Algorithm 1 Greedy Algorithm

```
1: function GREEDY()
2:     load ← 0
3:     unvisited ← [1, 2, ..., 2n]
4:     π ← [0]                                          ▷ π is a route (solution)
5:     while unvisited is not empty do
6:         candidates ← FILTERCANDIDATES(unvisited, π, load)
7:         nearestLoc ← GETNEARESTLOCATION(π[−1], candidates)
8:         append nearestLoc to π
9:         remove nearestLoc from unvisited
10:        if nearestLoc ≤ n then
11:            load ← load + 1
12:        else
13:            load ← load − 1
14:    append 0 to π
15:    return π

16: function FILTERCANDIDATES(list unvisited, list π, int load)
17:    candidates ← empty list
18:    if load < k then
19:        for nextLoc ∈ unvisited do
20:            if nextLoc ≤ n or (nextLoc − n) ∈ π then
21:                append nextLoc to candidates
22:    else
23:        for nextLoc ∈ unvisited do
24:            if (nextLoc − n) ∈ π then
25:                append nextLoc to candidates
26:    return candidates

27: function GETNEARESTLOCATION(int currentLoc, list candidates)
28:    nearestLoc ← −1
29:    minDist ← ∞
30:    for nextLoc ∈ candidates do
31:        if c_{currentLoc,nextLoc} < minDist then
32:            nearestLoc ← nextLoc
33:            minDist ← c_{currentLoc,nextLoc}
34:    return nearestLoc
```

Algorithm 2 Node relocation

```
1: function NodeRelocation()
2:     π ← Greedy()
3:     initialize routeIndex and loadList
4:     repeat
5:         for pickup ← 1 to n do
6:             i_pickup ← routeIndex[pickup]
7:             i_delivery ← routeIndex[pickup + n]
8:             Δ_best ← 0
9:             i_swap ← −1
10:            for i_cand ← 1 to i_delivery do
11:                if i_pickup ≠ i_cand then
12:                    Δ ← DeltaObjective(i_cand, i_pickup, π, routeIndex, loadList)
13:                    if Δ < Δ_best then
14:                        Δ_best ← Δ
15:                        i_swap ← i_cand
16:            if i_swap ≠ −1 then
17:                Swap(i_cand, i_pickup)
18:                update routeIndex and loadList
19:        for delivery ← n + 1 to 2n do
20:            i_delivery ← routeIndex[delivery]
21:            i_pickup ← routeIndex[delivery − n]
22:            Δ_best ← 0
23:            i_swap ← −1
24:            for i_cand ← i_pickup to 2n do
25:                if i_delivery ≠ i_cand then
26:                    Δ ← DeltaObjective(i_cand, i_delivery, π, routeIndex, loadList)
27:                    if Δ < Δ_best then
28:                        Δ_best ← Δ
29:                        i_swap ← i_cand
30:            if i_swap ≠ −1 then
31:                Swap(i_cand, i_delivery)
32:                update routeIndex and loadList
33:    until no more improvement found
34:    return π

35: function DeltaObjective(int i_c, int i, list π, list routeIndex, list loadList)
36:    if PrecedenceViolated(i_c, i, routeIndex) or CapacityViolated(i_c, i, loadList) then
37:        return ∞
38:    if |i_c − i| = 1 then                          ▷ check if i_c and i are next to each other
39:        i_c, i ← sorted(i_c, i)
40:        return c_{π[i_c−1],π[i]} + c_{π[i_c],π[i+1]} − c_{π[i_c−1],π[i_c]} − c_{π[i],π[i+1]}
41:    return c_{π[i_c−1],π[i]} + c_{π[i],π[i_c+1]} + c_{π[i−1],π[i_c]} + c_{π[i_c],π[i+1]}
           −c_{π[i_c−1],π[i_c]} − c_{π[i_c],π[i_c+1]} − c_{π[i−1],π[i]} − c_{π[i],π[i+1]}

42: function PrecedenceViolated(int i_c, int i, list π, list routeIndex)
43:    if π[i_c] ≤ n then                             ▷ check if node at i_c is a pickup node or not
44:        return i > routeIndex[π[i_c] + n]
45:    return i < routeIndex[π[i_c] − n]

46: function CapacityViolated(int i, int j, list π, list loadList)
47:    i, j ← sorted(i, j)
48:    x_i, x_j ← π[i], π[j]
49:    if (x_i ≤ n and x_j ≤ n) or (x_i > n and x_j > n) then
50:        return False
51:    if x_i ≤ n then
52:        for loc ← i to j − 1 do
53:            if loadList[loc] − 2 < 0 then
54:                return True
55:    else
56:        for loc ← i to j − 1 do
57:            if loadList[loc] + 2 > k then
58:                return True
59:    return False
```

