an edge from another vertex i.e. using an alternate path through another vertex.

This is also the concept for the relaxation operation, which will be used for **most** of the SSSP algorithms:

```
relax(u, v, w(u,v))
      if D[v] > D[u]+w(u,v) // if SP can be shortened
            D[v] ← D[u]+w(u,v) // relax this edge
            p[v] ← u // remember/update the predecessor
            // if necessary, update some data structure
```

The concept is as follows:

- For vertices first and second, we check if the current cost of travelling to second is greater than the cost of travelling to first + cost of travelling from first to second using the current edge.
- If it's greater, that means that the current edge + the path taken to reach first from source so far is the new shortest path
- However, it is not the shortest path yet. This is called a shortest path estimate.
- Subsequently, as even shorter paths get discovered from source to first, naturally the path from source to second (which is source to first then first to second) will relax.
- Perhaps even other paths that do not go through first are found, and the path to second relaxes further.

1. <u>O(V+E) BFS for Unweighted/Equal Weight Graphs</u>

As we had previously covered, BFS can be used to find the shortest path for graphs with unweighted edges or edges with all the same weight. This is because with no differences in edge weights, The SSSP can be viewed as a problem of finding the least number of edges traversed from source s to other vertices.

The O(V+E) Breadth First Search (BFS) traversal algorithm precisely measures this (BFS Spanning Tree = Shortest Paths Spanning Tree)

- Run BFS from source vertex s.

- If we want the shortest path to v, we can reconstruct the path using the resultant predecessor array starting from index v.
- (Path length)*(edge weight) is cost of the shortest path.
- Another way is to modify the BFS a little
    - Change the visited array to a distance array
    - Do a similar initialisation of the distance array as SSSP
    - Change the updating of the visited array to the updating of the shortest distance

Here is the new pseudo code:

```
// initialisation
for all v in V
      D[v] ← INF
      p[v] ← -1
Q ← {s} // start from s
D[s]←0

// execution
while Q is not empty
      u ← Q.dequeue()
      for all v adjacent to u // order of neighbor
            if D[v] = INF // influences BFS
                  D[v] ← D[u]+1 // visitation sequence
                  p[v]←u
                  Q.enqueue(v)

// we can then use information stored in D/p
```

However, for general cases i.e. weighted graphs, BFS will not work as sometimes the shortest path is one that takes more edges i.e. a detour. BFS cannot detect this and will only report the path with the minimum number of edges. Else, BFS is the more efficient O(V+E) algorithm to solve SSSP for unweighted graphs.

2. General SSSP Algorithm on Graphs with no Negative Weight Cycles

The way to determine if the SSSP problem has been solved by an algorithm is to check if any edge can relax further. When no edges can be relaxed further, i.e. for all edges (u,v), D[v] <= D[u] + w(u,v), the SSSP problem has been solved.

As such, the way to check would be:

```
initSSSP(s) // as defined previously

repeat // main loop
      select edge(u, v) ∈ E in arbitrary manner
      relax(u, v, w(u, v)) // as defined previously
until all edges have D[v] <= D[u] + w(u, v)
```

If we are given a graph without any negative weight cycles, the chance of this SSSP algorithm terminating is actually very low, and will take a lot of time. This is mainly due to the random selection of edges to try to relax.

This clearly isn't a very sound algorithm. However, the concept behind this algorithm does make sense. This is where Bellman Ford's Algorithm comes int, as it allows us to do these relaxations in a better order.

3. Bellman Ford's Algorithm

Bellman Ford's Algorithm basically involves the relaxation of all E edges V-1 times, without any need for any particular order of relaxation. The concept is that after V-1 relaxations of every edge, if there are no negative weight cycles, the SSSP problem is certain to be solved.

**Bellman Ford's Algorithm Implementation: Adjacency List or Edge List**

This is the pseudo code:

```
initSSSP(s)

// Simple Bellman Ford's algorithm runs in O(VE)
for i = 1 to |V|-1 // O(V) here, i is just for looping!
      for each edge(u, v) ∈ E // O(E) here
            relax(u, v, w(u,v)) // O(1) here

// At the end of Bellman Ford's algorithm,
// D[v] = δ(s, v) if no negative weight cycle exist
```

The overall time complexity is thus O(VE).

