Fundamentals of optimization

Mini-project: **CBUS**

Table of contents

01

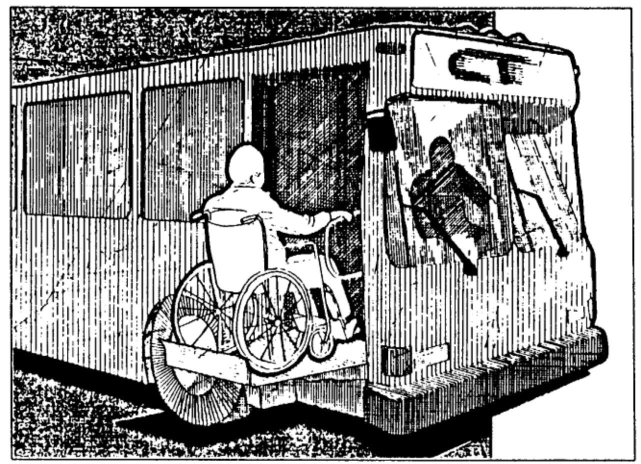
Introduction

Introduction

- Problem: **CBUS**
  - Design a route for a bus that must **pick up and drop off** a set of passengers
  - The bus must start and end its route from a **depot**
  - The bus has a **maximum capacity** that cannot be exceeded
  - Objective: **minimizing the total distance traveled**

Introduction

- Some application: transportation of the disabled and elderly, food delivery, etc.



02

Modelling

Modelling

- n: number of passengers/pickup – delivery pairs → number of nodes/locations: $2n + 1$
- k: capacity/max. load of the bus/vehicle
- $P = \{1, \dots, n\}$ and $D = \{n+1, \dots, 2n\}$: pickup and delivery nodes/locations, respectively
  - 0: origin/depot/start location of the bus
  - $2n + 1$: destination/end location of the bus (same geographical location as the depot)
  - Each pair $(i, i+n), i = \overline{1, n}$ represents a passenger request
  - $V = P \cup D$: set of possible locations (except the depot)
  - E: set of possible trips/edges; $(i, j) \in E$ means the bus travels from i to j
- c: 2-D array; $c_{i,j}$ is the travel cost/distance from node i to node j
- $x_{i,j}$: binary flow variable, $\begin{cases} x_{i,j} = 1: \text{ the bus travels on edge } (i,j) \in E \\ x_{i,j} = 0: \text{ otherwise} \end{cases}$

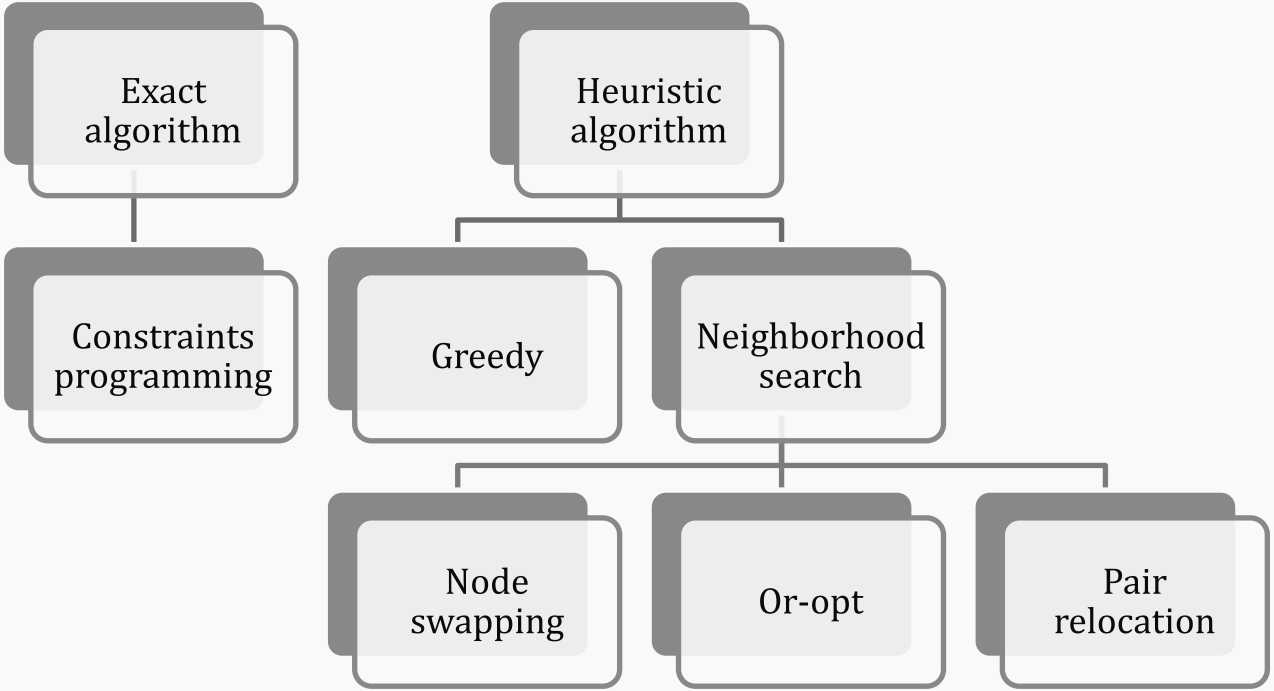
$$\Rightarrow \textbf{Objective}: \min \sum_{(i,j) \in E} c_{i,j} * x_{i,j}$$