Algorithm 3 Or-opt algorithm

function OR-OPT(int k_{Or}) ▷ k_{Or} is maximum block length
  $\pi \leftarrow$ GREEDY()
  initialize routeIndex and loadList
  cands $= [1, ..., 2n]$
  repeat
    SHUFFLE(cands)
    for cand $\in$ cands **do**
      move $\leftarrow$ empty tuple
      $s \leftarrow$ routeIndex[cand]
      for $e \leftarrow s$ to $\min(2n, s + k_{\text{Or}} - 1)$ **do**
        $\Delta_{\text{remove}} \leftarrow c_{\pi[s-1],\pi[e+1]} - c_{\pi[s-1],\pi[s]} - c_{\pi[e],\pi[e+1]}$
        $\Delta_{\text{minInsert}} \leftarrow -\Delta_{\text{remove}}$
        for $i \leftarrow s - 1$ to 1 **do**
          if $\pi[i] \leq n$ and $s \leq \text{routeIndex}[\pi[i] + n] \leq e$ **then**
            break
          if CAPACITYVIOLATED(s, e, i, π) **then**
            continue
          $\Delta_{\text{insert}} \leftarrow$ DELTAINSERTION(s, e, i, π)
          if $\Delta_{\text{minInsert}} > \Delta_{\text{insert}}$ **then**
            $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{insert}}$
            move $\leftarrow (s, e, i)$
        for $i \leftarrow e + 2$ to $2n + 1$ **do**
          if $\pi[i - 1] > n$ and $s \leq \text{routeIndex}[\pi[i - 1] - n] \leq e$ **then**
            break
          if CAPACITYVIOLATED(s, e, i, π) **then**
            continue
          $\Delta_{\text{insert}} \leftarrow$ DELTAINSERTION(s, e, i, π)
          if $\Delta_{\text{minInsert}} > \Delta_{\text{insert}}$ **then**
            $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{insert}}$
            move $\leftarrow (s, e, i)$
        if $\Delta_{\text{minInsert}} < 0$ **then break**
      if move is not empty **then** update π, routeIndex, loadList based on move
  until no more improvement found
  return π

function DELTAINSERTION(int s, int e, int i, list π)
  return $c_{\pi[i-1],\pi[s]} + c_{\pi[e],\pi[i]} - c_{\pi[i-1],\pi[i]}$

function CAPACITYVIOLATED(int s, int e, int i, list π)
  if $i > e$ **then**
    return CAPACITYVIOLATED$(e + 1, i - 1, s, \pi)$
  $\Delta_{\text{loadInside}} \leftarrow \text{loadList}[i - 1] - \text{loadList}[s - 1]$
  $\Delta_{\text{loadOutside}} \leftarrow 0$
  for $i_{\text{inside}} \leftarrow s$ to e **do**
    if $\text{loadList}[i_{\text{inside}}] + \Delta_{\text{loadInside}} > k$ **then**
      return True
    if $\pi[i_{\text{inside}}] > n$ **then**
      $\Delta_{\text{loadOutside}} \leftarrow \Delta_{\text{loadOutside}} - 1$
    else
      $\Delta_{\text{loadOutside}} \leftarrow \Delta_{\text{loadOutside}} + 1$
  if $\max(\text{loadList}[i, ..., s - 1]) + \Delta_{\text{loadOutside}} > k$ **then**
    return True
  return False

Algorithm 4 Pair relocation

function PairRelocation()
  $\beta \leftarrow$ Greedy()
  repeat
    cands = Shuffle($[1, ..., n]$)
    for x $\in$ cands **do**
      $\Delta_{\text{remove}} \leftarrow$ CalculateRemovalCost(β, x)
      $\Delta_{\text{minInsert}} \leftarrow -\Delta_{\text{remove}}$
      $i, j \leftarrow$ index of $x, x+n$ in β
      $\alpha \leftarrow \beta$ after removing x and $x+n$
      action $\leftarrow (i-1, j-2)$ ▷ action if no improvement found
      maxCapIdx, curMaxCap $\leftarrow$ CalculateCapacity(α)
      $\Delta_{\text{consec}}, \text{a}_{\text{consec}} \leftarrow$ ConsecutiveInsertion(α, x, maxCapIdx, curMaxCap)
      if $\Delta_{\text{consec}} < \Delta_{\text{minInsert}}$ **then**
        $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{consec}}$
        action $\leftarrow \text{a}_{\text{consec}}$
      $\Delta_{\text{nonConsec}}, \text{a}_{\text{nonConsec}} \leftarrow$ NonConsecutiveInsertion(α, x, maxCapIdx, curMaxCap)
      if $\Delta_{\text{nonConsec}} < \Delta_{\text{minInsert}}$ **then**
        $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{nonConsec}}$
        action $\leftarrow \text{a}_{\text{nonConsec}}$
      $\beta \leftarrow$ ApplyInsertion(α, action)
  until no more improvement found
  return β

