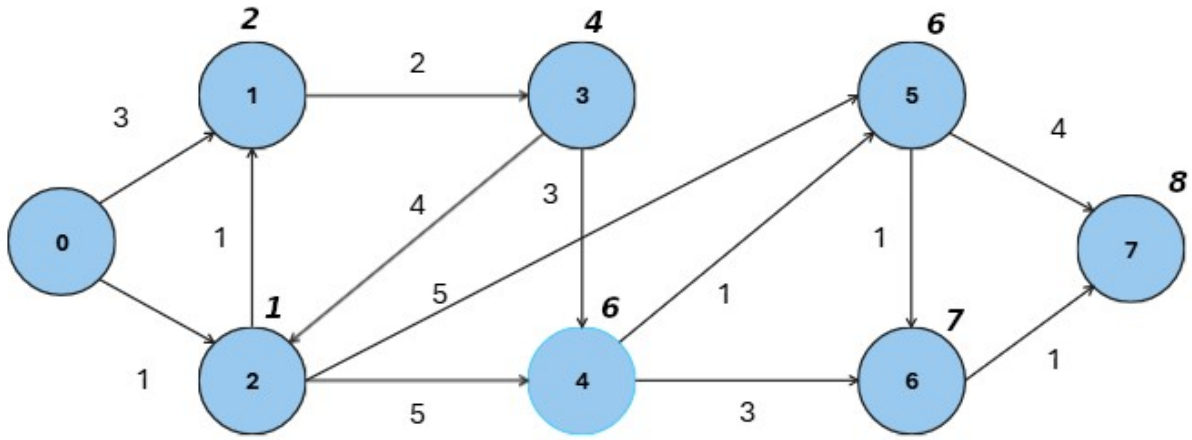
Using a script that uses Dijkstra's algorithm (with heap optimizations) and internally prints each step, here is the steps the algorithm takes. The final solutions produced were verified to be correct manually, computing the distances mentally. Here is the output of the program, and modified image with distances applied:

--- Dijkstra's Algorithm from Node 0 ---
Initial distances: {0: 0, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: inf}
Step 0: Visiting node 0 with current distance 0
    Updating distance to node 1: inf → 3
    Updating distance to node 2: inf → 1
Distances after step 0: {0: 0, 1: 3, 2: 1, 3: inf, 4: inf, 5: inf, 6: inf, 7: inf}
Step 1: Visiting node 2 with current distance 1
    Updating distance to node 1: 3 → 2
    Updating distance to node 4: inf → 6
    Updating distance to node 5: inf → 6
Distances after step 1: {0: 0, 1: 2, 2: 1, 3: inf, 4: 6, 5: 6, 6: inf, 7: inf}
Step 2: Visiting node 1 with current distance 2
    Updating distance to node 3: inf → 4
Distances after step 2: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: inf, 7: inf}
Step 3: Visiting node 1 with current distance 3
    Skipping since a shorter path is already found.
Step 3: Visiting node 3 with current distance 4
    No update required for node 2 (current: 1, new: 8)
    No update required for node 4 (current: 6, new: 7)
Distances after step 3: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: inf, 7: inf}
Step 4: Visiting node 4 with current distance 6
    No update required for node 5 (current: 6, new: 7)
    Updating distance to node 6: inf → 9
Distances after step 4: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: 9, 7: inf}
Step 5: Visiting node 5 with current distance 6
    Updating distance to node 6: 9 → 7
    Updating distance to node 7: inf → 10
Distances after step 5: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: 7, 7: 10}
Step 6: Visiting node 6 with current distance 7
    Updating distance to node 7: 10 → 8
Distances after step 6: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: 7, 7: 8}
Step 7: Visiting node 7 with current distance 8
Distances after step 7: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: 7, 7: 8}
Step 8: Visiting node 6 with current distance 9
    Skipping since a shorter path is already found.
Step 8: Visiting node 7 with current distance 10
    Skipping since a shorter path is already found.

Final shortest distances from node 0: {0: 0, 1: 2, 2: 1, 3: 4, 4: 6, 5: 6, 6: 7, 7: 8}



Program Code (python):

```
import heapq

def dijkstra_verbose(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]
    step = 0

    print(f"\n--- Dijkstra's Algorithm from Node {start} ---\n")
    print(f"Initial distances: {distances}")

    while pq:
        current_distance, current_node = heapq.heappop(pq)
        print(f"\nStep {step}: Visiting node {current_node} with
current distance {current_distance}")

        if current_distance > distances[current_node]:
            print("\tSkipping since a shorter path is already found.")
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                old_distance = distances[neighbor]
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
                print(f"\tUpdating distance to node {neighbor}:
{old_distance} → {distance}")
            else:
                print(f"\tNo update required for node {neighbor}
(current: {distances[neighbor]}, new: {distance})")
```

```
        print(f"Distances after step {step}: {distances}")
        step += 1

    print(f"\nFinal shortest distances from node {start}:
{distances}")
    return distances

# Compute SSSPT for each node and print the steps
graph = {
    0: {1: 3, 2: 1},
    1: {3: 2},
    2: {1: 1, 4: 5, 5: 5},
    3: {2: 4, 4: 3},
    4: {5: 1, 6: 3},
    5: {6: 1, 7: 4},
    6: {7: 1},
    7: {}
}

#Compute SSSPT for each node and print the steps
#for node in graph:
#    dijkstra_verbose(graph, node)

# Compute SSSPT for a specific node and print the steps
dijkstra_verbose(graph, 0)
```

@everywhere in Julia is an important macro in the Distributed package, as it tells each worker thread to define/import/allocate on the individual worker thread as well as the main thread. This is due to each worker thread having its own memory space, with nothing shared between them, so it is important to specify what pieces of code should be shared. Given the initial setup of importing the Distributed package and defining the number of worker threads, here are 3 pieces of code that need the @everywhere macro to work correctly on all threads:

Function Defining:

```
@everywhere function square(x)
    return x^2
end
```

The reason this is needed is because each worker thread, if using the square function needs to define it in its own memory space, as the workers cannot access function definitions from the master threads memory.

Variable Declaration:

```
@everywhere const N = 100
```

If N is used by all worker threads in a calculation, it needs to be shared across all worker threads with the @everywhere macro as otherwise the variable is not defined otherwise.

Package Importing:

```
@everywhere using LinearAlgebra
```

The reason for using everywhere on packages is for the same reason as the other two, the functions and consts defined in a package need to be initialized and for each worker thread