❖ *The terms above may be used interchangeably throughout the presentation*

03

Proposed approaches

Proposed approaches



Constraints programming (**CP**)

- Two additional variables are introduced for the CP problem:
    - For the capacity constraint: l_i – the load of the bus after leaving node i
        - Denote: l_i++ after leaving a node $i \in P$, l_i-- after leaving a node $i \in D$
    - For the precedence constraint: t_i – the time of arrival of the bus at node i
- With that, we have a CP model as below:

$$\text{minimize} \quad \sum_{(i,j)\in E} c_{i,j} * x_{i,j}$$

$$\text{subject to} \quad \sum_{j\in V\cup\{2n+1\}} x_{k,j} = \sum_{i\in\{0\}\cup V} x_{i,k} = 1, \qquad \forall\, k \in V$$

(each node is visited exactly once)

$$\sum_{j\in V} x_{0,j} = \sum_{i\in V} x_{i,2n+1} = 1$$

(the bus must go to a pickup node after leaving start node; the same idea for the end node)

$$x_{i,j} \in \{0, 1\}$$

(binary variable)

Constraints programming (**CP**)

$$t_0 = 0 \qquad \text{(time at start node)}$$

$$t_{2n+1} = 2n + 1 \qquad \text{(time at end node)}$$

$$t_i < t_{i+n}, \qquad \forall i \in P \qquad \text{(precedence constraint)}$$

$$x_{i,j}\left(t_i + 1 - t_j\right) = 0, \qquad \forall (i,j) \in E \qquad \text{(change in time after going through a node)}$$

$$0 \le t_i \le 2n + 1, \qquad t_i \in \mathbb{N} \qquad \text{(time range)}$$

$$l_0 = l_{2n+1} = 0 \qquad \text{(load at depot)}$$

$$0 \le l_i \le k, \qquad l_i \in \mathbb{N} \qquad \text{(load range)}$$

$$x_{i,j}\left(l_i + d_j - l_j\right) = 0, \qquad \text{(change in load after going through a node)}$$

$$\text{in which } d_i = \begin{cases} 0, \text{if } i = 0 \text{ or } i = 2n+1 \\ 1, \text{if } i \in P \\ -1, \text{if } i \in D \end{cases}$$

Greedy

- Main idea: starting from the depot, the bus iteratively **choose the nearest node that does not violate constraints** to travel to
- Update *load* variable (current load of the bus) accordingly

Algorithm 1 Greedy Algorithm

```
function GREEDY()
    load ← 0
    unvisited ← [1, 2, ..., 2n]
    π ← [0]                                         ▷ π is a route (solution)
    while unvisited is not empty do
        candidates ← FILTERCANDIDATES(unvisited, π, load)
        nearestLoc ← GETNEARESTLOCATION(π[−1], candidates)
        append nearestLoc to π
        remove nearestLoc from unvisited
        if nearestLoc ≤ n then
            load ← load + 1
        else
            load ← load − 1
    append 0 to π
    return π
```

Greedy

- Candidates for the next location are chosen bases on the value of the *load* variable:
  - If the max capacity is reached ($load = k$), only chooses delivery nodes that has their pickup node visited
  - Otherwise, chooses the unvisited pickup nodes too
- Time complexity: $O(n^2)$

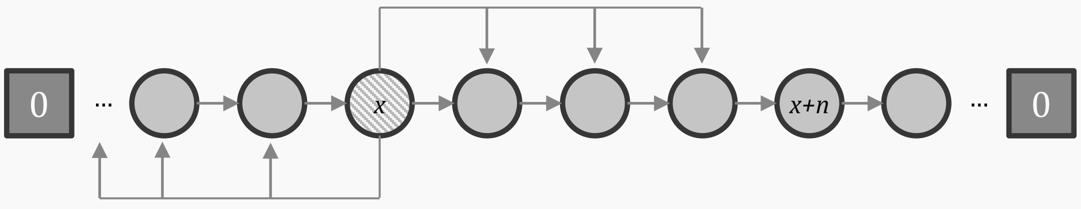
```
function FilterCandidates(list unvisited, list π, int load)
    candidates ← empty list
    if load < k then
        for nextLoc ∈ unvisited do
            if nextLoc ≤ n or (nextLoc − n) ∈ π then
                append nextLoc to candidates
    else
        for nextLoc ∈ unvisited do
            if (nextLoc − n) ∈ π then
                append nextLoc to candidates
    return candidates

function GetNearestLocation(int currentLoc, list candidates)
    nearestLoc ← −1
    minDist ← ∞
    for nextLoc ∈ candidates do
        if c_{currentLoc,nextLoc} < minDist then
            nearestLoc ← nextLoc
            minDist ← c_{currentLoc,nextLoc}
    return nearestLoc
```