function CalculateCapacity(list α)
  curCap $\leftarrow 0$
  maxCap $\leftarrow 0$
  for $i \leftarrow 1$ to len(α) $- 1$ **do**
    if $\alpha[i] \leq n$ **then**
      curCap $\leftarrow$ curCap $+ 1$
      if curCap $>$ maxCap **then**
        ids $\leftarrow [i]$
        maxCap $\leftarrow$ curCap
      else if curCap $=$ maxCap **then**
        append i to ids
    else
      curCap $\leftarrow$ curCap $- 1$
  append ∞ to ids ▷ for boundary cases
  return (maxCap, ids)

function CalculateRemovalCost(list β, int x)
  if $j - i = 1$ **then**
    $\Delta_{\text{remove}} \leftarrow c_{\beta[i-1],\beta[j+1]} - c_{\beta[i-1],\beta[i]} - c_{\beta[i],\beta[j]} - c_{\beta[j],\beta[j+1]}$
  else
    $\Delta_{\text{remove}} \leftarrow c_{\beta[i-1],\beta[i+1]} - c_{\beta[i-1],\beta[i]} - c_{\beta[i],\beta[i+1]} + c_{\beta[j-1],\beta[j+1]} - c_{\beta[j-1],\beta[j]} - c_{\beta[j],\beta[j+1]}$
  return Δ_{remove}

function ConsecutiveInsertion(list α, int x, list maxCapIdx, int curMaxCap)
  $p \leftarrow 0$
  $\Delta_{\text{minInsert}} \leftarrow \infty$
  action $\leftarrow$ None
  for $i' \leftarrow 0$ to $2n - 2$ **do**
    if $i' =$ maxCapIdx[p] **and** curMaxCap $\geq k$ **then**
      $p \leftarrow p + 1$
      continue
    $\Delta_{\text{insert}} \leftarrow c_{\alpha[i'],x} + c_{x,x+n} + c_{x+n,\alpha[i'+1]} - c_{\alpha[i'],\alpha[i'+1]}$
    if $\Delta_{\text{insert}} < \Delta_{\text{minInsert}}$ **then**
      $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{insert}}$
      action $\leftarrow (i', i')$
  return $\Delta_{\text{minInsert}}$, action

function NonConsecutiveInsertion(list α, int x, list maxCapIdx, int curMaxCap)
  $p \leftarrow 0$
  $\Delta_{\text{minInsert}} \leftarrow \infty$
  action $\leftarrow$ None
  for $i' \leftarrow 0$ to $2n - 1$ **do**
    if curMaxCap $< k$ **then** ▷ If the bus does not reach max capacity
      for $j' \leftarrow i' + 1$ to $2n - 2$ **do**
        $\Delta_{\text{insert}} \leftarrow c_{\alpha[i'],x} + c_{x,\alpha[i'+1]} - c_{\alpha[i'],\alpha[i'+1]} + c_{\alpha[j'],x} + c_{x,\alpha[j'+1]} - c_{\alpha[j'],\alpha[j'+1]}$
        if $\Delta_{\text{insert}} < \Delta_{\text{minInsert}}$ **then**
          $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{insert}}$
          action $\leftarrow (i', j')$
    else
      if $i' =$ maxCapIdx[p] **then**
        $p \leftarrow p + 1$
        continue
      for $j' \leftarrow i' + 1$ to min($2n - 2$, maxCapIdx[p] $- 1$) **do**
        $\Delta_{\text{insert}} \leftarrow c_{\alpha[i'],x} + c_{x,\alpha[i'+1]} - c_{\alpha[i'],\alpha[i'+1]} + c_{\alpha[j'],x} + c_{x,\alpha[j'+1]} - c_{\alpha[j'],\alpha[j'+1]}$
        if $\Delta_{\text{insert}} < \Delta_{\text{minInsert}}$ **then**
          $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{insert}}$
          action $\leftarrow (i', j')$
  return $\Delta_{\text{minInsert}}$, action