Algorithm 1 Floyd-Warshall with path reconstruction

```
$dist \leftarrow |V| \times |V|$ length array of minimum distances initialized to $\infty$
$next \leftarrow |V| \times |V|$ length array of vertex indices initialized to **null**
**procedure** FLOYD-WARSHALL(PATH RECONSTRUCTION)
    **for each** edge(u, v) **do**
        $dist[u][v] \leftarrow w(u, v)$
        $next[u][v] \leftarrow v$
    **for** k = 1 to $|V|$ **do**
        **for** i = 1 to $|V|$ **do**
            **for** j = 1 to $|V|$ **do**
                **if** $dist[u][v] > dist[i][k] + dist[k][j]$ **then**
                    $dist[u][v] \leftarrow dist[i][k] + dist[k][j]$
                    $next[i][j] \leftarrow next[i][k]$
**procedure** GETPATH(u, v)
    **if** $next[u][v] =$ **null then**
        **return** []
    $path = [u]$
    **while** $u \neq v$ **do**
        $u \leftarrow next[u][v]$
        path.append(u)
    **return** path
```

Algorithm 2 TSP by Nearest Neighbour

```
V ← vertices representing all destinations in the route
**procedure** NEAREST NEIGHBOUR(**Vertex** P)
    sortedVertices ← $V \setminus \{P\}$
    result = $[P]$
    **while** $|sortedVertices| > 0$ **do**
        sortedVertices.sortRelativeTo(P)
        $P \leftarrow sortedVertices[0]$
        sortedVertices.remove(P)
        result.append(P)
    result.append(result[0])
    **return** result
```

Algorithm 3 First Fit Decreasing

```
Vehicles ← all available vehicles
Orders ← all orders to distribute
procedure FFD
    $sortVehiclesByCapacity(Vehicles)$
    $sortOrdersByCapacityDescending(Orders)$
    for each order ∈ Orders do
        for each vehicle ∈ Vehicles do
            if $vehicle.usedCapacity + order.packageNumber <= vehicle.maxCapacity$ then
                vehicle.addOrder(order)
                break
    Orders.clear()
```

Algorithm 4 Get Largest Strongly Connected Component

```
graph = (V, E)
procedure GetSCCsByKosaraju
    result ← vector of vectors of nodes
    nodeStack ← stack of nodes
    setAllNodesToNotVisited(graph)
    for each node ∈ graph.getNodes() do
        SCCVisit(node, nodeStack)
    transpose ← graph.getTranspose()
    setAllNodesToNotVisited(graph)
    while not nodeStack.empty() do
        node ← nodeStack.top()
        nodeStack.pop()
        if not node.isVisited() then
            result.add(transpose.DFS(node))
    return result

procedure SCCVisit(node, nodeStack)
    node.setVisitedToTrue()
    for each edge ∈ node.getEdges() do
        if not edge.getDestination().isVisited() then
            SSCVisit(edge.getDestination(), nodeStack)
    nodeStack.push(node)

procedure GetLargestSCC
    SCCs ← GetSCCsByKosaraju
    nodes ← maxVectorBySize(SCCs)
    return nodes
```

Algorithm 5 Shortest Path Between Two Points By A* Algorithm

```
procedure distMin(node1, node2)
    return staright line distance between node1 and node2 based on coordinates

procedure GetAStarPath(G = (V, E), srcNode, destNode)
    for each v ∈ V do
        dist(v) ← INF → Distance from the source node
        path(v) ← null → Path from the source node
        visited(v) ← false → Check if it is visited
    pqueue → Priority Queue of pairs of distances and nodes by ascending order of distance
    dist(srcNode) ← 0.0
    path(srcNode).push(srcNode)
    ENQUEUE(pqueue, pair(0.0, srcNode))
    while not pqueue.isEmptty() do
        node ← DEQUEUE(pqueue)
        visited(node) ← true
        if node = destNode then
            break
        for each edge ∈ node.getEdges() do
            dest ← edge.getDestination()
            weight ← edge.getWeight()
            distNext ← distMin(dest, destNode)
            distCurrent ← distMin(node, destNode)
            aStarHeuristic ← weight + distNext - distCurrent
            if not visited(dest) and dist(dest) ¿ dist(node) + weight + aStarHeuristic then
                dist(dest) ← dist(node) + weight + aStarHeuristic
                ENQUEUE(pqueue, pair(dist(dest), dest))
                path(dest) = path(node)
                path(dest).push(dest)
    return path(nodeDest)
```