Neighborhood search

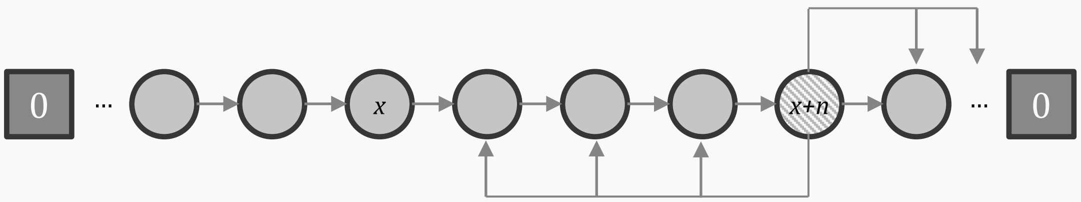
- Starts with a feasible solution, and iteratively improves it until local optimum is reached
  - Neighborhood is built on a *move* operator, which depends on the algorithm
  - Iteratively explores the neighborhood to find an improved one
  - Replaces current solution with this solution
  - Repeats the process until no further improvement exists
- Strategy: first find a starting solution by using greedy algorithm, then continue improving the solution using local search

Node swapping

- Idea: iteratively move a pickup/delivery node to a appropriate location by **swapping** with another node in the route



Move a pickup node



Move a delivery node

Node swapping

- Idea: iteratively move a pickup/delivery node to a appropriate location by **swapping** with another node in the route

- Create separate lists to store frequently accessed data (routeIndex, loadList)

- Time complexity: $O(n^2)$

Algorithm 2 Node relocation

```
function NodeRelocation()
    π ← Greedy()
    initialize routeIndex and loadList
    repeat
        for pickup ← 1 to n do
            i_pickup ← routeIndex[pickup]
            i_delivery ← routeIndex[pickup + n]
            Δ_best ← 0
            i_swap ← −1
            for i_cand ← 1 to i_delivery do
                if i_pickup ≠ i_cand then
                    Δ ← DeltaObjective(i_cand, i_pickup, π, routeIndex, loadList)
                    if Δ < Δ_best then
                        Δ_best ← Δ
                        i_swap ← i_cand
            if i_swap ≠ −1 then
                Swap(i_cand, i_pickup)
                update routeIndex and loadList
        for delivery ← n + 1 to 2n do
            i_delivery ← routeIndex[delivery]
            i_pickup ← routeIndex[delivery − n]
            Δ_best ← 0
            i_swap ← −1
            for i_cand ← i_pickup to 2n do
                if i_delivery ≠ i_cand then
                    Δ ← DeltaObjective(i_cand, i_delivery, π, routeIndex, loadList)
                    if Δ < Δ_best then
                        Δ_best ← Δ
                        i_swap ← i_cand
            if i_swap ≠ −1 then
                Swap(i_cand, i_delivery)
                update routeIndex and loadList
    until no more improvement found
    return π
```

Node swapping

- Calculate change of objective value (route cost) if i and i_c are swapped

function DeltaObjective(int i_c, int i, list π, list routeIndex, list loadList)
    if PrecedenceViolated(i_c, i, routeIndex) or CapacityViolated(i_c, i, loadList) **then**
        return ∞
    if $|i_c - i| = 1$ **then** $\triangleright$ check if i_c and i are next to each other
        $i_c, i \leftarrow$ sorted(i_c, i)
        return $c_{\pi[i_c-1],\pi[i]} + c_{\pi[i_c],\pi[i+1]} - c_{\pi[i_c-1],\pi[i_c]} - c_{\pi[i],\pi[i+1]}$
    return $c_{\pi[i_c-1],\pi[i]} + c_{\pi[i],\pi[i_c+1]} + c_{\pi[i-1],\pi[i_c]} + c_{\pi[i_c],\pi[i+1]}$
        $- c_{\pi[i_c-1],\pi[i_c]} - c_{\pi[i_c],\pi[i_c+1]} - c_{\pi[i-1],\pi[i]} - c_{\pi[i],\pi[i+1]}$

Node swapping

- Precedence violation:

function PRECEDENCEVIOLATED(int i_c, int i, list π, list routeIndex)
    if $\pi[i_c] \leq n$ **then** ▷ check if node at i_c is a pickup node or not
        return $i >$ routeIndex$[\pi[i_c] + n]$
    return $i <$ routeIndex$[\pi[i_c] - n]$

Node swapping

- Capacity violation:

```
function CapacityViolated(int i, int j, list π, list loadList)
    i, j ← sorted(i, j)
    x_i, x_j ← π[i], π[j]
    if (x_i ≤ n and x_j ≤ n) or (x_i > n and x_j > n) then
        return False
    if x_i ≤ n then
        for loc ← i to j − 1 do
            if loadList[loc] − 2 < 0 then
                return True
    else
        for loc ← i to j − 1 do
            if loadList[loc] + 2 > k then
                return True
    return False
```