Claim: The SSSP problem is solved if there are no negative weight cycles
Proof:

Theorem 1: If G = (V, E) contains no negative weight cycle, then the shortest path p from s to v is a simple path

Proof by Contradiction:

1. Suppose the shortest path p is not a simple path
2. Then p contains one (or more) cycle(s)
3. Suppose there is a cycle c in p with positive weight
4. If we remove c from p, then we have a shorter 'shortest path' than p
5. This contradicts the fact that p is a shortest path
6. Even if c is a cycle with zero total weight (it is possible!), we can still remove c from p without increasing the shortest path weight of p
7. So, p is a simple path (from point 5) or can always be made into a simple path (from point 6)

In other words, path p has at most |V|-1 edges from the source s to the "furthest possible" vertex v in G (in terms of number of edges in the shortest path).

Theorem 2: If G = (V, E) contains no negative weight cycle, then after Bellman Ford's terminates D[v] = δ(s, v), ∀v ∈ V

Proof by Induction:

1. Define $v_i$ to be any vertex that has shortest path p requiring i hops (number of edges) from s
2. Initially $D[v_0] = \delta(s, v_0) = 0$, as $v_0$ is just s
3. After 1 pass through E, we have $D[v_1] = \delta(s, v_1)$
4. After 2 passes through E, we have $D[v_2] = \delta(s, v_2)$, ...
5. After k passes through E, we have $D[v_k] = \delta(s, v_k)$
6. When there is no negative weight cycle, the shortest path p will be simple (see the previous proof)
7. Thus, after |V|-1 iterations, the "furthest" vertex $v_{|V|-1}$ from s has $D[v_{|V|-1}] = \delta(s, v_{|V|-1})$
    - Even if edges in E are processed in the worst possible order

**Using Bellman Ford's Algorithm to Check for Negative Weight Cycles**
Since we expect all shortest paths to be found after V-1 passes, if a subsequent pass still results in a relaxation of a path, then there exists a negative-weight cycle reachable from s.

- Corollary: If a value D[v] fails to converge after |V|-1 passes, then there exists a negative-weight cycle reachable from s

We can thus implement an additional check after running Bellman Ford's:

```
for each edge(u, v) ∈ E
      if (D[u] != INF && D[v] > D[u]+w(u, v)) // check for unreachable
            report negative weight cycle exists in G
```

**Performance Analysis**
Bellman Ford's Algorithm can work on:

- Small graph without negative weight cycle – O(VE)
- Small graph with negative weight cycle – O(VE) termination + detection
- Small graph with some negative weight edges and no negative cycle – O(VE)

Similar performance can be achieved with Adjacency List and Edge List, given a sparse graph. For a near complete graph, Adjacency Matrix may be more efficient due to cache locality.

4. <u>BFS/DFS for Trees</u>

In a tree, every path between any two vertices is the shortest path, as there is only one unique path between any two vertices. We will also avoid any potential negative weight cycles as firstly, most trees tend to be undirected, and secondly, even if they are bidirected with negative weight cycles, those cycles are likely to be **trivial** aka involves the looping between 2 adjacent vertices.

- Cycles are technically defined as paths that do not reuse edges to end up at a vertex already visited. Hence undirected edges are not exactly cycles.

This is why both BFS and DFS can be used. Furthermore, both BFS and DFS help to ensure that no repeated visiting occurs, hence avoiding the trivial negative weight cycles.

The time complexity is O(V) instead of O(V+E), since E = V-1.

5. One-pass Bellman Ford's for Directed Acyclic Graph

Cycles are the reason for the need to repeatedly relax the edges of a graph. This is because as the same edge can be used for multiple paths, and depending on the order of edge relaxation, we may not find the shortest path first.

As such, if we happen to have a directed acyclic graph (DAG), we can actually simplify Bellman Ford's algorithm and make it more efficient.

First, we will need to topologically sort the vertices using Kahn's Algorithm or DFS, as covered previously. Then, we will simply need to run Bellman Ford's algorithm once across all edges once in topological order.

We can also use the same method even if the source vertex is not the first vertex of the topological sort. We just need to ensure that the source vertex's distance is correctly recorded as 0, with every other path estimate being recorded as infinity, i.e. both those before and those after the source vertex.

This algorithm works as with directed edges, it is not possible for a vertex to reach the vertices before it in topological order. Only vertices after that source vertex would be reachable, and that is where the One-pass Bellman Ford's algorithm can cover.

The overall time complexity is O(V+E), as compared to the O(VE) of the original Bellman Ford's algorithm. This is because:

- Topological Sort – O(V+E)
- Iterating Through – O(V+ (V*k)) = O(V+E)
- Overall – O(V+E)

6. Original Dijkstra's Algorithm for Graph with no Negative Weight Edge

Bellman Ford's algorithm works for graphs with no negative weight edge, but it is unfortunately very slow, with O(VE) complexity.

