1. Initialization Progress

```
class InitMapper
  // Input: "EdgeID  FromNodeID  ToNodeID  Distance"
  method Map(edgeID, NodePairsWithDistance)
  // inverse the NodePairsWIthDistance, so we can convert single
  // target shortest path to single source target shortest path
    Emit(ToNodeID, "(FromNodeID, Distance)")
    Emit(FromNodeID, "(FromNodeID, 0)")


class InitReducer
  method Reducer(NodeID, [NeighbourNodesWithDistance])
    if NodeID == TargetID:
        NodeID.distance = 0
    else
        NodeID.distance = inf

    for all neighbour nodes with distance:
        NodeID.adjsPairs.append(NodeWithDistance)
    Emit(NodeID, distance + adjsPairs)
```

2. Iteration Progress

```
class STMMapper
  // Input: "ToNodeID  Distance (FromNode1, dis1), (FromNode2,
dis2), ..."
  // the distance is from TargetNode to ToNode, as we have
convert single
  // target shourtest path to single source target shortest path
  method Map(ToNodeID, distance + adjsPairs)
    // current node is unreacheble for TargetNode
    if ToNodeId.distance == "inf"
      return

    // re-calculate the distance from TargetNode to all nodes
    for all ToNodeID.neighbours:
      dis = distance from ToNodeID to NodeID
      Emit(NodeID, ToNodeID.distance + dis + adjsPairs)



class STMReducer
  // Input: "ToNodeID  Distance1 (FromNode1, dis1), (FromNode2,
dis2), ..."
  //        "ToNodeID  Distance2 (FromNode1, dis1), (FromNode2,
dis2), ..."
  method Reducer(NodeID, [NeighbourNodesWithDistance])
    // relax operation
    distance = Min(Distance1, Distance2, ... Distancen)

    Emit(NodeID, distance + adjsPairs)
```

3. Output Progress

```
class ResultMapper
  // Input: "ToNodeID  distance (FromNode1, dis1), (FromNode2, dis2), ..."
  method Map(ToNodeID, distance + adjsPairs)
    Emit(ToNodeID, distance)



class ResultReducer
  // Input: "ToNodeID  distance"
  method Reducer(ToNodeID, distance)
    if distance == "inf"
      return
    else
      Emit(TargetNode, ToNodeID + Distance)
```