Node swapping

- Capacity violation:
  - If node at i and j are both pickup (delivery) node, then swapping them will not change the load of the route

```
function CapacityViolated(int i, int j, list π, list loadList)
    i, j ← sorted(i, j)
    x_i, x_j ← π[i], π[j]
    if (x_i ≤ n and x_j ≤ n) or (x_i > n and x_j > n) then
        return False
    if x_i ≤ n then
        for loc ← i to j − 1 do
            if loadList[loc] − 2 < 0 then
                return True
    else
        for loc ← i to j − 1 do
            if loadList[loc] + 2 > k then
                return True
    return False
```

$n = 3, k = 2$:

	0	1	2	3	4	5	6	7
α	0	1	4	3	2	5	6	0
l	0	1	0	1	2	1	0	0

$\xrightarrow[\text{2 with 5}]{\text{swap index}}$

	0	1	2	3	4	5	6	7
α	0	1	5	3	2	4	6	0
l	0	1	0	1	2	1	0	0

Node swapping

- Capacity violation:
  - Else, if node at i is a pickup node, swapping i with j will **decrease** load of nodes from i to $j-1$ by 2

```
function CapacityViolated(int i, int j, list π, list loadList)
    i, j ← sorted(i, j)
    x_i, x_j ← π[i], π[j]
    if (x_i ≤ n and x_j ≤ n) or (x_i > n and x_j > n) then
        return False
    if x_i ≤ n then
        for loc ← i to j − 1 do
            if loadList[loc] − 2 < 0 then
                return True
    else
        for loc ← i to j − 1 do
            if loadList[loc] + 2 > k then
                return True
    return False
```

$n = 3, k = 2$:

	0	1	2	3	4	5	6	7
α	0	1	4	3	2	5	6	0
l	0	1	0	1	2	1	0	0

$\xrightarrow[\text{3 with 6}]{\text{swap index}}$

	0	1	2	3	4	5	6	7
α	0	1	4	6	2	5	3	0
l	0	1	0	-1	0	-1	0	0

Node swapping

- Capacity violation:
  - Else, swapping i with j will **increase** load of nodes from i to $j-1$ by 2

function CAPACITYVIOLATED(int i, int j, list π, list loadList)
    $i, j \leftarrow$ SORTED(i, j)
    $x_i, x_j \leftarrow \pi[i], \pi[j]$
    if $(x_i \leq n$ and $x_j \leq n)$ or $(x_i > n$ and $x_j > n)$ **then**
        return False
    if $x_i \leq n$ **then**
        for loc $\leftarrow i$ to $j-1$ **do**
            if loadList[loc] $- 2 < 0$ **then**
                return True
▶   **else**
        for loc $\leftarrow i$ to $j-1$ **do**
            if loadList[loc] $+ 2 > k$ **then**
                return True
    return False

$n = 3, k = 2$:

	0	1	2	3	4	5	6	7
α	0	1	4	3	2	5	6	0
l	0	1	0	1	2	1	0	0

$\xrightarrow[\text{2 with 4}]{\text{swap index}}$

	0	1	2	3	4	5	6	7
α	0	1	2	3	4	5	6	0
l	0	1	2	3	2	1	0	0

Or-opt

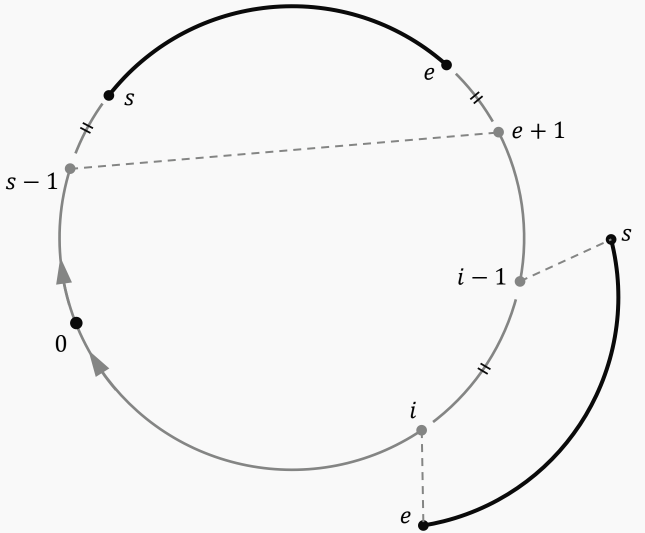
- Proposed by Ilhan Or (1976)
- A **restricted** version of 3-opt
- **Segment/block of nodes** in a route is relocated to a different position
- Modified to include precedence and capacity constraints

Or-opt

Algorithm 3 Or-opt algorithm

function OR-OPT(int k_{Or}) $\triangleright$ 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