- A reasonably sized weighted graph has around 1000 vertices and 100000 edges. O(VE) would be too slow.

Furthermore, most graphs have no negative weight edges, e.g. time taken to travel from one city to another. Dijkstra's Algorithm thus exploits this property to achieve greater efficiency.

**Assumption and Key Ideas of Dijkstra's Algorithm**

Formal assumption: For each edge(u, v) $\in$ E, we assume w(u, v) $\geq$ 0 (non-negative).

Method:

- We first look at all current path estimates and the "receiving" vertex, and pick the one with the minimum value. At the start, it will only be the source, which has estimate 0.
- We then relax the edges going out of the "receiving" vertex, updating the existing path estimates
- Once the "receiving" vertex has been chosen, it will be added to a set of Solved vertices, which is a set of vertices whose final shortest path weights have been determined. After the first iteration, Solved = {s}.
- We look at the rest of the updated path estimates and repeat the process.

This is yet another greedy algorithm

**Original Dijkstra's Algorithm Implementation: PriorityQueue and HashTable**

As we need to be able to constantly extract the shortest path estimate, we would thus need to use a PriorityQueue. This is in addition to the distance array, D, and the predecessor array, p.

- Store the shortest path estimate for vertex v as an IntegerPair (d, v) in the PriorityQueue, where d=D[v] (current shortest path estimate)
- Initialisation Process: Enqueue (inf, v) for all vertices except for source s. We will then enqueue (0, s) into the PriorityQueue.
- Main Loop until the PriorityQueue is empty:
    - Remove vertex u with minimum d from the PriorityQueue
    - Add u to the set of Solved vertices
    - Relax all of its outgoing edges
- If an edge(u, v) is relaxed, we need to find the vertex v that it is pointing to in the PriorityQueue and **update** the shortest path estimate.
    - We thus need to find v quickly. This is done with the help of a HashTable or HashMap.
    - Alternatively we can use a bBST

**Update Operation**

Use a HashTable or HashMap to store the item and its index inside the PriorityQueue to enable O(1) lookup. Subsequently, call both ShiftUp and

ShiftDown, where only one will happen. The overall update operation is thus O(log N).

**bBST to Replace PriorityQueue**
Allows for O(log N) extraction, O(log N) insertion and O(log N) updating, which involves removal, updating and insertion.

Claim: Dijkstra's Algorithm works, despite us only processing each vertex once

Loop invariant = Every vertex v in set Solved has the correct shortest path distance from solved, i.e. D[v] = δ(s, v). This is true initially since Solved = {s} and D[s] = δ(s, s) = 0.

Dijkstra's Algorithm then iteratively adds the next vertex u with the lowest D[u] into set Solved. Is this loop invariant always valid?

Proof:

Lemma 1: Subpaths of a shortest path are shortest paths

Let p be the shortest path: $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$
Let $p_{ij}$ be the subpath of p: $p_{ij} = \langle v_i, v_{i+1}, v_{i+2}, \dots, v_j \rangle, 0 \leq i \leq j \leq k$
Then $p_{ij}$ is a shortest path (from i to j)

Proving by contradiction:
If $p_{ij}$ is not the shortest path, the we should have another $p_{ij'}$ that is shorter than $p_{ij}$. We can then cut out $p_{ij}$ and replace it with $p_{ij'}$, which results in a shorter path from $v_0$ to $v_k$. However, p is the shortest path from $v_0$ to $v_k$, which results in a contradiction. Thus $p_{ij}$ must be a shortest path between $v_i$ and $v_j$.

Lemma 2: After a vertex v is added to Solved, shortest path from s to v has been found

Proving by contradiction:
Let p be path from s to v when v was added to Solved, and u is the predecessor of v along this path.

Suppose at some later point we find the "real" shortest path p', with u' being the predecessor of v along this path p'.

Therefore SP(s,u)+w(u,v) > SP(s,u')+w(u',v).

Since weights are non-negative, as vertices are added to Solved, the shortest path cost will not decrease (as the later vertices have their shortest paths built from the shortest paths of already solved vertices).

u' cannot be added to Solved earlier than v, otherwise the shortest path cost from s to u' would be in the PQ and when we add v to solve, the correct SP for v will have been computed.

Now if u' is added later than v, SP(s,v) <= SP(s,u'), as seen from the fact that the shortest path cost will not decrease.