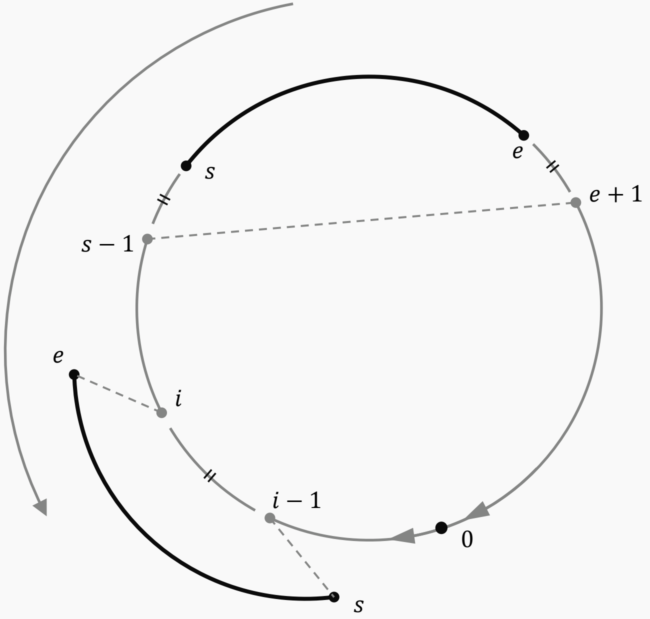
- s, e: start and end index of the block
- i: insert the block **before** index i in the old route
- k_{Or}: maximum block length
- Time complexity: $O(k_{Or} * n^2)$



Or-opt

- Insert **before** the old location (move backward):

```
for i ← s − 1 to 1 do
    if π[i] ≤ n and s ≤ routeIndex[π[i] + n] ≤ e then
        break
    if CapacityViolated(s, e, i, π) then
        continue
    Δ_insert ← DeltaInsertion(s, e, i, π)
    if Δ_minInsert > Δ_insert then
        Δ_minInsert ← Δ_insert
        move ← (s, e, i)
```

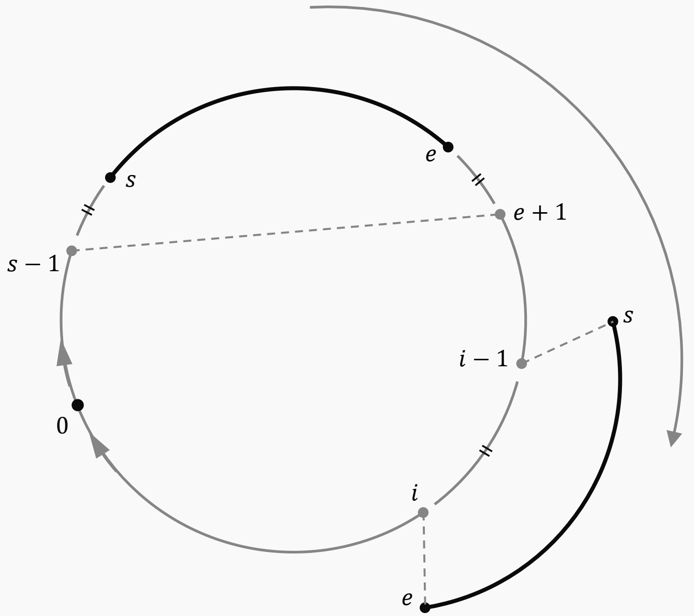


- After the move, node at i got moved **after** the block
  ⇒ have to check precedence violation for node at i

Or-opt

- Insert **after** the old location (move forward):

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 $\textsc{CapacityViolated}(s, e, i, \pi)$ **then**
        continue
    $\Delta_{\text{insert}} \leftarrow \textsc{DeltaInsertion}(s, e, i, \pi)$
    if $\Delta_{\text{minInsert}} > \Delta_{\text{insert}}$ **then**
        $\Delta_{\text{minInsert}} \leftarrow \Delta_{\text{insert}}$
        move $\leftarrow (s, e, i)$



- After the move, node at $i-1$ got moved **before** the block
  $\Rightarrow$ have to check precedence violation for node at $i-1$

Or-opt

- Calculating insertion cost:

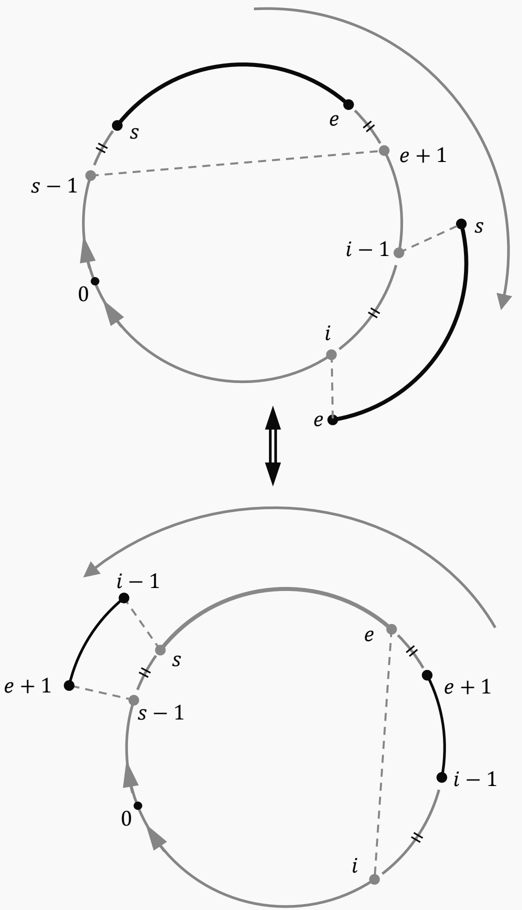
$$
\begin{array}{l}
\textbf{function } \textsc{DeltaInsertion}(\text{int } s, \text{ int } e, \text{ int } i, \text{ list } \pi) \\
\qquad \textbf{return } c_{\pi[i-1],\pi[s]} + c_{\pi[e],\pi[i]} - c_{\pi[i-1],\pi[i]}
\end{array}
$$

Or-opt

- Capacity constraints - move evaluation:
  - Every move operation that move the block forward is **equivalent** to a backward move:

$$(s, e, i) \equiv (e+1, i-1, s), \text{ if } i > e$$

```
function CapacityViolated(int s, int e, int i, list π)
    if i > e then
        return CapacityViolated(e + 1, i − 1, s, π)
    Δ_loadInside ← loadList[i − 1] − loadList[s − 1]
    Δ_loadOutside ← 0
    for i_inside ← s to e do
        if loadList[i_inside] + Δ_loadInside > k then
            return True
        if π[i_inside] > n then
            Δ_loadOutside ← Δ_loadOutside − 1
        else
            Δ_loadOutside ← Δ_loadOutside + 1
    if max(loadList[i, ..., s − 1]) + Δ_loadOutside > k then
        return True
    return False
```



Or-opt

	0	1	2	3	4	5	6	7	8	9	10	11
α	0	1	2	6	7	5	10	3	4	8	9	0
l	0	1	2	1	0	1	0	1	2	1	0	0

	0	1	2	5	6	7	3	4	8	9	10	11
α	0	1	2	5	10	3	6	7	4	8	9	0
l	0	1	2	3	2	3	2	1	2	1	0	0

unchanged | changed | unchanged

```
function CapacityViolated(int s, int e, int i, list π)
    if i > e then
        return CapacityViolated(e + 1, i − 1, s, π)
    Δ_loadInside ← loadList[i − 1] − loadList[s − 1]
    Δ_loadOutside ← 0
    for i_inside ← s to e do
        if loadList[i_inside] + Δ_loadInside > k then
            return True
        if π[i_inside] > n then
            Δ_loadOutside ← Δ_loadOutside − 1
        else
            Δ_loadOutside ← Δ_loadOutside + 1
    if max(loadList[i, ..., s − 1]) + Δ_loadOutside > k then
        return True
    return False
```

Explanation:

$n = 5$

$\text{move} = (s, e, i) = (5, 7, 3)$

$\Delta_{\text{loadInside}} = \text{loadList}[2] - \text{loadList}[4]$
$= 2 - 0 = 2$

Because the block has 2 node smaller than n, and 1 node greater than n

$\Rightarrow \Delta_{\text{loadOutside}} = 2 - 1 = 1$

Pair relocation

- Idea: relocating a pickup and delivery pair $(x, x+n)$ from its current positions (i, j) to new positions (i', j'), such that $i' < j'$
- Time complexity: $O(n^2)$

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 $\leftarrow (i-1, j-2)$ $\triangleright$ 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

Pair relocation - constraints

- Constraints:
  - Due to how the pairs are getting relocated ⇒ precedence constraints never get violated
  - Capacity constraints: make a list to store nodes that has highest load, then choose insertion locations based on the information

```
function CalculateCapacity(list α)
    curCap ← 0
    maxCap ← 0
    for i ← 1 to len(α) − 1 do
        if α[i] ≤ n then
            curCap ← curCap + 1
            if curCap > maxCap then
                ids ← [i]
                maxCap ← curCap
            else if curCap = maxCap then
                append i to ids
        else
            curCap ← curCap − 1
    append ∞ to ids                                  ▷ for boundary cases
    return (maxCap, ids)
```

❖ α: the route **after** remove $(x, x+n)$

Pair relocation - constraints

- Example: $n = 5, k = 3, (x,\ x+n) = (4, 9)$

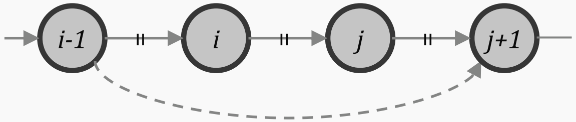
	0	1	2	3	4	5	6	7	8	9
α	0	1	2	**5**	10	**3**	6	7	8	0
l	0	1	2	**3**	2	**3**	2	1	0	0