Therefore in order for SP(s,u)+w(u,v) > SP(s,u')+w(u',v), either

- There are negative weight edge(s) in the shortest path from s to u' such that SP(s,u') < SP(u,v)
- w(u',v) is negative

Both of which will contradict the condition that there are only positive weight edges.

Therefore by Lemma 2, since the shortest path to v has been found once it is put into Solved, we will never need to revisit it again. Thus, this greedy algorithm works.

**Analysis of Dijkstra's Algorithm**

There are two parts to the algorithm:

- PriorityQueuing
    - Each vertex will only be queued and dequeued once
    - As such, it will happen O(V) times
    - Each queue and dequeue will take O(log V) time (since at most there are V items in the queue) if implemented using Binary Min Heap (ExtractMin) or balanced BST, findMin()
    - Altogether it will take O(V log V) time
- Relaxation
    - All O(E) edges are processed only once
    - The update of the shortest path estimate will take O(log V) time
        - Using Binary Min Heap and Hash Table
        - Using bBST
    - Altogether it will take O(E log V) time
- Overall: O((V+E) log V)

The above algorithm will unfortunately not work when there is a negative weight edge present, since the negative edges cause the vertices "greedily" chosen first to eventually not have the shortest path from the source.

7. Modified Dijkstra's Algorithm for Graphs with no Negative Weight Cycles

This algorithm is useful if there are negative weight edges, but no negative weight cycles.

Similarly, there is a formal assumption: the graph has no negative weight cycles.

The key concept is to use the PriorityQueue as a queue of events to be processed, where the event is the relaxation of an edge. The processing of this event would thus spawn subsequent events (as it may relax the outgoing edges as well), which will all be queued into the PriorityQueue. We then keep track of the latest relaxation for each edge using the distance array, where any other relaxations that are not the latest will be ignored when dequeued.

This uses the Lazy Data Structure concept:

- Extract pair (d, u) in front of the priority queue PQ with the minimum shortest path estimate so far
- If d = D[u], we relax all edges out of u
- Else if d > D[u], we discard this inferior (d, u) pair.
- If during edge relaxation, D[v] of a neighbour v of u decreases, enqueue a new (D[v], v) pair for future propagation of shortest path estimate
    - There is no longer any need to do updating
    - Thus we can use Java API for the PriorityQueue, since we no longer need to implement the update function

The pseudo code is as such:

```
initSSSP(s)

PQ.enqueue((0, s)) // store pair of (dist[u], u)
while PQ is not empty // order: increasing dist[u]
      (d, u) ← PQ.dequeue()
      if d == D[u] // important check, lazy DS
            for each vertex v adjacent to u
                  if D[v] > D[u] + w(u, v) // can relax
                        D[v] = D[u] + w(u, v) // relax
                        PQ.enqueue((D[v], v)) // (re)enqueue this
```

**Modified Dijkstra's Algorithm Analysis for Graphs with no Negative Weight Edge**

We prevent a processed vertex v to be re-processed again if its d > d[v], i.e. it's an inferior copy. If there is no negative weight edge, there will never be another path that can decrease D[u] once u is greedily processed. This is proven in the above Dijkstra's Algorithm proof.

- Each vertex is processed once – O(V)
- Each extraction/dequeue still runs in O(log V) time
- Overall: O(V log V)

Each time a vertex is processed, we try to relax all of its edges. Hence all O(E) vertices are processed.

- If relaxing (u, v) decreases D[v], we re-enqueue the same vertex with a better shortest path estimate. Duplicates may occur, and up to O(V) copies may be made if each edge to v causes a relaxation.
- However, the previous check prevents reprocessing of any inferior copies.
- In terms of the PriorityQueue, there are at most O(E) pairs.
- Each insert and extractMin hence still runs in O(log V), for a total of O(E log V).

Overall, if there are no negative weight edges, Modified Dijkstra's Algorithm runs in O((V+E) log V).

**Modified Dijkstra's Algorithm Killers**

There may be extreme cases where the algorithm re-processes the same vertices repeatedly. Such extreme cases that causes exponential time complexity are rare and thus in practice, the modified Dijkstra's implementation runs much faster than the Bellman Ford's algorithm.

- If you know your graph has only a few (or no) negative weight edges, this version is probably one of the best current implementations of Dijkstra's algorithm.
- But if you know that your graph has a high probability of having a negative weight cycle, use the tighter (and also simpler) O(VE) Bellman Ford's Algorithm as this Modified Dijkstra's Algorithm implementation can be trapped in an infinite loop.