$\Rightarrow \text{maxCap} = 3,\ \text{ids} = [3, 5, \infty]$

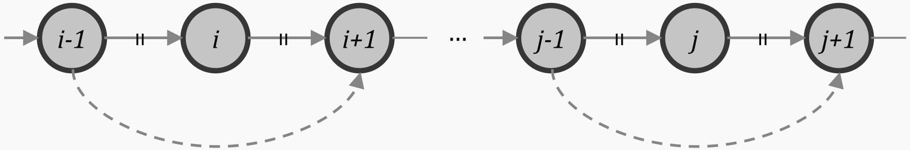
Pair relocation – removal cost

- Calculate removal cost – two cases:

  - i is next to j



  - i and j are not next to each other



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}

❖ β: the route **before** remove $(x, x+n)$

Pair relocation – consecutive insertion

- $n = 5, k = 3, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	5	10	3	6	7	8	0
l	0	1	2	**3**	2	**3**	2	1	0	0

Pair relocation – consecutive insertion

- $n = 5, k = 3, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	5	10	3	6	7	8	0
l	0	1	2	**3**	2	**3**	2	1	0	0

$\text{maxCap} = 3$

$\text{maxCapIds} = [3, 5, \infty]$

- Observations:
  - We only care about the load of $(x, x + n)$, since the load at other nodes are unchanged.

Pair relocation – consecutive insertion

- $n = 5, k = 3, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	5	10	3	6	7	8	0
l	0	1	2	**3**	2	**3**	2	1	0	0

$\text{maxCap} = 3$

$\text{maxCapIds} = [3, 5, \infty]$

- Observations:
  - Cannot insert the pair **immediately** after index 3 or 5 (indices in maxCapIds):

	0	1	2	3			4	5	6	7	8	9
α	0	1	2	5	4	9	10	3	6	7	8	0
l	0	1	2	**3**	**4**	3	2	**3**	2	1	0	0

Pair relocation – consecutive insertion

- $n = 5, k = 3, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	5	10	3	6	7	8	0
l	0	1	2	**3**	2	**3**	2	1	0	0

$\text{maxCap} = 3$
$\text{maxCapIds} = [3, 5, \infty]$

- Observations:
  - Any other locations are fine:

	0	1	2	3	4			5	6	7	8	9
α	0	1	2	5	10	4	9	3	6	7	8	0
l	0	1	2	**3**	2	**3**	2	**3**	2	1	0	0

Pair relocation – consecutive insertion

- $n = 5, k = 3, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	5	10	3	6	7	8	0
l	0	1	2	**3**	2	**3**	2	1	0	0

maxCap $= 3$

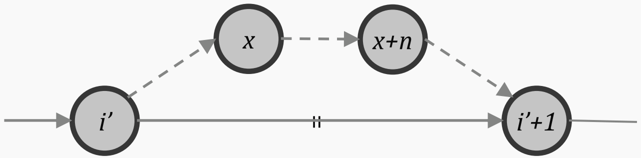
maxCapIds $= [3, 5, \infty]$

- Observations:
  - If the maximum capacity is not reached, the pair can be freely inserted.

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	6	7	5	3	8	10	0
l	0	1	2	1	0	1	2	1	0	0

Pair relocation – consecutive insertion

- Find the best insertion location in $O(n)$ time



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 $i' = \text{maxCapIdx}[p]$ **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

Pair relocation – non-consecutive insertion

- $n = 5, k = 2, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	6	7	5	3	8	10	0
l	0	1	**2**	1	0	1	**2**	1	0	0

Pair relocation – non-consecutive insertion

- $n = 5, k = 2, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	6	7	5	3	8	10	0
l	0	1	**2**	1	0	1	**2**	1	0	0

$\text{maxCap} = 2$
$\text{maxCapIds} = [2, 6, \infty]$

- Observations:
  - Non-consecutive insertion has some similarities with consecutive insertion:
    - If the max capacity is not reached, the pair can be freely inserted.
    - Pickup node cannot be immediately inserted after a index in maxCapIds (2, 6 in this case)

Pair relocation – non-consecutive insertion

- $n = 5, k = 2, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	6	7	5	3	8	10	0
l	0	1	**2**	1	0	1	**2**	1	0	0

$\text{maxCap} = 2$

$\text{maxCapIds} = [2,6,\infty]$

- Observations:
  - The load of nodes between x and $x + n$ are **increased by 1.**

  $\Rightarrow$ the pair must be inserted inside between **two consecutive indices** in maxCapIds:

	0	1	2	3		4	5	6	7	8		9
α	0	1	2	6	4	7	5	3	8	10	9	0
l	0	1	**2**	1	2	1	2	**3**	2	1	0	0

Pair relocation – non-consecutive insertion

- $n = 5, k = 2, (x,\ x + n) = (4, 9)$

	0	1	2	3	4	5	6	7	8	9
α	0	1	2	6	7	5	3	8	10	0
l	0	1	**2**	1	0	1	**2**	1	0	0

maxCap $= 2$

maxCapIds $= [2,6,\infty]$

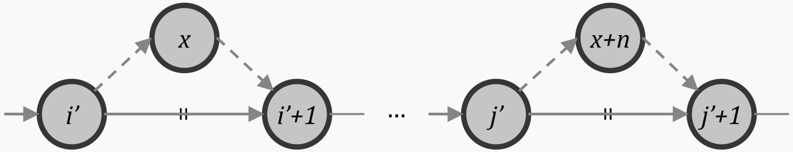
- Observations:
  - The load of nodes between x and $x + n$ are **increased by 1.**

  $\Rightarrow$ the pair must be inserted inside between **two consecutive indices** in maxCapIds:

	0	1	2	3		4	5		6	7	8	9
α	0	1	2	6	4	7	5	9	3	8	10	0
l	0	1	**2**	1	2	1	2	1	**2**	1	0	0

Pair relocation – non-consecutive insertion

- Find the best insertion location in $O(n^2)$ time



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 $\min(2n-2, \text{maxCapIdx[p]} - 1)$ **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

04

Experimental result

Experimental results

- The experiments were conducted on a 2.30 GHz Intel i7-12700H CPU with 16 GB RAM, and were implemented in Python 3.11
- Two types of instances: instances from online submission platform, and randomly generated data

Instance		CP				Greedy			
n	k	min_c	max_c	avg_c	avg_t (s)	min_c	max_c	avg_c	avg_t (s)
5	3	37	37	37	0.0539	49	49	49	1.62×10^{-5}
10	6	38	38	38	1.45	41	41	41	3.52×10^{-5}
100	40	----	----	----	> 300	144	144	144	0.00602
500	40	----	----	----	> 300	6552	6552	6552	0.593
1000	40	----	----	----	> 300	12176	12176	12176	5.11

Instance		Node swapping				Or-opt (iter = 5, k_{Or} = 5)				Pair relocation			
n	k	min_c	max_c	avg_c	avg_t (s)	min_c	max_c	avg_c	avg_t (s)	min_c	max_c	avg_c	avg_t (s)
5	3	44	44	44	7.69×10^{-5}	**37**	41	**38.96**	0.000439	39	43	40.568	8.67×10^{-5}
10	6	40	40	40	0.000275	**38**	38	**38**	0.00224	**38**	38	**38**	0.000349
100	40	140	140	140	0.0331	125	130	**127.32**	0.465	**123**	131	127.45	0.216
500	40	6363	6363	6363	1.27	5528	5664	5595.05	11.2	**5135**	5563	**5387.3**	7.99
1000	40	11732	11732	11732	7.42	10306	10486	10417.1	52.6	**9335**	10059	**9680.4**	36.4

Comparison of the approaches on instances from online submission platform

Experimental results

- Random instances generation:
  - $c_{i,i} = 0$
  - $c_{i,j}$ is a random integer in $[1, 3n]$
  - $c_{i,j} = c_{j,i}$
- Two cases: big and small value of k (corresponding to more constrainted or more relaxed capacity constraints)

Instance		$k = n/50$							
		Greedy		Node swapping		Or-opt (iter = 5, k_{Or} = 5)		Pair relocation	
n	k	avg_c	avg_t	avg_c	avg_t	avg_c	avg_t	avg_c	avg_t
50	1	100	1	100	2.13	**97.2**	78.1	99.5	**1.99**
100	2	100	1	98.8	2.55	**85.7**	272	86.7	**2.19**
200	4	100	1	98.6	1.99	**84.9**	183	91.9	**1.79**
500	10	100	1	98.4	**1.53**	**90.9**	10.5	95.3	1.77
1000	20	100	1	97.9	**1.41**	**95.2**	7.12	96.9	1.50

Instance		$k = n/5$							
		Greedy		Node swapping		Or-opt (iter = 5, k_{Or} = 5)		Pair relocation	
n	k	avg_c	avg_t	avg_c	avg_t	avg_c	avg_t	avg_c	avg_t
50	10	100	1	80.1	**9.58**	**73.4**	53.4	85.5	13.1
100	20	100	1	87.9	**4.44**	**82.4**	44.1	90.5	11.3
200	40	100	1	100	**2.09**	**85.6**	37.5	96.9	16.2
500	100	100	1	97.6	**2.47**	**89.3**	35.6	95.8	15.0
1000	200	100	1	93.5	**2.56**	**85.2**	30.7	94.6	17.3

Comparison of the approaches on randomly generated instances (relative to greedy method)

Experimental results

- Conclusion:
  - **Constraints programming**: gives optimal results, but has a very long run time, and fails to solve when $n > 25$
  - **Greedy**: provides fast results with acceptable route cost, suitable for generating first feasible solution for neighborhood search algorithms
  - **Node swapping**: fast solving time, but do not have much improvement over greedy method
  - **Or-opt**: gives the best results among the heuristic methods (especially when k is large), but has long run time
  - **Pair relocation**: good balance between speed and efficiency; fast solving time k is small

Thanks!

Do you have any questions?

For more information, please visit
https://github.com/thanh309/Mini_Project_CBUS